UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Programa de Maestría y Doctorado en Ingeniería

Diseño de la arquitectura de un procesador neuronal y su implementación en un FPGA.

T E S I S

QUE PARA OBTENER EL GRADO DE:
MAESTRO EN INGENIERÍA
PRESENTA

JOSÉ RAMÓN GUZMÁN MOSQUEDA

Asesor: Dr. Jesús Savage Carmona

MÉXICO, D.F. MAYO de 2005.

Agradecimientos

Un agradecimiento especial para mis padres *Enrique* y *Martha* quienes me han apoyado a lo largo de toda mi trayectoria académica y mi vida personal.

A mis hermanos *Susana, Rodrigo, Rocio Carolina, Enrique Manuel y Erick Mario.*

A la **Universidad Nacional Autónoma de México** quien siempre me ha dado un lugar para seguir adelante.

A mi tutor *Jesús Savage Carmona* quien hizo posible este trabajo.

A mis familiares y amigos que ocupan una lista bastante extensa pero que en su momento me brindaron su apoyo y lo siguen haciendo hasta este momento.

Al CONACyT por haberme brindado su apoyo económico y así hacer posible mis estudios a nivel maestría.

ÍNDICE

INTRODUCCIÓN

Una de las tareas del laboratorio de biorobótica de la Facultad de Ingeniería de la UNAM es el estudio de distintos métodos de inteligencia artificial y sus aplicaciones en robots móviles y agentes inteligentes. El objetivo es que los robots efectúen tareas como navegación dentro de un ambiente, exploración en entornos desconocidos, reconocimiento de patrones y toma de decisiones entre otras, sin la supervisión directa del ser humano. Se ha observado que algunas tareas de procesamiento, como la navegación utilizando sonares [20], seguimiento de caminos [21], sistema de navegación, generación de comportamientos, evasión de obstáculos, exploración, percepción y visión, entre otros, se pueden resolver recurriendo a métodos como las redes neuronales artificiales (RNA).

Actualmente (año 2005) se requieren resolver algunos de estos problemas utilizando estos métodos, por ejemplo, el problema de seguimiento de caminos o líneas para robots móviles. Una solución consiste en montar una cámara de video en el robot para obtener una imagen que será procesada por una red neuronal la cual controlará la dirección. Algunos centros de investigación, como la Universidad de Munich y la Universidad Carnegie-Mellon, han adoptado esta técnica para resolver el problema. Cuentan con vehículos especialmente adaptados con cámaras y equipo necesario para que éstos puedan seguir un camino por sí solos [21]. El esquema propuesto se muestra en la figura 1.

Este diseño fue propuesto por Pomerleau en la Universidad de Carnegie-Mellon y lo llamó ALVINN (Autonomous Land Vehicle in Neural Nets) [21]. En el esquema que propuso se tiene una retina o imagen de 30x32 pixeles como la capa de entrada, la cual se conectó a 4 neuronas ocultas y éstas a su vez a 30 neuronas discretas en la capa de salida con las que se controlará la dirección del vehículo. Una red de tipo Retropropagación puede ser entrenada para realizar este trabajo [21].

Tareas como esta se pueden resolver utilizando RNA que, desde su aparición hasta la fecha, se han introducido cada vez más como métodos alternativos. El uso de RNA tiene ventajas que otros métodos no habían presentado antes, como la capacidad para aprender a partir de ejemplos, su capacidad de generalización, etc. [4]. Sin embargo, estas herramientas requieren del procesamiento de grandes cantidades

de datos en forma simultánea con el fin de poder implementar soluciones en tiempo real. Esto ha dado lugar al desarrollo de sistemas dedicados especificamente al procesamiento de RNA [4].

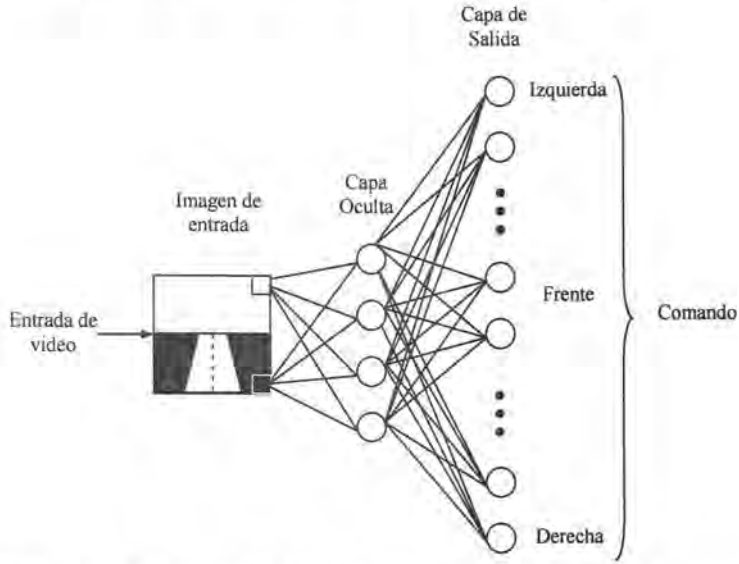


Fig. 1. Esquema del seguidor de camino propuesto con una red neuronal [21].

En el laboratorio se cuenta con tarjetas basadas en el microcontrolador MC68HC11 de motorola o PIC de microchip. Estas tarjetas son programadas en lenguajes como C o Basic. Se ha observado que el uso de estas tarjetas es ineficiente debido a que no nada más se encargan del procesamiento de la red neuronal, sino que controlan completamente al robot móvil. Es por eso que se tiene la necesidad de un dispositivo exclusivo para el procesamiento de datos, en este caso, para el procesamiento de redes neuronales artificiales.

Por otro lado, se han desarrollado estructuras cada vez más completas y complejas de los denominados dispositivos lógicos programables (PLD). Los cuales tienen la ventaja de poder implementar circuitos lógicos secuenciales y combinacionales en un solo circuito integrado. En la actualidad, se cuentan con familias de dispositivos tan grandes que llegan a tener hasta millones de compuertas lógicas programables y permiten el desarrollo de arquitecturas completas utilizando un solo circuito [11]. Una ventaja adicional que se tiene diseñando circuitos con dispositivos lógicos programables es que se reduce considerablemente el costo en comparación con el diseño ASIC (*circuitos integrados de aplicación especifica*) [7].

Antecedentes.

Existe actualmente una tendencia a establecer un nuevo campo de las ciencias de la computación que integra los diferentes métodos de resolución de problemas que no pueden ser descritos fácilmente mediante un enfoque algorítmico tradicional. Estos métodos tienen su origen en la emulación del comportamiento de los sistemas biológicos [4]. Inspirados en estos sistemas, surgieron las redes neuronales artificiales, máquinas de computación masivamente paralela, las cuales ofrecen un nuevo paradigma, donde *el aprendizaje a través de ejemplos* o *aprendizaje a través de la interacción*, reemplaza a la programación [1].

La mayor parte de las aplicaciones de redes neuronales son ejecutadas como simulaciones de software convencional en máquinas secuenciales con hardware no dedicado [1]. Estas máquinas se implementan sobre computadoras basadas en la filosofía de funcionamiento expuesta inicialmente por Von Neumann y se apoyan en una descripción secuencial del proceso de tratamiento de la información [4].

Existen principalmente 3 tecnologías utilizadas en los sistemas neuronales: Simuladores de software, chips de silicio (neurocomputadores) y procesadores ópticos [19]. En la práctica se suele trabajar con monoprocesadores, en este caso no existe paralelismo real, el procesamiento de la red se realiza secuencialmente realizando los cálculos de capa en capa. La propia estructura y modo de operar de las RNA las hace fáciles de implementar sobre multiprocesadores. Conociendo los fundamentos de la teoría, es fácil diseñar programas específicos para simular redes neuronales y existen paquetes matemáticos comerciales como *matlab* o *mathematica* los cuales son muy flexibles para desarrollar o modificar algoritmos.

Existen también sistemas de programación de procesos paralelos específicamente desarrollados para simular redes neuronales como el P3. Que es un sistema modular que funciona sólo sobre una máquina Symbolics 3600 en LISP. Dentro de los simuladores de software también se encuentran los aceleradores de hardware que son tarjetas que se conectan como periféricos y dan soporte en hardware para simular unidades de proceso. Suelen incluir software específico para su programación. Consiguen gran aceleración a pesar de compartir y multiplexar recursos, no ofrecen proceso en paralelo real [19].

Algunos de estos aceleradores de hardware son el Mark III y Mark IV, formados por conjuntos de procesadores Motorola 68020 con coprocesadores de punto flotante 68881, los cuales procesan 45,000 y 5,000,000 de conexiones por segundo respectivamente. El Anza consta de un microprocesador MC68020 junto con un coprocesador 68881 y 4MB de memoria, procesa 25,000 sinapsis por segundo mientras que el

Anza Plus procesa 6,000,000 de sinapsis por segundo. Otra placa procesadora es la Delta-Sigma formada por un procesador Delta 80386 y un coprocesador 80387 (Sigma), procesa 2,000,000 de sinapsis por segundo [4].

Por la naturaleza paralela de la computación neuronal, se han desarrollado máquinas específicas para su implementación, Lo ideal es que se cuente con una unidad de proceso elemental por cada elemento de la red y de conexiones físicas independientes donde se pudieran implementar las sinapsis. Con esto se conseguirían velocidades de proceso extremadamente altas [19], esto es, que un procesador con 1000 elementos independientes, sería 1000 veces más rápido que un acelerador de hardware que utilice una unidad similar de proceso.

En la actualidad se han hecho muchos desarrollos de procesadores neuronales utilizando hardware reconfigurable el cual provee las mejores características de circuitos de hardware dedicado e implementaciones de software: diseños de alta densidad y alto rendimiento, flexibilidad y realizaciones de prototipos rápidas [1]. Los diseños de los procesadores son diversos, dependen del modelo de neurona y topología de la red. La mayoría implementan un solo tipo de topología de red, como hopfield, perceptrón o retropropagación. Normalmente la topología es fija pero también se encuentran trabajos donde el número de neuronas se puede aumentar incrementando el número de dispositivos programables [18].

Finalmente los procesadores ópticos se encuentran en desarrollo pero la tecnología ofrece características importantes para este tipo de computación: el uso de luz permite muchos canales muy juntos unos de otros sin producir interferencias como sucede con los cables eléctricos. Permite conexiones más pequeñas con una densidad 4 veces mayor y un haz de luz puede atravesar a otro sin producir interferencia [19].

Planteamiento y limitación del problema.

El problema que se tiene al implementar soluciones de RNA en computadoras (procesadores o microcontroladores) convencionales es que se realiza mediante software y se deben tomar en cuenta los retardos que se tienen al realizar los cálculos debido a que se necesita de un uso extensivo de operaciones y la naturaleza de estos procesadores es secuencial, en contraparte se tiene la ventaja de que se pueden programar las redes de tal manera que sus parámetros sean configurables como son el número de entradas, salidas, modelo o modelos de las neuronas, topología de la red y representación numérica principalmente. Por otra parte, al diseñar procesadores neuronales utilizando dispositivos programables, se tiene que la mayor parte de esas soluciones se aplican a problemas específicos [13], [17]. La desventaja, es que se tienen arquitecturas limitadas, como por ejemplo,

algunas de ellas incorporan topologías de redes fijas, en otras se tienen redes en las que se pueden variar el número de neuronas y sinapsis pero obedecen a un tipo de red específico. La mayoría cuenta con una función de activación determinada, etc. (ver ref. [13] a [18]). En otros casos, se necesitan varios dispositivos programables para incrementar el tamaño de la red.

En las aplicaciones donde se diseñan procesadores con topologías y modelo de neurona fijos, se alcanzan velocidades de procesamiento que no son posibles utilizando computadoras secuenciales o inclusive con tarjetas aceleradoras de hardware, esto se debe a que estas arquitecturas están optimizadas para estas operaciones específicas y realizan los cálculos de forma paralela, en contraparte se tiene que los parámetros son fijos (entradas, salidas, número de neuronas y topología) y en ocasiones una vez configurada la red asignando pesos a las sinapsis, éstos no pueden ser modificados posteriormente [19]. En este sentido, se propone el diseño de una arquitectura que se encuentre dentro de estas dos metodologías de implementación de RNA y que cumpla con los siguientes:

Objetivos.

En este trabajo se propone el diseño de una arquitectura para el procesamiento de redes neuronales artificiales y digitales. El objetivo es crear un dispositivo flexible donde se puedan configurar sus parámetros como el número de entradas, número de salidas, función de activación para el modelo de neurona, número de neuronas, número de entradas por neurona e interconexión entre neuronas. El modelo de neurona es de tipo Adaline, en el cual se aplica una función de activación a la suma ponderada de las entradas, la respuesta es continua y no se consideran retardos axónicos (sección 3.1).

Así el dispositivo podrá ser utilizado en varios modelos de redes neuronales (perceptrón, adaline, retropropagación, hopfield, etc., o una especificada por el usuario), y permitirá el uso de una función de activación definida por el usuario. El tamaño máximo de la red dependerá de la memoria disponible y no del número de dispositivos programables con los que se cuente, de esta forma, para la tarjeta utilizada se tendrán alrededor de 1000 neuronas.

Otras metas que se buscan son la velocidad de procesamiento, en la cual se ofrezca un rendimiento que pueda al menos competir con computadoras o dispositivos actuales (una computadora a 1.8GHz puede procesar hasta 46 millones de conexiones por segundo, ver cap. 4), esto implica que debe ser una arquitectura con un alto grado de paralelismo. Se pueden considerar dos aspectos importantes: facilidad de uso y portabilidad, para que pueda ser

utilizado por sistemas autónomos, ya que la idea de realizar este trabajo surgió a partir de implementar algoritmos de control para robots móviles utilizando RNA. Al realizar el diseño de la arquitectura, se pudo observar que podía ser utilizado en diversas aplicaciones debido a sus características principales, de ahí que su uso en robots móviles se estableció como aplicaciones y no como un objetivo.

Desarrollo.

El trabajo se divide en cuatro capítulos donde se presenta el desarrollo completo de la arquitectura. Cada capítulo constituye una parte fundamental del proyecto y el contenido es el siguiente:

El capítulo uno es una introducción a las redes neuronales, se presenta con el fin de que un usuario que no tenga conocimientos acerca del tema, pueda ver los mecanismos que se encuentran detrás de toda red neuronal. No se exponen todos los tipos de RNA's, ya que este no es el objetivo. Lo que se hace es analizar las primeras configuraciones de redes hasta llegar al estudio de una de las más importantes, la red de retropropagación. Con eso se tienen las bases para comprender todas las operaciones que tiene qué realizar una arquitectura para un procesador neuronal.

El capítulo dos presenta la forma en la que trabajan los dispositivos lógicos programables, en especial los FPGA. Se muestra la configuración interna desde los dispositivos más pequeños, con el fin de que se pueda entender la estructura y funcionamiento de los circuitos con arreglos internos. Así se pueden analizar los tipos de aplicaciones que pueden ser programadas y el alcance de los mismos. La segunda parte del capítulo, presenta una introducción al lenguaje VHDL con el fin de mostrar la forma en la que son configurados los distintos tipos de PLD's. Se verá que el lenguaje es estándar y permite la programación de los dispositivos a alto nivel.

Existe mucha información acerca de RNA's y PLD's, se pueden encontrar libros enteros dedicados a su estudio. Aquí solo se utilizan como herramientas, ya que son la base para el desarrollo del proyecto y por lo tanto solo se presentan algunas características que fueron tomadas en cuenta para el diseño de la arquitectura.

En el capítulo tres, se explica de forma detallada la arquitectura, comenzando por analizar las operaciones requeridas y los objetivos que se tienen que alcanzar. Con esto se propone un tipo de arquitectura específica que, como se verá más adelante, es de tipo secuencial. Aquí se explica con diagramas la forma en la que funciona el procesador. Se exponen los bloques más importantes que realizan las operaciones de sumas de productos y funciones de activación. Uno de los principales objetivos a lograr es que se realice el procesamiento de una sinapsis en

un ciclo de reloj, no importando la topología de la red ni la función de activación de las neuronas. Se propone como objetivo debido a la naturaleza del procesador que no es paralela. En este capítulo también se explica la forma de utilizar el procesador a través de sus buses y señales de control. También se menciona las herramientas y el hardware utilizado para su implementación, dentro de este trabajo se pueden encontrar algunas especificaciones técnicas del dispositivo utilizado. También se encuentra un listado de las características principales que presenta el diseño final.

Finalmente en el capítulo cuatro, se presentan los resultados de las pruebas realizadas al procesador, configurando una red de tipo Hopfield para que reconozca imágenes. Debido a las limitaciones de memoria los patrones son pequeños pero se puede apreciar la funcionalidad de la aplicación y del procesador.

En el mismo capítulo se mencionan las limitaciones que tiene la arquitectura, se compara el desempeño con otros dispositivos y se menciona la forma en la que se pueden realizar mejoras o modificaciones al diseño, de tal forma que se pueda obtener un mejor rendimiento ya sea en velocidad o funcionalidad. Finalmente, se presentan las conclusiones obtenidas al final del desarrollo del proyecto. Se menciona el trabajo a futuro y se concluye que los objetivos fueron alcanzados de acuerdo a los resultados obtenidos.

CAPÍTULO 1

Redes Neuronales Artificiales

Las redes neuronales artificiales (RNA) encuentran cada vez un mayor número de aplicaciones en la actualidad, como el control adaptivo, reconocimiento de patrones, aproximación de funciones, filtrado, reconocimiento de voz, clasificación de señales, visión por computadora, compresión de datos, control de robots, etc.

El campo de estudio de las RNA se ha ampliado y ha sido cada vez más utilizado debido a sus características y potencialidades que la computación convencional no ha podido lograr de manera exitosa con las computadoras, procesadores de señales o inclusive, supercomputadoras.

Todas las redes neuronales artificiales cuentan con características comunes, ya que están basadas sobre el mismo principio que es la neurona biológica; no así otras características como su topología, algoritmo de aprendizaje o entrenamiento, función de activación, aplicación, etc. Es por esta razón que se comienza el estudio de las RNA por el funcionamiento de la neurona biológica, seguido por los modelos más simples que es el Perceptrón de una capa. Aquí es donde comienzan a observarse las ventajas y desventajas de este tipo de metodologías dentro de las aplicaciones ya señaladas. En este capítulo se mencionan solo dos tipos de redes, el Perceptrón, que es la red neuronal más antigua y después se introduce a la red de tipo Retropopagación.

Cabe aclarar que este capítulo introduce al lector a los conceptos básicos de las redes neuronales así como sus principios y aplicaciones con el fin de ilustrar los mecanismos básicos que se encuentran detrás de toda red neuronal. Para un estudio más profundo sobre las redes neuronales artificiales se pueden consultar las referencias recomendadas al final de la Tesis.

1.1 Conceptos Básicos

El ser humano a través del tiempo ha buscado la manera de desarrollar herramientas más poderosas que le permitan trabajar y estudiar de una manera más eficiente, rápida y sencilla. Sin duda alguna, una de las herramientas más poderosas que ha desarrollado en los últimos años es la computadora, y en general, los sistemas digitales.

Se ha visto que se pueden realizar tareas bastantes complejas con los dispositivos de procesamiento de datos actuales como los microprocesadores, microcontroladores, computadoras, supercomputadoras, etc. Aunque si bien existen diferencias significativas entre cada uno de estos dispositivos tienen algunas cosas en común, la más importante, es que esencialmente procesan los datos de manera secuencial ya que están basadas en las ideas de Von Neumman y Alan Turing [1]. Esta característica es bastante importante ya que limita el tipo de aplicaciones de este tipo de dispositivos, por ejemplo, se ha observado que estos procesadores son muy buenos para tareas como el desarrollo de fórmulas matemáticas, desarrollo de algoritmos para implementar la solución matemática, codificación del algoritmo para una máquina específica o ejecución de código [2]. En pocas palabras realizan tareas que involucran muchos cálculos muy simples y específicos, como operaciones matemáticas (aritméticas y lógicas) y saltos condicionales, es decir, resuelven problemas de tipo algorítmico.

El tipo de tareas para lo que se ha visto limitada la computación convencional es para tareas que al ser humano le parecen sencillas como el reconocimiento de patrones, voz, adaptación, percepción y aprendizaje [2]. Es por esta razón que se ha buscado otro tipo de cómputo no convencional inspirado en el éxito que se ha obtenido con los modelos biológicos. Es aquí donde surge el campo de estudio de las RNA.

La primera red neuronal conocida fue desarrollada por Warren McCulloc y Walter Pitts en 1943 [3], la cual consistía un una suma de señales de entrada multiplicadas por unos pesos escogidos aleatoriamente. La entrada era comparada con un patrón preestablecido o umbral. Si en la comparación la suma de productos era mayor o igual al umbral, la salida se ponía a "1", en caso contrario la señal era "0". En un principio se creyó que este modelo podía procesar cualquier función aritmética o lógica.

En 1957 el psicólogo Frank Rosenblatt inventó la red de tipo Perceptrón. Este modelo era capaz de generalizar, es decir, después de haber aprendido una serie de patrones era capaz de reconocer otros similares, aunque no se le hubieran presentado anteriormente. Sin embargo, no fue sino hasta 1959 que una red neuronal fue aplicada a un problema real, Bernard Widrow y Marcial Hoff desarrollaron el modelo Adaline y la implementaron como un filtro adaptivo para eliminar ecos en las líneas telefónicas [4].

La compleja operación de las redes neuronales es el resultado de abundantes lazos de realimentación junto con *no linealidades* de los elementos de proceso y cambios adaptivos de sus parámetros [4].

Otro aspecto importante de las redes neuronales biológicas es su tamaño: en el sistema nervioso central hay del orden de 10^{11} neuronas y el número de interconexiones es mayor, alrededor de 10^{15}. Esto, junto con la función de la neurona, presenta las siguientes ventajas con respecto a las computadoras modernas [5]:

- Paralelismo masivo.
- Representación y computación distribuida.
- Habilidad de aprendizaje.
- Habilidad de generalización.
- Adaptabilidad.
- Procesamiento de información contextual inherente.
- Tolerancia a fallos y
- Bajo consumo de energía.

Con esto se observa que la computación neuronal ofrece una alternativa a la solución de tareas que difícilmente se resuelven con la computación convencional. Además, con las RNA surgieron nuevas ideas y paradigmas donde el *aprendizaje por ejemplos* o el *aprendizaje por interacción* reemplaza a la programación [1].

1.2. La neurona biológica

La teoría y modelado de las redes neuronales está inspirada en el funcionamiento de los sistemas nerviosos, donde la neurona es el elemento fundamental [4]. Una neurona es una célula viva y contiene los elementos que forman parte de todas las células biológicas pero poseen otros elementos que las diferencian de éstas. Dentro de los elementos más importantes se encuentran las dendritas, el núcleo y el axón. Una característica especial de las neuronas es su capacidad para comunicarse, recibe entradas a través de las dendritas y genera una respuesta que es transportada por el axón. La figura 1.1 muestra la estructura básica de la neurona.

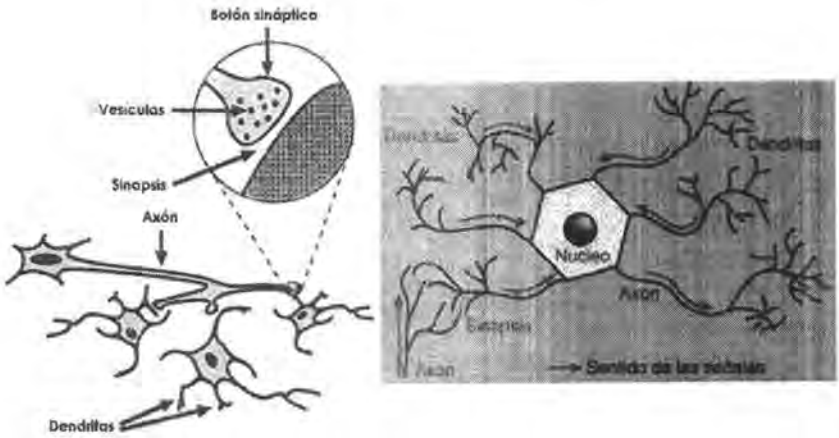


Fig. 1.1. Estructura básica de una neurona [6].

A las conexiones entre neuronas se les llama sinapsis. La mayoría de las neuronas no se tocan entre sí, las sinapsis utilizan señales de dos tipos: eléctricas y químicas [6]. La señal que se transporta a lo largo del axón es un impulso eléctrico, mientras que la señal que se transmite de éste a la dendrita de la siguiente neurona es de origen químico. Aunque también existen sinapsis puramente eléctricas pero éstas tienen menos posibilidades que las sinapsis químicas [6]. En las terminaciones de los axones se encuentran unas vesículas que contienen sustancias químicas a las que se les denomina neurotransmisores.

La generación de las señales eléctricas está íntimamente relacionada con la composición de la membrana celular. Existen muchos procesos relacionados con la generación de estas señales, la composición de la neurona difiera a la composición del medio externo, la diferencia más significativa se da entre la relación de la concentración de iones de sodio y de potasio. El medio externo es más rico en sodio que el interno y esto produce un potencial de aproximadamente -70 mV en el interior de la célula. Este es el *potencial de reposo*. Las señales de otras neuronas actúan acumulativamente bajando el potencial de reposo que cuando llega a cierto valor crítico comienza una entrada masiva de iones de sodio que invierte la polaridad de la membrana. Esta inversión de voltaje se conoce como *potencial de acción* y se propaga a lo largo del axón que a su vez provoca la emisión de los neurotransmisores en las terminales axónicas [4].

La importancia de estudio de la neurona biológica radica en su funcionamiento, ya que las redes neuronales son modelos que intentan reproducir el comportamiento del cerebro y por lo tanto, sus características principales en cuanto a comportamiento.

1.3 Modelo de la neurona

El modelo matemático de la neurona se puede dividir en tres fases, cada una correspondiente a uno de los elementos de la neurona biológica como son las dendritas, el cuerpo de la neurona o soma y el axón. Así se tienen tres operaciones: Dendrítica, Somática y Axónica.

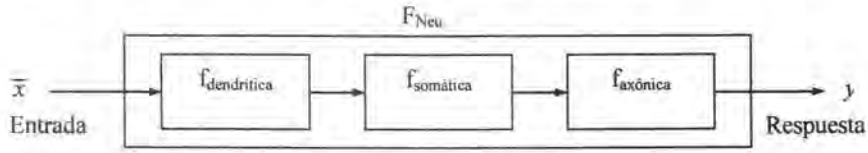


Fig. 1.2. Transformación Neuronal.

Una neurona recibe información de sus entradas y responde con una salida. Las entradas pueden provenir de otras neuronas, del medio o de ambos. Cada neurona de la red está caracterizada en cualquier instante por un valor numérico denominado *estado de activación*. Esta señal es enviada a otras neuronas a través de canales unidireccionales y en estos canales la señal se modifica de acuerdo con la sinapsis asociada a cada uno de ellos. Las señales que llegan a una determinada neurona se combinan entre ellas para generar la entrada total.

Así la función dendrítica se puede expresar como una suma de las señales de entrada, cada una multiplicada por un valor de peso:

$$f_{dendritica} = \sum_{i} (x_i w_i)$$

Donde: x_i i-ésima entrada a la neurona

w_i peso correspondiente a la entrada x_i.

Como el objetivo del modelo es intentar reproducir el comportamiento de la neurona, se elimina información redundante, así la función somática se puede expresar simplemente como la función identidad:

$$f_{somatica} = I(x)$$

Finalmente, se tienen varias funciones de salida o de transferencia que corresponden a la función axónica, las más utilizadas son la función escalón, rampa, sigmoidal, tangente hiperbólica y gaussiana. Cabe señalar una característica importante de la operación o transformación neuronal: *las neuronas biológicas son elementos no lineales*. Esta no linealidad se puede expresar en la función de salida.

Algunos autores representan a la función dendrítica como el producto de la entrada por el peso correspondiente de cada sinapsis y a la función somática como la suma de estos productos, otros autores simplemente señalan el comportamiento general de la neurona sin mencionar estos tres tipos de funciones. Lo más importante es la operación en conjunto, es decir, el modelo total de la neurona.

Las señales de salida o transferencia se muestran en la figura 1.3

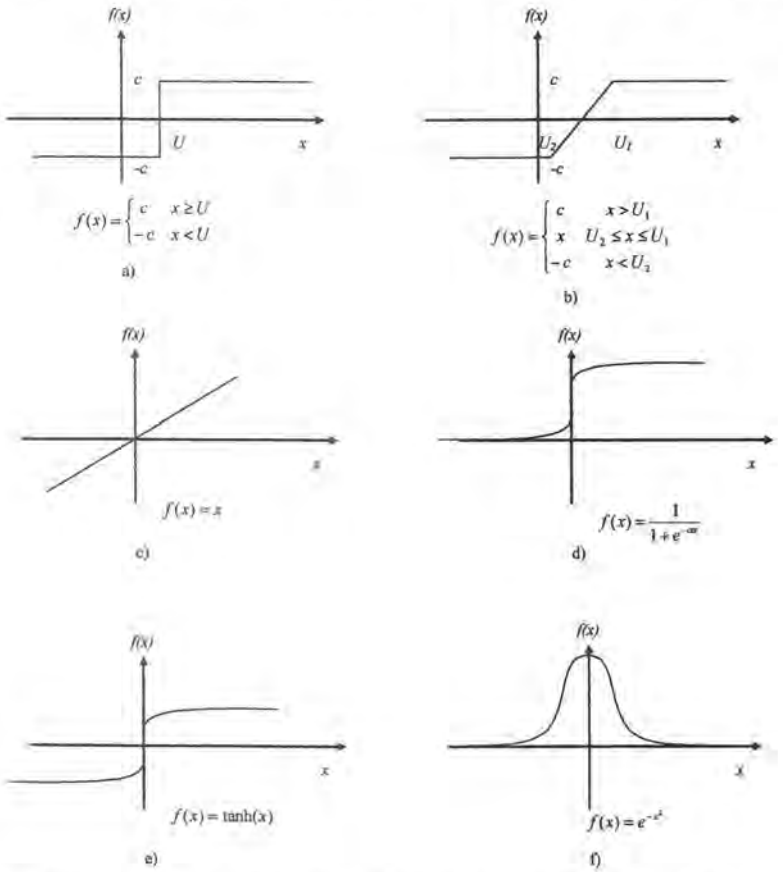


Fig. 1.3. Funciones de transferencia: a) Escalón, b) Mixta, c) Lineal, d) Sigmoidal, e) Tangente Hiperbólica, f) Gaussiana.

El modelo matemático de la neurona se expresa en la ecuación 1.1.

$$y_i^k = f\left(\sum_{j=1}^{N} x_j^i w_j^i\right) \qquad (1.1)$$

donde:

y	salida de la neurona i en la capa k
i	numero de neurona
k	numero de capa
N	numero de entradas a la neurona
x_j	entrada j a la neurona
w_j	peso asociado a la entrada j
f	función neuronal

Las conexiones que unen a las neuronas en la RNA tienen asociado un peso que es el que hace que la red adquiera conocimiento. Biológicamente, se suele aceptar que la información memorizada en el cerebro está más relacionada con los valores sinápticos de las conexiones entre las neuronas que con ellas mismas, es decir, el conocimiento se encuentra en las sinapsis. En el caso de las RNA, se considera que el conocimiento se encuentra en los pesos de las conexiones entre neuronas [4].

El modelo de la neurona se puede ilustrar en la figura 1.4.

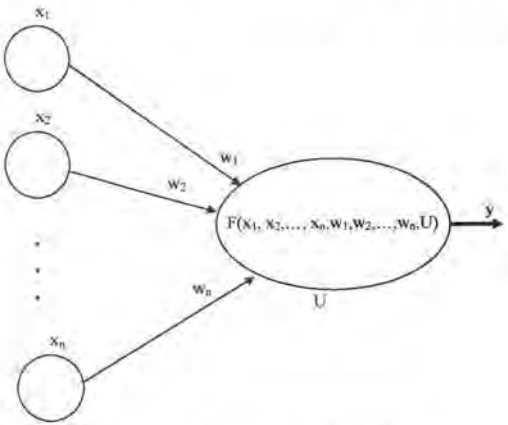


Fig. 1.4. Neurona Artificial.

La distribución de las neuronas dentro de la red se realiza formando niveles o capas de un número determinado de neuronas cada una. Se pueden distinguir tres tipos de capas:

- **De entrada**, es la capa que recibe la información del exterior.
- **Ocultas**, son capas internas a la red y no tienen contacto directo con el exterior. El número de estas capas es variable y las neuronas de estas capas se pueden conectar de distintas maneras, determinando la topología de la red.
- **De salida**, las neuronas de esta capa transfieren la información de la red hacia el exterior.

La figura 1.5 muestra la estructura de una red multinivel.

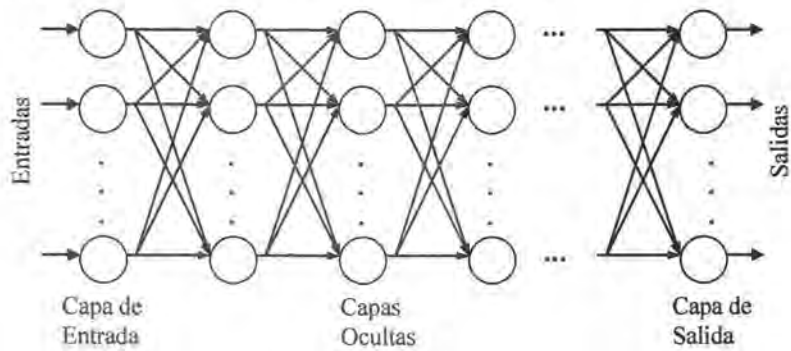


Fig. 1.5. Estructura de una red neuronal multinivel o multicapa.

Existen otros tipos de topologías, que se desarrollaron para redes que tienen distintas características o comportamientos, la figura 1.6 muestra algunas de ellas.

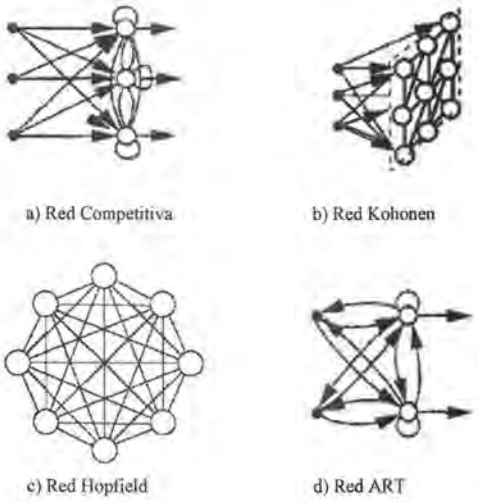


Fig. 1.6. Distintas topologías de RNA.

1.4 Topologías de Redes Neuronales

Existen muchos tipos de RNA, aquí sólo se mencionarán dos de ellos con el fin de ejemplificar su funcionamiento, estructura, y aprendizaje.

1.4.1 El Perceptrón

Como ya se mencionó, el Perceptrón es la red neuronal más antigua y la desarrolló Rosenblatt en 1958. Tiene la capacidad para aprender a reconocer patrones sencillos. La figura 1.7 muestra al Perceptrón.

En la figura se puede observar la estructura de la red, su modelo y función de transferencia así como su desempeño. La única neurona de salida del Perceptrón realiza la suma ponderada de las entradas y el resultado lo pasa a la función de transferencia de tipo escalón. Así responde con +1 ó -1 dependiendo a qué clase pertenece la entrada.

Para analizar el comportamiento del Perceptrón se puede representar un mapa de regiones de decisión creadas en el espacio multidimensional de entradas de la red. En estas regiones se visualiza qué patrones pertenecen a una clase y qué patrones a otra. El Perceptrón separa las regiones por un hiperplano cuya ecuación queda determinada por los pesos de las

conexiones y el umbral. Los valores de los pesos son los que varían acorde al algoritmo de entrenamiento de la red para variar estas regiones.

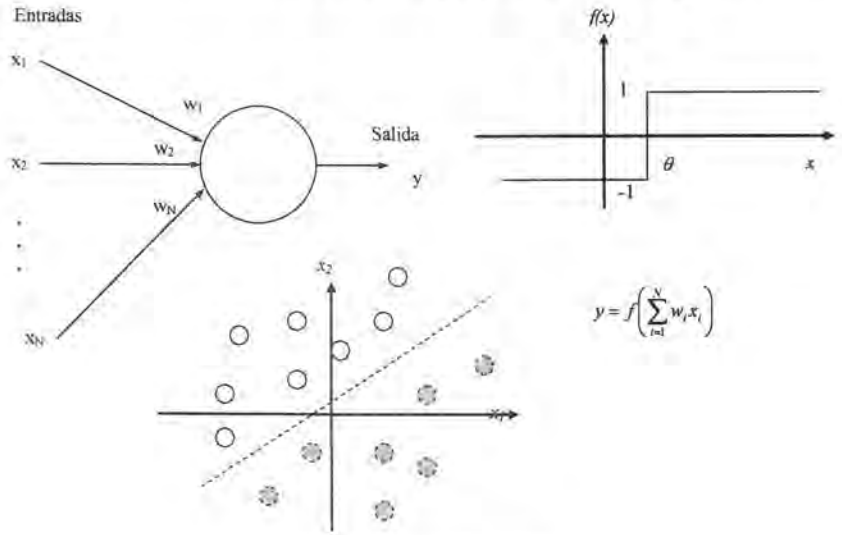


Fig. 1.7. El Perceptrón.

1.4.2 Red Retropropagación

La regla de aprendizaje del Perceptrón y el algoritmo LMS fueron diseñados para entrenar redes de una sola capa. Estas redes tienen la desventaja que sólo pueden resolver problemas linealmente separables, por lo que surgieron las redes multicapa para resolver este problema, pero acarrearon uno nuevo: su entrenamiento.

El primer algoritmo de entrenamiento para redes multicapa fue desarrollado por Paul Werbos en 1974 [3], pero se desarrolló en un contexto general para cualquier tipo de redes por lo que el algoritmo no fue aceptado por los desarrolladores de redes neuronales. En 1986, Rumelhart, Hinton y Williams, formalizaron un método para que una red neuronal aprendiera la asociación que existe entre los patrones de entrada a la misma y las clases correspondientes, utilizando más de un nivel de neuronas. Este método, conocido como *backpropagation* (retropropagación o propagación del error hacia atrás), está basado en la generalización de la regla delta y ha ampliado el rango de aplicaciones de la RNA.

La importancia de este método consiste en que, conforme se entrena la red, las neuronas de las capas ocultas o intermedias se organizan de tal

modo que aprenden la relación que existe entre un conjunto de patrones dados como ejemplo y sus salidas correspondientes, para así poder aplicar esa misma relación, después del entrenamiento, a nuevos valores de entrada incompletos o con ruido, dando la salida correspondiente. Esta característica tan importante es la capacidad de *generalización*, que es la facilidad de dar salidas satisfactorias a entradas que el sistema no ha visto nunca en su fase de entrenamiento.

La estructura de una red de retropropagación se muestra en la figura 1.8. En ella existe una capa de entrada de *n* neuronas, una capa de salida de *m* neuronas y al menos una capa oculta de *l* neuronas. El número de neuronas por capa oculta puede ser distinto, cada neurona de una capa recibe entradas de todas las neuronas de la capa anterior. No hay conexiones hacia atrás ni laterales entre neuronas de la misma capa. La función de transferencia de cada capa de neuronas puede ser distinta pero todas las neuronas de una misma capa deben tener la misma función de activación.

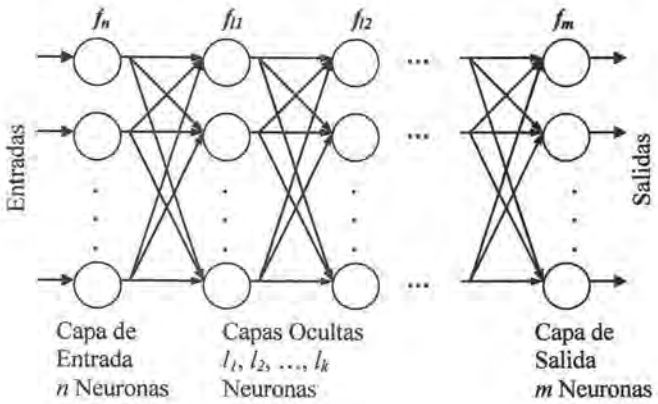


Fig. 1.8. Red Retropropagación.

Como ejemplo, supóngase que se tienen los siguientes pares de puntos y se desea entrenar una red de tipo retropropagación para que aprenda los patrones y genere una salida satisfactoria para una entrada determinada:

P = {0 0}, {1 1}, {2 2}, {3 3}, {4 4}, {5 3}, {6 2}, {7 1}, {8 2}, {9 3},
{10 4}

Se puede obtener un vector de entradas y uno de salidas:

```
X = [0 1 2 3 4 5 6 7 8 9 10];
Y = [0 1 2 3 4 3 2 1 2 3 4];
```

Se propone una red con 5 neuronas de entrada y una de salida como se muestra en la figura 1.9, no existe ningún criterio con el que se tomó esta estructura determinada, simplemente se desea ver la salida que puede generar una red específica.

La función de activación de las neuronas de la capa de entrada es tangente hiperbólica, mientras que la función de activación de la neurona de salida es lineal. Las figura 1.10 muestra el comportamiento de la red antes y después de su entrenamiento. La red fue entrenada en matlab y se le indicó que realizara el ciclo 50 veces.

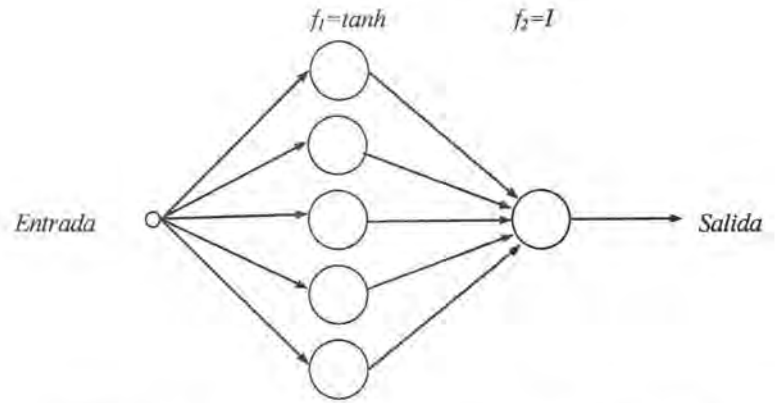
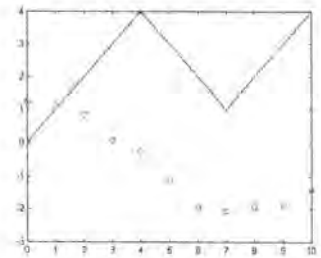
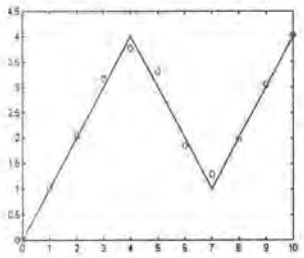


Fig. 1.9. Arquitectura de red de retropropagación propuesta.

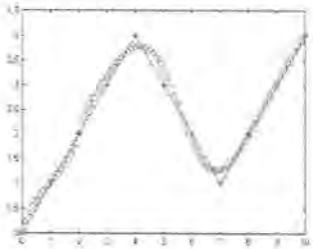
Existen muchos otros tipos de redes neuronales, las diferencias se encuentran en sus topologías, algoritmos de aprendizaje, etc. Pero como se ha visto, todas las redes neuronales realizan las mismas funciones porque están basadas en el modelo de la neurona, lo único que varía en determinado caso es la función de activación o transferencia. Existe un extenso acervo bibliográfico acerca de las redes neuronales artificiales, no es tema de esta Tesis el exponer y formalizar matemáticamente cada una de ellas sino ilustrar sus principios y funcionamiento.

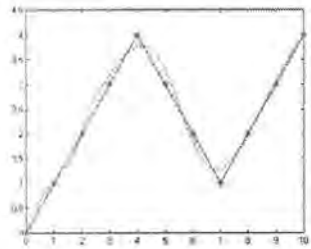


a) Respuesta de la red antes de la fase de entrenamiento.



b) Respuesta de la red después de la fase de entrenamiento.





c) Respuesta de la red a diferentes entradas.

Fig. 1.10. Respuesta de la red de retropropagación antes y después del entrenamiento.

La siguiente tabla tomada de [4], muestra algunos modelos de redes neuronales así como sus características más importantes:

MODELO DE RED	TOPOLOGIA	APRENDIZAJE			ASOCIACION AUTO/HETERO	INFORMAC. DE ENTRADA Y SALIDA	AUTOR(ES)
		ON/ OFF LINE	SUPERVISA/NO SUPERVISA	REGLA			
ADALINE/ MADALINE	2 CAP AS Feedforward	OFF	SUPERVIS.	CORRECCION ERROR ("LEAST MEAN SQUARE")	HETEROASOC	E:ANALOG S:BINARI	WIDROW HOFF 1960
ADAPTIVE BIDIRECTIONAL ASSOCIATIVE MEMORY. ABAM	2 CAPAS Feedforward /FEEDBACK	ON	NO SUPERV	HEBBIANO DIFERENCIAL	HETEROASOC	ANALOG.	KOSKO 1987
ADAPTIVE HEURISTIC CRITIC. AHC	3 CAPAS Feedforward	ON	SUPERVIS.	REFUERZO POR AJUSTE TEMPORAL	HETEROASOC	E:ANALOG S:BINARI	BARTO 1983
ADAPTIVE RESONANCE THEORY 1.ART1	2 CAPAS Feedforward /FEEDBACK CONEX. LAT. AUTO-RECU.	ON	NO SUPERV	COMPETITIVO (RESONANCIA ADAPTATIVA)	HETEROASOC	BINARIAS	CARPENTER GROSSBERG
ADAPTIVE RESONANCE THEORY 2.ART2	2 CAPAS Feedforward /FEEDBACK CONEX. LAT. AUTO-RECU.	ON	NO SUPERV	COMPETITIVO (RESONANCIA ADAPTATIVA)	HETEROASOC	ANALOG.	CARPENTER GROSSBERG

Tabla 1.1. Modelos de RNA y sus características principales. [4]

MODELO DE RED	TOPOLOGÍA	APRENDIZAJE			ASOCIACIÓN AUTO/HETERO	INFORMAC. DE ENTRADA Y SALIDA	AUTOR(ES)
		ON/ OFF LINE	SUPERVISA/NO SUPERVISA	REGLA			
ADITIVE GROSSBERG. AG	1 CAPA CONEXIONES LATERALES AUTO-RECU.	ON	NO SUPERV	HEBBIANO 0 COMPETITIVO	AUTOASOC.	ANALOG.	GROSSBERG 1968
ASSOCIATIVE REWARD PENALTY. ARP	2 CAPAS Feedforward	ON	SUPERVIS.	REFUERZO ESTOCASTICO	HETEROASOC	E:ANALOG S:BINARI	BARTO 1985
BACK-PROPAGATION	N CAPAS Feedforward	OFF	SUPERVIS.	CORRECCIÓN ERROR (REGLA DELTA GENERALIZADA)	HETEROASOC	ANALOG.	RUMELHART etc. 1986
RECURREN! BACKPROPAGAT.	N CAPAS FF/F.BACK	OFF	SUPERVIS.	CORRECCIÓN ERROR (R.DELTA GENER.)	HETEROASOC	ANALOG.	PINEDA etc. 1987
BIDIRECTIONAL ASSOCIATIVE MEMORY. BAM	2 CAPAS Feedforward /FEEDBACK	ON	NO SUPERV	HEBBIANO	HETEROASOC	BINARIAS	KOSKO 1988
BOLTZMANN MACHINE. BM	1 CAPA CONEX.LAT. 3 CAPAS Feedforward	OFF	SUPERVIS.	ESTOCASTICO ("SIMULATED ANNEALING") + HEBBIANO ó +CORRECCION ERROR	HETEROASOC	BINARIAS	HINTON ACKLEY SEJNOWSKI 1984

Tabla 1.1. Modelos de RNA y sus características principales. [4]

MODELO DE RED	TOPOLOGÍA	APRENDIZAJE			ASOCIACIÓN AUTO/HETERO	INFORMAC. DE ENTRADA Y SALIDA	AUTOR(ES)
		ON/ OFF LINE	SUPERVISA/NO SUPERVISA	REGLA			
BRAIN-STATE IN-A-BOX	1 CAPA CONEX.LAT. AUTO-RECU.	OFF	SUPERVIS.	CORRECCIÓN ERROR (REGLA DELTA)	AUTOASOC.	ANALOG.	ANDERSON 1977
CAUCHY MACHINE. CM	1 CAPA CONEX.LAT. 3 CAPAS Feedforward	OFF	NO SUPERV	ESTOCÁSTICO ("FAST SIMULATED ANNEALING")	HETEROASOC	E:ANALOG S:BINARI	SZU 1986
COGNITRON/ NEOCOGNITRON	JERARQUÍA DE NIVELES CON CAPAS BIDIMENS. FF/F.BACK	OFF	NO SUPERV	COMPETITIVO	HETEROASOC	BINARIAS	FUKUSHIMA 1975/1980
COMPETITIVE ABAM. CABAM	2 CAPAS FF/F.BACK CONEX.LAT. AUTO-RECU.	ON/ OFF	NO SUPERV	HEBBIANO + COMPETITIVO	HETEROASOC	ANALOG.	KOSKO 1987
COUNTER-PROPAGATION	3 CAPAS Feedforward CONEX.LAT. y AUTO-RECU.	OFF	SUPERVISADO	CORRECCIÓN ERROR + COMPETITIVO	HETEROASOC	ANALOG.	HECHT-NIEL-SEN87
DRIVE-REINFORCEMENT	2 CAPAS Feedforward	OFF	NO SUFERV	HEBBIANO (DRIVE REINFORCEMENT)	HETEROASOC	ANALOG.	KLOPF J986

Tabla 1.1. Modelos de RNA y sus características principales. [4]

MODELO DE RED	TOPOLOGÍA	APRENDIZAJE			ASOCIACIÓN AUTO/HETERO	INFORMAC. DE ENTRADA Y SALIDA	AUTOR(ES)
		ON/ OFF LINE	SUPERVISA/NO SUPERVISA	REGLA			
FUZZY ASSOCIATIVE MEMORY. FAM	2 CAPAS FF/ FEEDBACK	OFF	NO SUPERV	HEBBIANO BORROSO	HETEROASOC	ANALOG.	KOSKO 1987
CONTINOUS HOPFIELD	1 CAPA CONEX. LATERALES	OFF	NO SUPERV	HEBBIANO	AUTOASOC.	ANALOG.	HOPFIELD 1984
DISCRETE HOPFIELD	1 CAPA CONEX. LAT.	OFF	NO SUPERV	HEBBIANO	AUTOASOC.	BINARIAS	HOPFIELD 1982
LEARN. MATRIX. LM	1 CAPA CROSSBAR	OFF	NO SUPERV	HEBBIANO	HETEROASOC	BINARIAS	STEINBUCH 1961
LEARNING VECTOR QUANTIZER. LVQ	2 CAPAS FF CONEX. LAT. IMPLIC. AUTORREC.	OFF	NO SUPERV	COMPETITIVO	HETEROASOC.	ANALOG.	KOHONEN 1981
LINEAR ASSOCIATIVE MEMORY. LAM	2 CAPAS Feedforward	OFF	NO SUPERV	HEBBIANO	HETEROASOC	ANALOG.	ANDERSON 1968 KOHONEN 77
LINEAR REWARD PENALTY. LRP	2 CAPAS Feedforward	ON	SUPERVIS.	REFUERZO ESTOCÁSTICO	HETEROASOC	E:ANALOG S:BINARI	BARTO 1985

Tabla 1.1. Modelos de RNA y sus características principales. [4]

MODELO DE RED	TOPOLOGÍA	APRENDIZAJE			ASOCIACIÓN AUTO/HETERO	INFORMAC. DE ENTRADA Y SALIDA	AUTOR(ES)
		ON/ OFF LINE	SUPERVISA/NO SUPERVISA	REGLA			
OPTIMAL LINEAR ASSOCIATIVE MEMORY. OLAM	2CAPAS/FF. 1 CAPA CONEX.LAT. AUTO-RECU.	OFF	NO SUPERV	HEBBIANO ("OPTIMAL LEAST MEAN SQUARE CORRELATION")	HETEROASOC AUTOASOC.	ANALOG.	WEE 1968 KOHONEN73
PERCEPTRON	2 CAPAS Feedforward	OFF	SUPERVIS.	CORRECCIÓN ERROR	HETEROASOC	E:ANALOG S:BINARIA	ROSENBLAT 1958
SHUNTING GROSSBERG. SG	1 CAPA CONEX.LAT. AUTO-RECU.	ON	NO SUPERV	HEBBIANO O COMPETITIVO	AUTOASOC.	ANALOG.	GROSSBERG 1973
SPARSE DISTRIBUTED MEMORY. SDM	3 CAPAS Feedforward	OFF	NO SUPERV	HEBBIANO + RANDOM VECTOR (LVQ) PREPROCESSING	HETEROASOC	BINARIAS	KANERVA 1984
TEMPORAL ASSOCIAT. MEMORY. TAM	2 CAPAS FF/FEEDBACK	OFF	NO SUPERV	HEBBIANO	HETEROASOC	BINARIAS	AMARI 1972
TOPOLOGY PRESERVING MAP. TPM	2 CAPAS FF CONEX. LAT. IMPLIC. AUTORREC.	OFF	NO SUPERV	COMPETITIVO	HETEROASOC	ANALOG.	KOHONEN 1982

Tabla 1.1. Modelos de RNA y sus características principales. [4]

CAPÍTULO 2

Dispositivos Lógicos Programables y VHDL

El rápido avance tecnológico de la actualidad se debe en gran parte al rápido desarrollo de circuitos cada vez más complejos, pequeños, confiables y eficientes. El diseño electrónico digital ha cambiado con el paso del tiempo, el uso de computadoras con programas de diseño avanzados facilitaron el diseño y la simulación de circuitos que cada vez se realizaban en menor tiempo. No obstante, todavía se tenian algunos problemas o inconvenientes con la implementación. En estos días, con la aparición de los dispositivos lógicos programables, las fases de diseño e implementación de circuitos digitales ha cambiado de forma radical de tal manera que se pueden crear circuitos a la medida en una sola tarjeta y en un tiempo considerablemente rápido.

El tipo de aplicaciones de los PLD's (dispositivos lógicos programables) depende de la imaginación del desarrollador, pueden ir desde un simple circuito de conmutación hasta el desarrollo de una arquitectura avanzada de un procesador, DSP, etc. Existen muchos tipos de PLD's, la elección de uno u otro depende de la aplicación, tanto de la función como del tamaño final del circuito así como del costo de cada dispositivo.

Actualmente existen muchos trabajos relacionados con los PLD's, en especial con los denominados FPGA's por su facilidad de uso y sus potencialidades. Las aplicaciones se encuentran en todo tipo de rubros y son tan difundidos que existe mucha información sobre ellos. También existen muchas empresas dedicadas a la fabricación de estos dispositivos, dentro de las principales se encuentra Altera, Xilinx, Cypress, Atmel, Mantis, Lattice, National Semiconductor, etc.

Este apartado presenta una introducción a los dispositivos lógicos programables, en especial a los FPGA's mencionando sus características y arquitectura. También se presenta una introducción al lenguaje de descripción de hardware VHDL con el fin de mostrar cómo es que se puede realizar el diseño lógico en este tipo de dispositivos y lo fácil que es llegar hasta la fase de implementación y pruebas.

2.1 Tipos de dispositivos lógicos programables

En la actualidad existe una gran variedad de dispositivos lógicos programables y se utilizan para reemplazar a los circuitos SSI, MSI y VLSI ya que ahorran espacio y reducen de manera significativa el tiempo y costo de los diseños. Estos dispositivos se clasifican por su arquitectura:

- PLA: Programmable Logic Array.
- PAL: Programmable Array Logic.
- GAL: Generic Array Logic.
- PROM: Programmable Read-Only Memory.
- CPLD: Complex PLD.
- EPLD: Erasable PLD.
- HCPLD: High Complexity PLD.
- LCA: Logic Cell Array.
- FPGA: Field Programmable Gate Array

2.1.1 Dispositivos PAL

Los dispositivos PLA, PAL, GAL, y PROM están formados por arreglos o matrices de compuertas programables, los CPLD, HCPLD y LCA se encuentran estructurados mediante bloques lógicos configurables y los FPGA contienen celdas lógicas de alta densidad.

La arquitectura básica de un PLD está formada por un arreglo de compuertas *AND* y *OR* conectadas a las entradas y salidas del dispositivo. Estos arreglos pueden ser fijos o programables. La figura 2.1 muestra este tipo de arreglos.

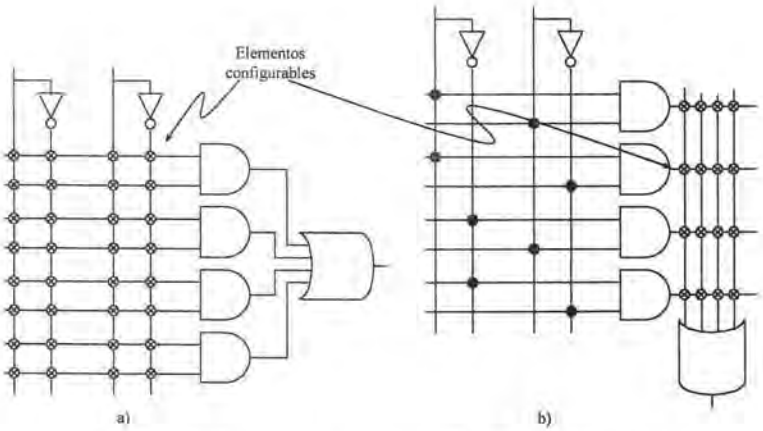


Fig. 2.1. Arreglos programables. a) AND y b) OR

En la figura se observa que se pueden tener matrices AND u OR configurables o incluso, ambas. Este tipo de arquitectura permite cambiar las conexiones de entrada para las distintas compuertas, al poder cambiar estas conexiones, se puede lograr que el dispositivo implemente a la salida cualquier función dependiendo del arreglo configurable de conexiones.

Los dispositivos PROM no se utilizan como un dispositivo lógico, sino como memoria debido a las limitaciones que presenta con las compuertas AND fijas.

EL PLA es un PLD formado por arreglos AND y OR programables.

EL PAL es un arreglo formado por AND's programables y OR's fijas. Se desarrolló para superar algunas limitaciones del PLA, como retardos provocados por la implementación de lógica reconfigurable adicional por la utilización de dos arreglos programables [7].

El arreglo lógico genérico GAL, es similar al PAL ya que contiene un arreglo AND programable y un arreglo OR fijo con salida programable. La diferencia con el PAL es que una GAL es reprogramable y contienen configuraciones de salida programables. La figura 2.2 muestra la arquitectura interna de una GAL.

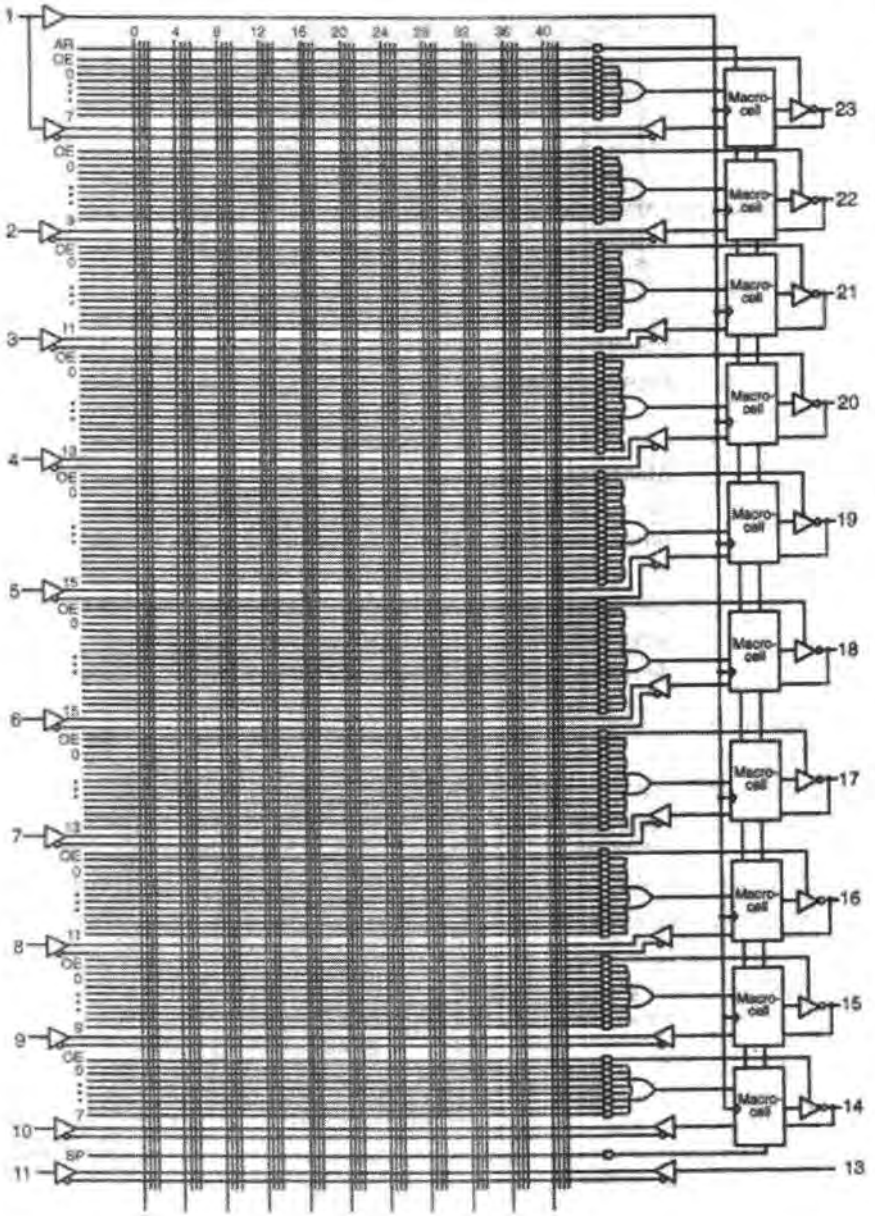


Fig. 2.2. GAL22V10 [7].

Una caracteristica importante de los dispositivos GAL es que a la salida cuenta con bloques configurables llamados macroceldas que

están formados por circuitos lógicos y flip-flops, así el dispositivo se puede programar con lógica combinacional o secuencial. La figura 2.3 muestra una macrocelda de la GAL22V10.

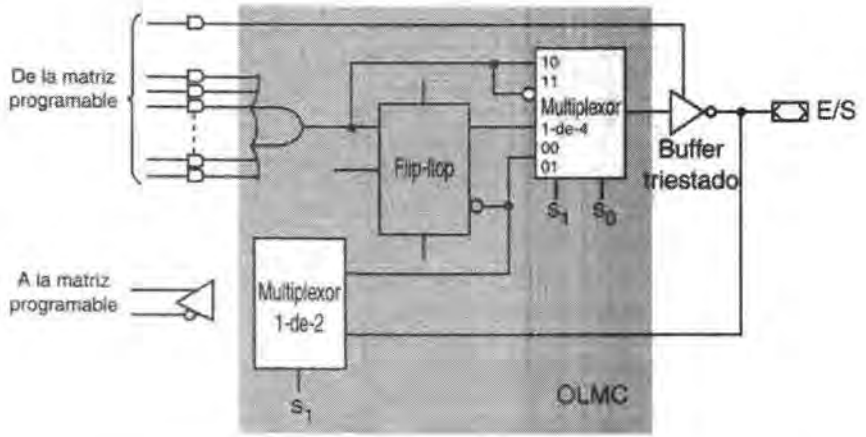


Fig. 2.3. Macrocelda de una GAL22V10 [7].

Como puede observarse en la figura, la macrocelda se puede configurar para elegir a la salida la señal proveniente del arreglo programable o del flip-flop, también realimenta la salida a la matriz reprogramable. Esto permite no solo implementar circuitos combinacionales, sino implementar también circuitos secuenciales ya que se tienen elementos de memoria y señales de realimentación. También las macroceldas generalmente contienen buffers tres estados, esto permite configurar el pin como entrada, salida o ambos.

2.1.2 Dispositivos CPLD

La arquitectura de los dispositivos CPLD depende de la familia a la que pertenezcan pero básicamente consisten en múltiples arreglos PLD's interconectados entre sí. Estos dispositivos también se conocen como EPLD (Enhanced PLD), Super PAL, Mega PAL, etc. [7]. Tienen un alto nivel de integración y se miden en bloques lógicos programables, por ejemplo, 20 PLD's.

La estructura de estos dispositivos se muestra en la figura 2.4. Se puede apreciar que está constituido principalmente por bloques lógicos programables y bloques de E/S conectados a través de una interconexión programable (PIA). Los bloques lógicos también son conocidos como celdas generadoras y están formados por un arreglo de productos de términos (AND) y un arreglo de distribución de términos que crea las sumas de productos (OR). La figura 2.5 muestra la estructura de una macrocelda de la familia MAX7000.

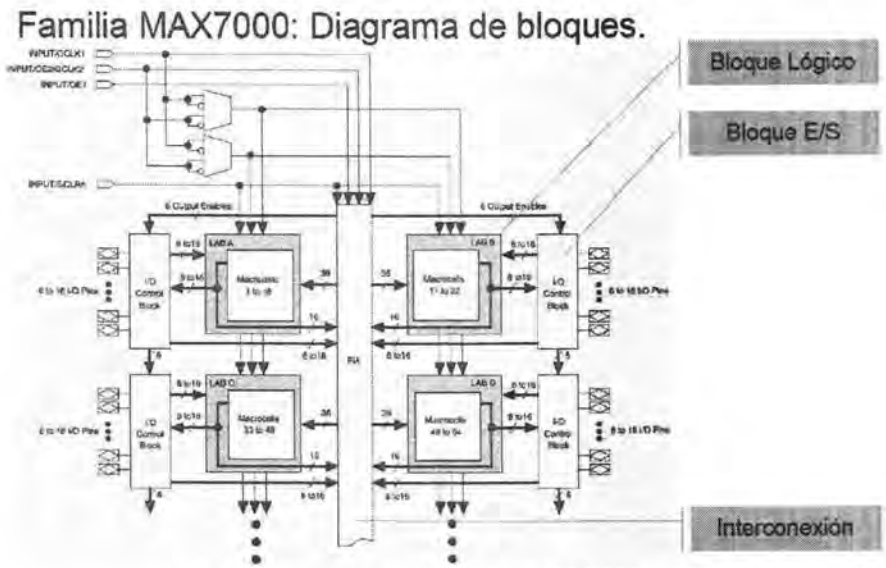


Fig. 2.4. Estructura de un CPLD MAX7000 [10].

Algunos fabricantes separan la macrocelda como bloque externo, otros la consideran dentro del bloque lógico programable. Es importante el número de macroceldas que contiene cada bloque lógico y el tamaño de éste, ya que esto determina la complejidad de las funciones que se pueden implementar en el dispositivo. Por ejemplo, en los dispositivos de esta familia (MAX7000) los bloques de arreglos lógicos (LAB) consisten de una serie de macroceldas y éstas a su vez contienen arreglos de compuertas programables y elementos de memoria como registros. Las salidas de las macroceldas van directamente a la interconexión programable para que puedan se utilizadas por otras macroceldas o incluso por la misma.

Las macroceldas también contienen ampliadores que permiten implementar funciones que no caben en una sola macrocelda y permiten compartir recursos entre ellas. La figura 2.6 muestra cómo están implementados estos ampliadores.

Así, en un CPLD se pueden implementar circuitos más completos que en un dispositivo simple como PAL o GAL. Mientras que en un dispositivo de estos tan solo se puede programar un bloque, en un CPLD se pueden programar varios de ellos e interconectar entre sí.

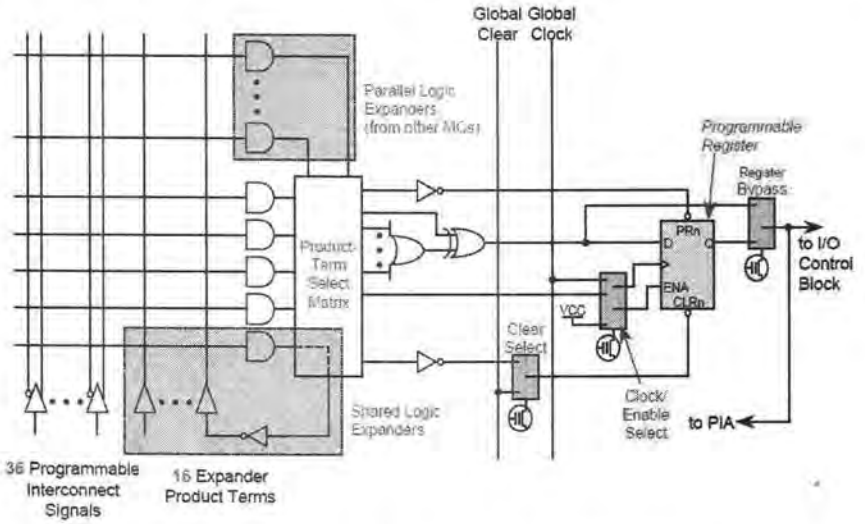


Fig. 2.5. Macrocelda de un CPLD MAX7000 [10].

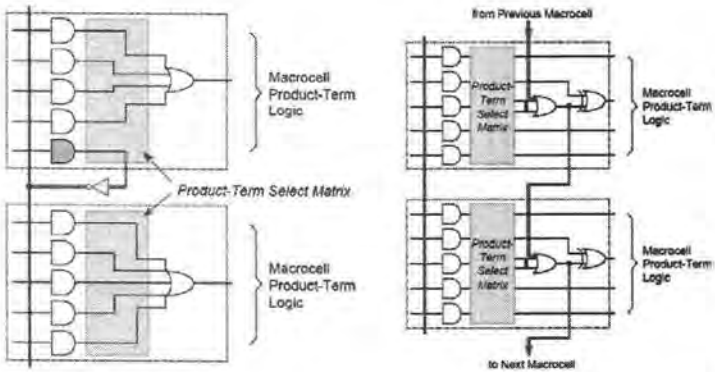


Fig. 2.6. Ampliadores para compartir recursos entre macroceldas [10].

La elección de un CLPD está en función del número de bloques lógicos, macroceldas, pines de E/S y velocidad entre otros. La tabla 2.1 muestra las características de algunos CPLD de la familia MAX7000 de Altera.

	EPM7032S	...	EPM7128S	...	EPM7256S
Puertas	600		2.500		5.000
Macrocells	32		128		256
LABs	2		8		16
Pines I/O	36		100		164

Tabla 2.1. Características de los dispositivos MAX7000 [10].

En la tabla pueden observarse el número de compuertas, macroceldas, bloques de arreglos lógicos y pines de E/S.

2.2 FPGA's

Los dispositivos lógicos programables de mayor densidad son los denominados Arreglos de compuertas programables en campo (FPGA). Estos dispositivos constan básicamente de tres elementos: bloques lógicos configurables (CLB), bloques de entrada y salida (IOB) y canales de comunicación. Existen algunos dispositivos FPGA que incluyen bloques de memoria configurable e incluso otros más avanzados que contienen microprocesadores embebidos (por ejemplo, la familia Virtex II Pro de Xilinx que llega a tener hasta 4 procesadores PowerPC embebidos en el chip o la familia excalibur de Altera que contiene procesadores ARM).

2.2.1 Arquitectura FPGA

La figura 2.7 muestra la arquitectura básica de un FPGA. Pueden observarse los elementos básicos antes descritos. Los FPGA contienen una matriz de bloques lógicos idénticos, normalmente, de forma cuadrada conectados vertical y horizontalmente [7].

Los bloques lógicos permiten la implementación de funciones lógicas y habitualmente están hechos de tablas de verdad denominadas LUT's (look-up tables) (o multiplexores) y flip-flops. La figura 2.8 Muestra la forma de implementar una función con este tipo de métodos.

Existen muchas configuraciones de cada uno de los elementos principales de los FPGA's, la arquitectura depende de la familia a la que pertenezca el dispositivo y ésta a su vez depende del fabricante. Hasta este momento se han dado aspectos generales que aplican a todos los PLD's, por lo tanto no es de vital importancia mostrar o analizar los diagramas para una familia específica. Por ejemplo, la figura 2.9 muestra la arquitectura de la familia Virtex II de Xilinx, donde se puede

observar que cuenta con los elementos citados anteriormente aunque con algunas diferencias que mejoran el desempeño respecto a otros FPGA's.

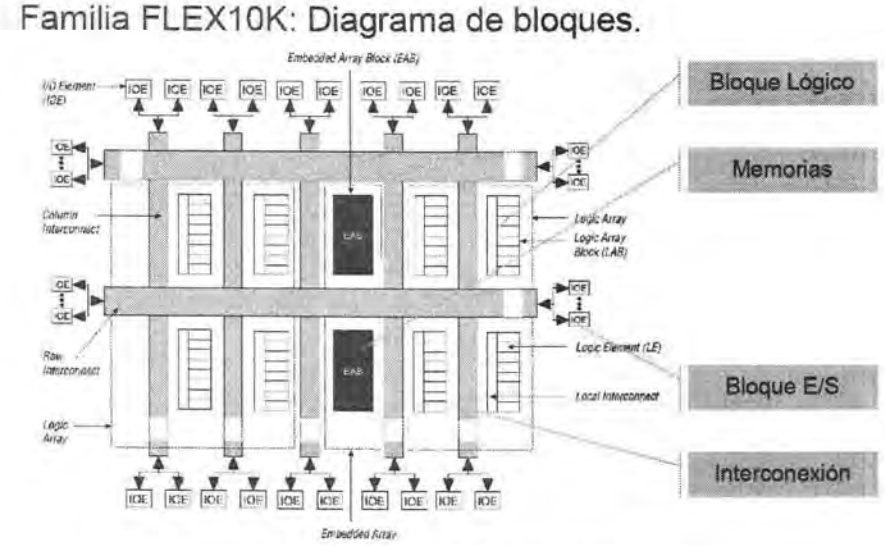


Fig. 2.7. FPGA FLEX 10K [10].

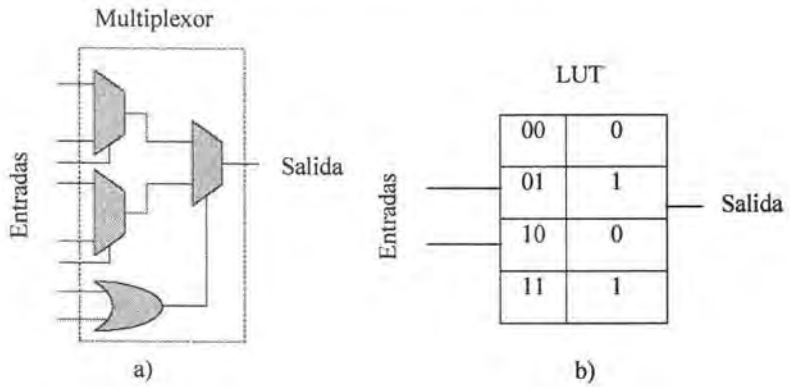


Fig. 2.8. Implementación de funciones lógicas con multiplexores y LUT's.

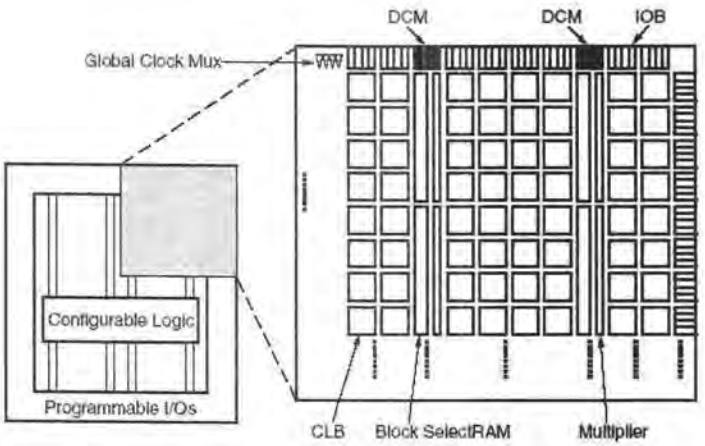


Fig. 2.9. Arquitectura de la familia Virtex II de Xilinx [11].

En la figura 2.9 puede observarse que el dispositivo cuenta con bloques lógicos configurables, bloques de entrada y salida, canales de interconexión, bloques de memoria y esta familia cuenta con bloques de manejo de reloj digital (DCM) y bloques multiplicadores.

Los bloques de E/S de la familia Virtex II se muestran en la figura 2.10. Estos bloques contienen 6 elementos de almacenamiento. Cada uno de estos elementos puede ser configurado como un flip-flop D o como un latch. En la entrada, salida y selector de 3-estados pueden ser utilizados uno o dos registros DDR. La transmisión de datos a doble tasa (DDR) es realizada por los dos registros en cada lado disparados por los flancos del reloj.

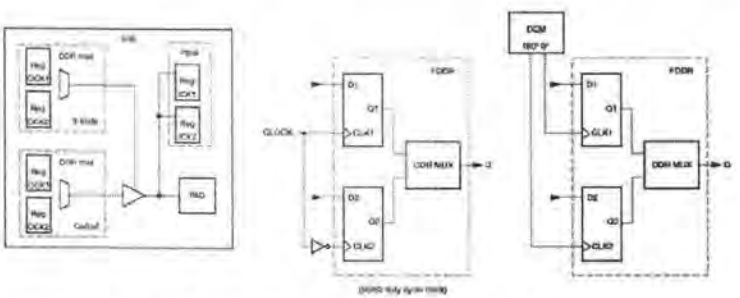


Fig. 2.10. IOB de la familia Virtex II [11].

La figura 2.11, muestra cada uno de los elementos CLB de la misma familia. Estos bloques están organizados en un arreglo y son utilizados para construir diseños lógicos combinacionales y síncronos.

Un elemento CLB consta de 4 partes similares con una realimentación local rápida dentro del mismo CLB. Cada parte incluye dos generadores de funciones de 4 entradas, acarreo lógico, compuertas aritméticas lógicas, multiplexores y dos elementos de almacenamiento como se muestra en la figura 2.12.

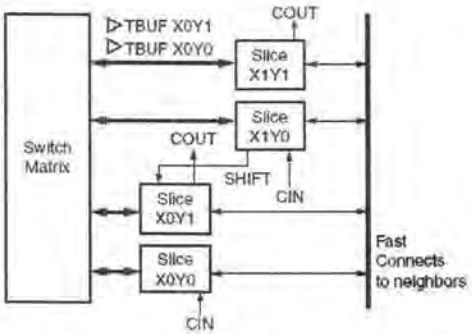


Fig. 2.11. CLB de la familia Virtex II [11].

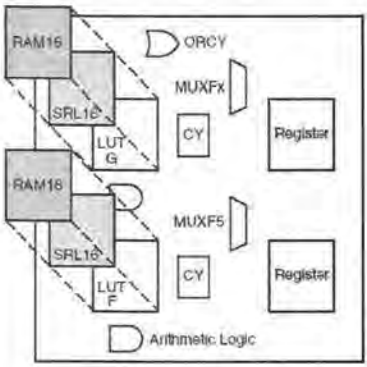


Fig. 2.12. Estructura de un componente (Slice) del CLB [11].

Cada generador de función es programado como una LUT de 4 entradas, memoria RAM de 16 bits distribuida o un registro de 16 bits.

Estos bloques lógicos difieren un poco a los bloques de otras familias aunque en esencia cuentan con los mismos componentes básicos. El componente principal de un CBL es la tabla de verdad LUT con la que se genera la función lógica. Las figuras 2.13 y 2.14 muestran la configuración de los CBL para la familia FLEX10K de Altera.

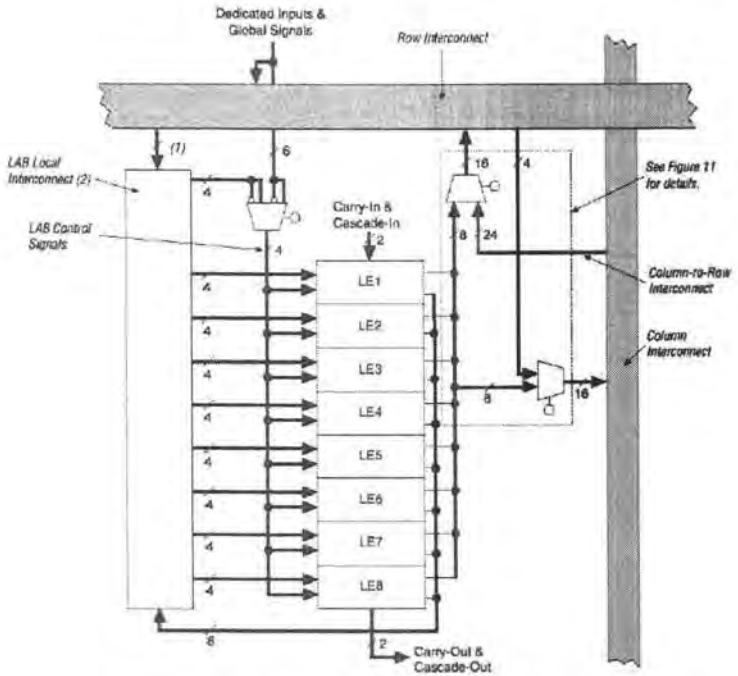


Fig. 2.13. CLB de la familia FLEX10K [10].

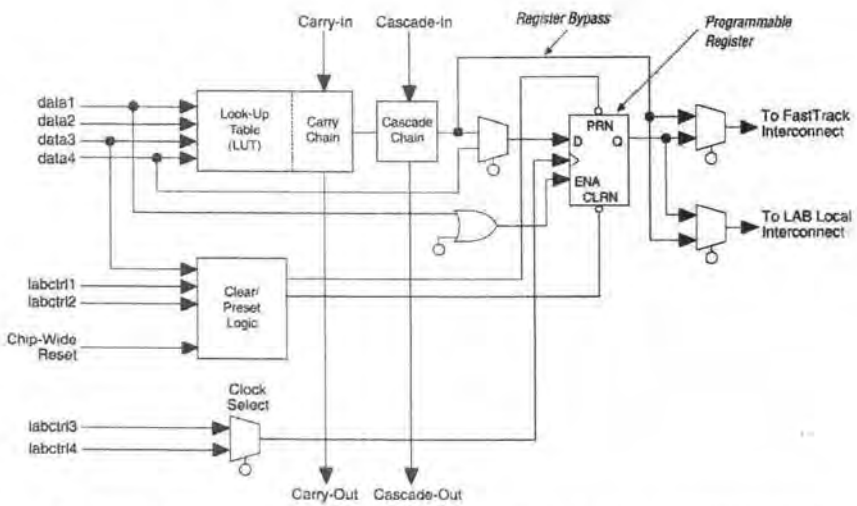


Fig. 2.14. Elemento Lógico (LE) del CBL de la familia FLEX10K [10].

Como puede observarse, en ambas arquitecturas los CLB se dividen en componentes similares que contienen tablas generadoras de funciones (LUT) y cuentan con otros componentes, principalmente multiplexores y registros. Las salidas en ambos esquemas se conectan a una bus rápido local y a la interconexión programable. Estos esquemas difieren un poco al de los dispositivos CPLD aunque en esencia cumplen la misma función de generar la operación lógica correspondiente y tener una salida de almacenamiento con interconexión a otros niveles.

La selección de un dispositivo FPGA es un poco más compleja que un CPLD o cualquier otro tipo de PLD ya que los fabricantes suelen dar información más detallada y específica de cada familia. La información incluye número de compuertas, registros, arreglos lógicos, memoria, multiplicadores y otros componentes embebidos que dependen de cada familia de dispositivos así como velocidad y pines de E/S.

Se puede encontrar información detallada y específica para cada dispositivo en sus hojas de especificaciones que provee cada fabricante, a continuación se muestra una tabla con distintas familias donde se puede comparar las diferencias entre estos tipos de dispositivos.

Entre las diferentes familias de la tabla 2.2 se puede observar que contienen elementos en común y elementos que sólo pertenecen a una familia específica y sus capacidades. Actualmente la familia más avanzada de la compañía Xilinx es la Virtex-4 con densidades superiores al dispositivo con mayor capacidad Virtex II, teniendo por ejemplo, hasta 200000 celdas lógicas a comparación de las 105000 del XC2V8000 el cual contiene alrededor de 8000k compuertas lógicas.

Claramente se pueden ver las capacidades de los dispositivos FPGA y de ahí su importancia en el diseño e implementación de circuitos digitales altamente complejos o circuitos que necesiten un extensivo uso de recursos, todo integrado en un mismo chip.

Como ya se mencionó, se deben tener varias consideraciones antes de elegir un PLD, entre las más importantes se encuentran: Número de compuertas y registros, RAM, bloques adicionales internos (como multiplicadores y procesadores embebidos), velocidad, número de pines de entrada y salida, configuración (ya sea dentro o fuera del sistema), programación y herramientas de diseño, tensión de alimentación, precio, etc. Aunque, sin duda alguna, si es posible encontrar el tipo de dispositivo adecuado para cada aplicación debido a la gran variedad existente.

Device	XC 3S50	XC 3S200	XC 3S400	XC 3S1000	XC 3S1500	XC 3S2000	XC 3S4000	XC 3S5000
System Gates	50K	200K	400K	1000K	1500K	2000K	4000K	5000K
Logic Cells	1,728	4,320	8,064	17,280	29,952	46,080	62,208	74,880
18x18 Multipliers	4	12	16	24	32	40	96	104
Block RAM Bits	72K	216K	288K	432K	576K	720K	1,728K	1,872K
Distributed RAM Bits	12K	30K	56K	120K	208K	320K	432K	520K
DCMs	2	4	4	4	4	4	4	4
I/O Standards	23	23	23	23	23	23	23	23
Max Differential I/O Pairs	56	76	116	175	221	270	312	344
Max Single Ended I/O	124	173	264	391	487	565	712	784

Familia Espartan 3

Feature/Product	XC 2VP2	XC 2VP4	XC 2VP7	XC 2VP20	XC 2VPX20	XC 2VP30	XC 2VP40	XC 2VP50	XC 2VP70	XC 2VPX70	XC 2VP100
EasyPath cost reduction	-	-	-	-	-	XCE 2VP30	XCE 2VP40	XCE 2VP50	XCE 2VP70	XCE 2VPX70	XCE 2VP100
Logic Cells	3,168	6,768	11,088	20,880	22,032	30,816	46,632	53,136	74,448	74,448	99,216
BRAM (Kbits)	216	504	792	1,584	1,584	2,448	3,456	4,176	5,904	5,544	7,992
18x18 Multipliers	12	28	44	88	88	136	192	232	328	308	444
Digital Clock Management Blocks	4	4	4	8	8	8	8	8	8	8	12
Config (Mbits)	1.31	3.01	4.49	8.21	8.21	11.36	15.56	19.02	26.1	26.1	33.65
PowerPC Processors	0	1	1	2	1	2	2	2	2	2	2
Max Available 3.125 Gbps RocketIO Transceivers*	4	4	8	8	0	8	12*	16*	20	0	20*
Max Available 10.3125 Gbps RocketIO X Transceivers*	0	0	0	0	8	0	0	0	0	20	0
Max Available User I/O*	204	348	396	564	552	644	804	852	996	992	1164

Familia Virtex II Pro

Tabla 2.2. Caracteristicas de las familias Espartan 3 y Virtex II Pro de Xilinx.

2.3 Configuración de los PLD's

Una característica que no se debe pasar por alto es la forma en la que se puede configurar el tipo de PLD que se requiera para así poder utilizarlo. Una ventaja de estos dispositivos es que no se requiere de muchos recursos para el diseño y la implementación. Normalmente solo se necesita una computadora, software adecuado y la interfaz (programador) para el dispositivo a utilizar.

Existen dos tipos de configuración, la *no volátil* en la que se utilizan dispositivos con tecnologías PROM, EPROM, EEPROM, etc., que al programarse la configuración es retenida si el dispositivo es

desconectado de la alimentación y la *volátil* que se implementa con RAM.

Así mismo existen dos modos de programación, el primero es para los dispositivos antiguos o muy sencillos donde el chip físicamente es separado de la tarjeta e insertado en un programador especial y el segundo donde no es necesario desconectar físicamente al circuito. A este método suele llamársele ISP (programación en sistema) y requieren de pines especiales para la operación.

Existen distintos métodos de diseño para configurar un PLD. Se puede diseñar con ecuaciones boleanas, diagramas o esquemáticos, máquinas de estado, tablas de verdad, lenguajes de descripción de hardware o una combinación de ellos. El método depende mucho de la herramienta que se utilice y de los modos que ofrezca. Actualmente existen muchos programas que pueden ser utilizados para diseñar con dispositivos PLD, unos son generales y otros específicos para ciertas familias, dependiendo de la empresa desarrolladora del software. Algunos programas son PALASM, OPAL, PLP, ABEL, CUPL, etc. [7].

PALASM (ensamblador PAL) fue creado por AMD para dispositivos PAL; acepta formato de ecuaciones boleanas. OPAL (lenguaje de optimización para arreglos programables) fue creado por National Semiconductor para dispositivos PAL y GAL; acepta formatos de ecuaciones boleanas, máquinas de estado, tablas de verdad o una combinación de ellos. PLPL (lenguaje de programación de lógica programable) creado por AMD, acepta formatos de ecuaciones boleanas, tablas de verdad, diagramas de estado y sus combinaciones. ABEL (lenguaje avanzado de expresiones boleanas) creado por Data I/O Corporation, acepta formatos de ecuaciones boleanas, tablas de verdad y diagramas de estado. CUPL (compilador universal de lógica programable) también creado por AMD fue planeado para el desarrollo de diseños complejos. Programa varios tipos de PLD.

Todos estos programas de diseño lógico tienen la función de procesar y sintetizar el diseño que se va a introducir en un PLD.

El primer método para configurar un PLD es con ecuaciones boleanas que describen la funcionalidad del diseño. El formato depende de la herramienta que se esté utilizando, así como otras características propias de la sintaxis del sintetizador. De aquí surgieron los lenguajes de descripción de hardware (HDL). EL siguiente listado muestra un ejemplo de diseño con ecuaciones boleanas para PALASM:

```
F1 = /A*B+/A*C

F2 = A*B+B*/C
```

Que equivalen a las expresiones boleanas:

$$F1 = \overline{A}B + \overline{A}C$$
$$F2 = AB + B\overline{C}$$

Para los dispositivos pequeños se puede realizar todo el diseño en un solo archivo que contiene la información de la conexión de cada pin así como las ecuaciones que describen el funcionamiento del circuito, incluso, se pueden realizar simulaciones en el mismo. Normalmente estos programas también aceptan descripción de máquinas de estados, por ejemplo, CUPL tiene el siguiente formato:

```
Encabezado:
        Name   XXXXX;
        Partno XXXXX;
        Date
        Revision
        Designer
        Company
        Assembly
        Location
Declaraciones:
        PIN [1,2] = [A,B] ;     /* Entradas */
        PIN 3 = !enable ;               /* Entrada*/
        PIN [12..15] = [Y0..3] ;        /* Salidas Decodificadas */
        PIN 14 = no_match ;
Cuerpo principal:
        CONDITION {
                IF enable & !B & !A out Y0;
                IF enable & !B & A out Y1;
                IF enable & B & !A out Y2;
                IF enable & B & A out Y3;
                default no_match;
        }
```

Existe otro método de diseño en el que no se utilizan ecuaciones y no es necesario saber algún lenguaje de programación. No todas las herramientas soportan este tipo de entrada: esquemáticos. Este tipo de diseño es gráfico y los distintos componentes o compuertas se conectan como si se estuviese diseñando en papel. Aquí se asignan nombres a las entradas y salidas y se insertan las compuertas o bloques para posteriormente realizar la conexión de cada una de las señales. La figura 2.15 muestra un ejemplo de un diseño con esquemáticos.

Se pueden realizar diseños completos con este tipo de entrada o hay herramientas que soportan la combinación de todos los tipos de formatos. Este es el tipo más fácil de diseño puesto que se cuenta ya con los elementos a conectar, sin embargo, los archivos generados por un sintetizador no necesariamente los reconoce otro sintetizador.

Estas herramientas llegan a contar con una biblioteca donde están definidos componentes comerciales, esto permite que en vez de utilizar compuertas genéricas tipo NAND, se utilice, por ejemplo, un circuito 74LS00. También llegan a permitir entradas de máquinas de estados de forma gráfica, aunque cada una tiene una forma de especificar los estados, las transiciones entre ellos y las salidas.

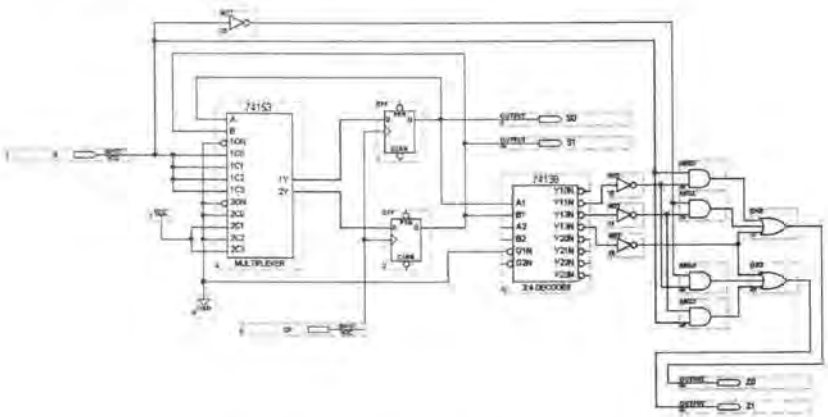


Fig. 2.15. Diseño lógico con esquemáticos.

Finalmente se encuentran los denominados lenguajes de descripción de hardware, que permiten implementar el diseño a través de una descripción que puede ser a alto nivel, no necesariamente con ecuaciones boleanas.

Existen muchos lenguajes como CUPL, ABEL, VHDL, AHDL, Verilog, etc., pero solo dos han sido aceptados como lenguajes estándar y son VHDL y Verilog, así que solo se estudiará a grandes rasgos las características del VHDL por ser el lenguaje elegido para la programación de la arquitectura del tema de esta Tesis, el fin es ilustrar la manera en la que se pueden programar estos dispositivos.

2.4 Introducción a VHDL

VHDL proviene de las siglas VHSIC (Very High Speed Integrated Circuit Hardware Description Language). VHDL es un lenguaje de descripción y modelado diseñado para describir la funcionalidad y la organización de sistemas hardware digitales, placas de circuitos, y componentes. El VHDL fue desarrollado de forma muy parecida al ADA debido a que el ADA fue también propuesto como un lenguaje puro pero que tuviera estructuras y elementos sintácticos que permitieran la

programación de cualquier sistema hardware sin limitación de la arquitectura [8].

Existen muchas ventajas al programar con VHDL, la primera es que se pueden programar cualquier tipo de dispositivos, como es un estándar, los sintetizadores no tienen problema en aceptar el lenguaje, aunque VHDL no solo fue creado para programar dispositivos PLD. El código puede ser reutilizable y así realizar proyectos en menor tiempo y con menor costo. Los diseños tienen independencia de los dispositivos y de las tecnologías utilizadas. VHDL permite diseñar, modelar y comprobar un sistema desde un alto nivel, hasta el nivel de definición estructural de compuertas. También permite diseño por bloques permitiendo la división en estructuras más pequeñas y simples.

2.4.1 Elementos básicos de diseño

La estructura general de un programa VHDL está formada por módulos de diseño, cada uno de ellos compuesto por un conjunto de declaraciones e instrucciones que definen, describen, estructuran, analizan y evalúan el comportamiento del sistema.

Existen cinco tipos de unidades de diseño: **declaración de entidad**, **arquitectura**, **configuración**, **declaración del paquete** y **cuerpo del paquete**. En el diseño se pueden ocupar tres de los cinco módulos pero dos de ellos (entidad y arquitectura) son indispensables en la estructuración de un programa.

Las declaraciones de entidad, paquete y configuración son unidades de diseño primarias, mientras que arquitectura y cuerpo del paquete son secundarias ya que necesitan de una unidad primaria.

2.4.2 Entidades

La entidad (**entity**) es el bloque elemental de diseño en VHDL. Una entidad es un elemento o componente electrónico (contador, multiplexor, sumador, compuerta, memoria, registro, etc.) que forma de manera individual o en conjunto un sistema digital. La entidad define las entradas y salidas.

Cada señal de entrada o salida de una entidad es referida como puerto, que es similar a un pin de un circuito. Todos los puertos declarados deben tener un nombre, modo y tipo de dato. Junto con la palabra *entity* hay asociados dos campos: *generic*: utilizado para pasar parámetros a la entidad y *port*: utilizado para definir las señales que relacionan la entidad con el resto del diseño. El siguiente listado muestra la forma de declarar una entidad:

```
ENTITY contador IS
GENERIC(N:integer:=10);
PORT(
    clk: in BIT;
      rst: in BIT;
      count: out BIT_VECTOR(N-1 DOWNTO 0);
);
END contador;
```

El código representa una descripción de las entradas y salidas de un contador de tamaño 10. La gran ventaja de esta declaración es que es posible redefinir el parámetro *N* tantas veces como sea necesario y extender el contador a la longitud que convenga en cada momento sin modificar el código, aunque en la forma más básica de una entidad solo en necesario definir la parte de entradas y salidas, es decir, el campo *port* que indica las señales que relacionan a la entidad con el resto del diseño, la sintaxis es:

```
Nombre_señal : modo tipo_de_señal;
```

El nombre de la señal identifica unívocamente a la señal en cuestión. No puede existir otra señal dentro de la arquitectura con el mismo nombre. El modo de la señal indica el sentido del flujo de la misma y puede tomar los siguientes valores:

IN: indica que la señal es una entrada.

OUT: indica que la señal es una salida.

INOUT: indica que la señal es una entrada o salida.

BUFFER: es una de salida con realimentación a la misma entidad.

Una señal *in* recibe sus valores desde el exterior de la entidad. Por tanto, no puede ser reasignada en el interior de la entidad, es decir no puede aparecer a la izquierda de una asignación en la arquitectura. Una señal *out* genera valores al exterior de la entidad. No puede ser asignada a ninguna otra señal en la arquitectura. Una señal *inout* puede ser asignada en ambos sentidos y es responsabilidad del diseñador determinar en que condiciones de la función lógica descrita la señal puede ser

in o *out*. Una señal *buffer* es señal de salida asignable a otra señal en la arquitectura.

Por otra parte, los tipos de señal son los valores que se establecen para los puertos de entrada y salida dentro de la entidad. Se asignan de acuerdo a las características del diseño. Algunos tipos son:

BIT: toma valores binarios *1* o *0*.

BOOLEAN: define valores de *verdadero* o *falso* en una expresión.

BIT_VECTOR: es un vector de bits que representa un conjunto de los mismos.

INTEGER: representa un número entero.

En el listado anterior, se define un contador con dos señales de entrada, *clk* y *rst* y una señal de salida *count*. Las entradas son binarias mientras que la salida es un vector de tamaño *N*. La palabra reservada **end** determina el final de la declaración seguida del nombre de la entidad. La siguiente figura muestra un diagrama de la entidad.

Fig. 2.16. Entidad contador.

En cuanto a la sintaxis, los nombres de las señales, declaraciones y etiquetas en general, se deben formar con caracteres alfanuméricos y guiones bajos. Pueden tener cualquier longitud pero no deben comenzar con número sino con letra. No se permiten dos guiones bajos seguidos ni otros tipos de caracteres. Tampoco se pueden utilizar las palabras reservadas. VHDL considera las mayúsculas y las minúsculas iguales. Se pueden escribir comentarios insertados en el código con doble guión "--", todo el texto que se encuentre después del doble guión es un comentario.

Todos los números se consideran de base 10 a menos que se especifique lo contrario. Para poner un número en otra base se utiliza el símbolo "#", por ejemplo, 2#1100#, 16#C# representan el 12. Las cadenas de bits se incluyen entre comillas y se puede especificar la base, por ejemplo, B"1100", X"C".

Regresando a la declaración de entidades, la mayoría de veces se necesitan manejar datos en forma conjunta (buses), éstos se pueden manipular de forma sencilla agrupándolos en conjuntos o vectores. Existen dos formas de declarar estos vectores utilizando la sentencia **bit_vector**:

```
nombre : modo BIT_VECTOR(Max downto Min)
nombre : modo BIT_VECTOR(Min to Max)
```

La diferencia en ambos modos radica en que en el primero el primer elemento es el bit más significativo y el último es el bit menos significativo mientras que la segunda declaración es opuesta. Es importante definir de forma adecuada los vectores para que no se

invierta el orden de los bits a la hora de asignar las señales. El siguiente listado muestra la forma de declarar un multiplexor de 2 a 1 con buses de 16 bits.

```
Entity Mux2a1x16 IS
Port(
        A:      in  bit_vector(15 downto 0);
        B:      in  bit_vector(15 downto 0);
        Sal:    out bit_vector(15 downto 0);
        Sel:    in  bit;
);
End Mux2a1x16;
```

2.4.3 Paquetes

Así como otros lenguajes de programación, VHDL permite el uso de bibliotecas y paquetes que permiten declarar y almacenar estructuras lógicas para posteriormente utilizar esos recursos en el diseño. En VHDL se encuentran definidas dos bibliotecas llamadas **ieee** y **work**. En *ieee* se encuentra el paquete *std_logic_1164* mientras que en *work* se encuentran las declaraciones del usuario.

Un paquete es un elemento de diseño que permite desarrollar un programa de manera ágil, debido a que contiene algoritmos preestablecidos que ya tienen optimizado su comportamiento, como sumadores, contadores, etc. Un paquete es una unidad de diseño formada por declaraciones, programas, componentes y subprogramas. Para utilizar un paquete es necesario incluir la biblioteca donde se encuentra con la siguiente declaración:

***library** ieee;*

Así, se permite utilizar todos los componentes incluidos en la biblioteca. Para el caso especial de la biblioteca de trabajo *work*, no se requiere la declaración **library** debido a que siempre está presente en el desarrollo del diseño.

El paquete *std_logic_1164* contiene todos los tipos de datos que suelen emplearse en el lenguaje como *std_logic* que es un tipo de datos más general que *bit*, ya que a una señal se le pueden asignar valores de *1*, *0* y *Z* (alta impedancia) entre otros. El acceso a la información contenida en un paquete se realiza a través de la sentencia **use** especificando el nombre de la biblioteca y el paquete:

use nombre_de_biblioteca.nombre_de_paquete.**all**;

Otros paquetes de la biblioteca ieee son:

- Paquete *numeric_std* que define funciones para realizar operaciones entre diferentes tipos de datos, además los tipos pueden representarse con o sin signo.
- Paquete *numeric_bit* que define tipos de datos binarios con o sin signo.
- Paquete *std_arith* que define funciones y operadores aritméticos, como igual, mayor que, menor que, etc.

2.4.4 Arquitecturas

Una arquitectura (**architecture**) es la estructura que define o describe el funcionamiento de una entidad, de tal manera que permita el desarrollo de los procedimientos que se realizarán. Una arquitectura siempre está referida a una entidad concreta por lo que no tiene sentido hacer declaraciones de arquitectura sin especificar la identidad.

La declaración de una arquitectura se realiza mediante la palabra reservada **architecture** y su sintaxis es:

```
ARCHITECTURE nombre_arquitectura OF nombre_entidad IS
        Declaraciones
BEGIN
        Instrucciones
END nombre_arquitectura;
```

Antes de definir la funcionalidad en el bloque *begin...end*, hay una parte donde se definen los subprogramas (funciones, procedimientos, etc.), declaraciones de tipo, declaraciones de constantes, declaraciones de señales, declaraciones de componentes, etc. Es importante destacar que las señales sólo pueden ser declaradas dentro de la parte declarativa de una arquitectura.

Una ventaja del lenguaje VHDL es que presenta varias formas de describir el funcionamiento de una entidad, los estilos de programación utilizados en el diseño de arquitecturas se clasifican como:

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

Descripción funcional

La descripción funcional expone la forma en la que trabaja el sistema sin importar cómo está organizado por dentro. Las descripciones consideran las relaciones entre las entradas y las salidas del circuito. Por ejemplo, a continuación se muestra el listado que declara un multiplexor de dos entradas y una línea de selección, el código define el funcionamiento en forma funcional:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Mux2a1x16 is
    Port ( A    : in  std_logic_vector(15 downto 0);
           B    : in  std_logic_vector(15 downto 0);
           Sal  : out std_logic_vector(15 downto 0);
           Sel  : in  std_logic);
end Mux2a1x16;

architecture ArqMux2a1x16 of Mux2a1x16 is
begin

        process (A, B, Sel)
        begin
                if (Sel = '0') then
                        Sal <= A;
                else
                        Sal <= B;
                end if;
        end process;

end ArqMux2a1x16;
```

En la primera línea del listado se incluye la biblioteca ieee. Las líneas siguientes se cargan los paquetes de tipos de datos y operaciones aritméticas que están contenidos en la biblioteca.

La declaración de la entidad es similar a la que se había dado de ejemplo, la diferencia es que se utilizan los tipos de datos *std_logic* y *std_logic_vector* en vez de *bit* y *bit_vector*. El nombre de la arquitectura es *ArqMux2a1x16* y está referida a la entidad *Mux2a1x16*.

En este caso se declara un proceso (**process**) que se utiliza para la definición de algoritmos. Seguido de la palabra *process* se añade una lista sensitiva (opcional) que sirve para indicar qué señales disparan el proceso. En este caso, el cambio de cualquiera de las señales de entrada es motivo para que se ejecute el proceso que define la funcionalidad del circuito. Si en la lista sensible no se añaden las señales *A* y *B*, por ejemplo, al existir un cambio en éstas no se reflejaría a la salida, sólo cuando exista un cambio en *Sel* se evaluaría la expresión con los valores actuales de todas las señales.

El proceso se ejecuta mediante declaraciones secuenciales, en este caso del tipo **if-then-else**. En este caso las asignaciones se parecen mucho a las asignaciones hechas en los lenguajes de programación convencionales. Se van evaluando las líneas conforme aparecen en el programa. Con este tipo de método se puede definir le comportamiento de cualquier sistema, desde una compuerta hasta un componente o circuito complejo. A continuación se muestran varias formas de implementar una compuerta OR:

```
architecture ArqCompOR of CompOR is
begin

        process (A, B)
        begin
                if (A = '0' and B= '0') then
                        Sal <= '0';
                else
                        Sal <= '1';
                end if;
        end process;

end ArqCompOR;


architecture ArqCompOR of CompOR is
begin

        process (A, B)
        begin
                if (A = '1' or B= '1') then
                        Sal <= '1';
                else
                        Sal <= '0';
                end if;
        end process;

end ArqCompOR;
```

En ambos casos se considera que se ha hecho una declaración de una entidad y de una arquitectura con señales de entrada *A* y *B* y una señal de salida *Sal* todas de tipo *bit* o *std_logic*. Se puede observar que ambos procesos son similares, simplemente se cambió la expresión lógica de la sentencia *if*, al negado, por lo tanto, se invirtieron las asignaciones de las señales de salida.

Cuando se tiene más de una condición que se encarga de seleccionar una asignación, se puede utilizar la palabra **elsif**. La sintaxis es la siguiente:

```
if (condición) then
        asignación_1;
elsif (condición_2) then
        asignación_2;
else
        asignación_3;
end if;
```

Por ejemplo, si se tiene un multiplexor de 4 señales de entrada (*A*, *B*, *C* y *D*) y una señal de salida (*Sal*), se puede utilizar el siguiente código:

```
entity Mux4a1x1 is
   Port ( A     : in  std_logic;
          B     : in  std_logic;
          C     : in  std_logic;
          D     : in  std_logic;
          Sal   : out std_logic;
          Sel   : in  std_logic_vector(1 downto 0));
end Mux4a1x1;

architecture ArqMux4a1x1 of Mux4a1x1 is
begin

        process (A, B, Sel)
        begin
                if (Sel = "00") then
                        Sal <= A;
                elsif (Sel = "01") then
                        Sal <= B;
                elsif (Sel = "10") then
                        Sal <= C;
                else
                        Sal <= D;
                end if;
```

```
        end process;

end ArqMux4a1x1;
```

También se pueden realizar asignaciones con la sentencia **case-when**, la sintaxis es:

```
CASE señal_condicional IS
        WHEN "cond1" => asignación_1;
        WHEN "cond2" => asignación_2;
        .
        .
        .
        WHEN OTHERS => asignación;
END CASE;
```

Para el ejemplo anterior la asignación sería:

```
architecture ArqMux4a1x1 of Mux4a1x1 is
begin

        process(A, B, C, D, Sel)
        begin
                case Sel is
                        when "00" => Sal <= A;
                        when "01" => Sal <= B;
                        when "10" => Sal <= C;
                        when others => Sal <= D;
                end case;
        end process;

end ArqMux4a1x1;
```

VHDL permite muchos tipos de asignaciones para las señales. Se pueden utilizar operadores aritméticos (+, -, *, /, etc.). Se pueden consultar las palabras reservadas y los operadores en el apéndice A.

Descripción por flujo de datos

La descripción de flujo por datos indica la forma en que los datos se pueden transferir de una señal a otra sin necesidad de declaraciones secuenciales. Este tipo de descripciones permite definir el flujo que tomarán los datos entre módulos encargados de realizar operaciones.

La instrucción básica de la ejecución concurrente es la asignación entre señales que se realiza con el operador "**<=**". Para facilitar la tarea de realizar asignaciones concurrentes, VLDH introduce algunos elementos de alto nivel como son instrucciones condicionales, de selección, etc.

Una instrucción condicional que utiliza el lenguaje para asignar valores a señales de forma concurrente es **when-else**, que es equivalente a la instrucción secuencial **if-then-else**. La sintaxis es la siguiente:

```
señal <= asignación_1 <= WHEN condición_1 ELSE asignación_2;
```

El código equivalente con una sentencia *if* sería:

```
if (condición) then
        señal <= asignación_1;
else
        señal <= asignación_2;
end if;
```

Las sentencias son muy similares, básicamente la diferencia estriba en que la instrucción *if* sólo tiene sentido dentro de un proceso mientras que la instrucción *when* sólo lo tiene fuera de él. Por ejemplo, el siguiente código muestra la forma de implementar el mismo multiplexor de 2 a 1 con buses de 16 bits:

```
entity Mux2a1x16 is
   Port ( A    : in  std_logic_vector(15 downto 0);
          B    : in  std_logic_vector(15 downto 0);
          Sal  : out std_logic_vector(15 downto 0);
          Sel  : in  std_logic);
end Mux2a1x16;

architecture ArqMux2a1x16 of Mux2a1x16 is
begin

        Sal <= A when Sel = '0' else B;

end ArqMux2a1x16;
```

Se puede observar que con este tipo de asignación la descripción es más sencilla, en una sola línea se puede describir el comportamiento

del circuito. En este tipo de asignación concurrente, también se permite la descripción funcional con ecuaciones boleanas, por ejemplo, para describir el comportamiento de una compuerta OR con entradas *A* y *B* y salida *Sal*, se puede realizar la siguiente asignación:

```
architecture ArqCompOR of CompOR is
begin

        Sal <= A or B;

end ArqCompOR;
```

En vez de poner todas las instrucciones que se necesitan para asignar la misma funcionalidad con un proceso. Pero para circuitos más complejos se pueden ver las bondades de la sentencia *if*, ya que si quisiéramos expresar su funcionamiento a nivel de compuertas tendríamos qué hacer un análisis para encontrar las expresiones boleanas que definen su comportamiento. Por otro lado, en las sentencias *if* se pueden realizar asignaciones de varias señales y en los casos *elsif* o *else* se pueden realizar asignaciones a otras señales, mientras que en las sentencias *when...else* solo se puede realizar la asignación a una señal específica, se necesita una sentencia para cada señal y por lo tanto la descripción de la condición se debe repetir para cada caso.

Existe otro tipo de asignación concurrente que se puede realizar con la sentencia **with...select**. Su sintaxis es la siguiente:

```
WITH señal_condicional SELECT
        señal_a_asignar <=   asignación_1 WHEN "condición1",
                             asignación_2 WHEN "condición2",

                             asignación WHEN others;
```

Por ejemplo, para describir el funcionamiento del multiplexor de 4 señales de entrada y una de salida se puede utilizar el siguiente código:

```
architecture ArqMux4a1x1 of Mux4a1x1 is
begin

        with Sel select
```

```
        Sal <= A when "00",
               B when "01",
               C when "10",
               D when others;

end ArqMux4a1x1;
```

Descripción estructural

Las sentencias estructurales basan su comportamiento en modelos lógicos establecidos. Con ellas se puede hacer uso de un componente o circuito definido con anterioridad sin necesidad de incluirlo en la descripción que se está realizando; sólo habrá que hacer una llamada a dicho componente para usarlo con las especificaciones propias del diseño actual. La descripción estructural se basa en la sentencia **port map** (mapa de puertos). Por ejemplo, podríamos construir un multiplexor de 4 señales de entrada con buses de 16 bits utilizando los multiplexores de 2 a 1 y 16 bits que ya hemos construido, la solución se muestra en la figura 2.18.

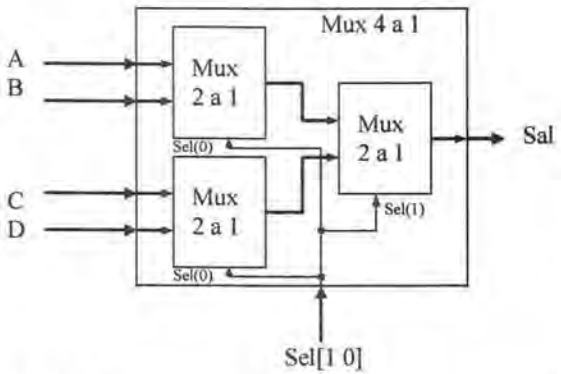


Fig. 2.18. Multiplexor de 4 a1 utilizando multiplexores de 2 a 1.

La sintaxis de la sentencia *port map* es:

```
etiqueta:    nombre_componente    PORT MAP (parámetros...);
```

El código siguiente muestra la forma de describir un circuito de forma estructural:

```
entity Mux4a1x16 is
   Port ( A    : in  std_logic_vector(15 downto 0);
          B    : in  std_logic_vector(15 downto 0);
          C    : in  std_logic_vector(15 downto 0);
          D    : in  std_logic_vector(15 downto 0);
          Sal  : out std_logic_vector(15 downto 0);
          Sel  : in  std_logic_vector(1 downto 0));
end Mux4a1x16;

architecture ArqMux4a1x16 of Mux4a1x16 is
signal SalMux1std_logic_vector(15 downto 0);
signal SalMux2std_logic_vector(15 downto 0);
begin

Mux1Entrada: Mux2a1x16    port map (A, B, SalMux1, Sel(0));
Mux2Entrada: Mux2a1x16    port map (C, D, SalMux2, Sel(0));
MuxSalida          Mux2a1x16    port map(SalMux1, SalMux2, Sel(1));

end ArqMux4a1x16;
```

En el ejemplo anterior se utiliza un módulo ya definido para construir otro más grande. Nótese la declaración de dos señales entre las líneas de *architecture* y *begin*, estas señales se utilizan para asignar los valores de salidas de los multiplexores que se encuentran en la entrada. Todas las señales que se utilicen para guardar valores de señales internas a una entidad se declaran en este lugar. Se pueden declarar cualquier tipo de datos, inclusive, datos definidos por el usuario.

Existen muchos conceptos más acerca del lenguaje VHDL, como el diseño jerárquico, declaraciones de bloques, subrutinas, lazos, tipos de datos, sobrecarga de operadores, configuraciones, variables, etc., aquí sólo se presentó un panorama muy general, si bien con esto se pueden realizar diseños, no se aprovecha al máximo las capacidades del lenguaje. Para una descripción más detallada se puede consultar cualquier libro especializado o las referencias recomendadas al final de este trabajo.

CAPÍTULO 3

Diseño de la arquitectura de un procesador neuronal

El estudio de las redes neuronales artificiales es tan amplio como su campo de aplicación. Los investigadores proponen distintos tipos de redes y algoritmos de aprendizaje, estudian su comportamiento y sus aplicaciones. Muchas aplicaciones de las redes neuronales solo se pueden llevar a cabo en computadoras personales por la gran cantidad de datos, porque el programa sería muy extenso o porque, aunque se puede programar un algoritmo en una tarjeta con cierto microcontrolador u otro tipo de procesador, éste resultaría muy lento para la mayoría de aplicaciones, más las que requieren una gran cantidad de procesamiento.

Existen muchas arquitecturas propuestas por muchos investigadores, estudiantes y en general desarrolladores que permiten el uso eficiente de algoritmos de RNA. Sin embargo, la mayoría de esos trabajos se enfocan a un solo tipo específico de red, algunos incluyen su algoritmo de entrenamiento. Otros trabajos se enfocan al diseño de arquitecturas paralelas, donde se obtienen resultados en tiempo real pero requieren de varios elementos procesadores (por ejemplo, PLD's o DSP's). Las referencias bibliográficas [13] a [18] son un ejemplo de trabajos que se han realizado utilizando dispositivos lógicos programables para el procesamiento de redes neuronales artificiales. Existe una gran cantidad de este tipo de trabajos, cada uno presenta soluciones a distintos problemas o aplicaciones.

En este trabajo se propone una arquitectura que si bien no es paralela, ofrece otro tipo de ventajas como el uso de un solo circuito (FPGA) en una tarjeta que puede ser muy simple y no contener elementos de memoria externos (entendiéndose que si se cuenta con

memoria externa se aumenta la capacidad de la arquitectura). Se cuenta con mucha flexibilidad a la hora de configurar el circuito ya que se puede programar casi cualquier tipo de topología de red, dependiendo de elementos extras con los que se cuente. Su ejecución es rápida, pudiendo programar muchas aplicaciones en tiempo real, dependiendo del tamaño de la red, de la velocidad del FPGA y de la frecuencia del reloj. Para este caso específico la arquitectura está probada en una tarjeta con una frecuencia de reloj de 20 MHz.

La arquitectura no cuenta con algoritmo de entrenamiento ya que está pensada para cualquier tipo de red, como se vio en el capítulo sobre RNA, existen muchos tipos de aprendizaje y algoritmos con los que se puede llevar a cabo. La red está pensada solo para su fase de ejecución aunque no se descarta un trabajo futuro en el que de alguna manera se puedan modificar los pesos de las sinapsis.

Finalmente, se aprovechan las ventajas que ofrecen las arquitecturas del tipo Pipeline en cuanto al número de operaciones que realizan al mismo tiempo. En este caso, no se cuentan con los niveles convencionales de búsqueda, decodificación, ejecución y almacenamiento [12]. El principio es el mismo pero los niveles tan solo se encargan de realizar una función en común que a grandes rasgos es una suma de productos.

Como se explicará más adelante, las limitaciones de la arquitectura están muy ligadas al hardware, que si bien debería ser independiente, existen elementos de los que no se puede independizar del todo como la memoria del sistema, la velocidad máxima de ejecución y los puertos (pines) de entrada y salida, principalmente.

Cabe señalar que la arquitectura del procesador neuronal por sí sola está completa, sin embargo, se recomienda el diseño de un interfaz especial para la comunicación entre el procesador y el exterior. Se propondrá una interfaz simple pero poderosa aunque está en fase de construcción y por eso no se incluye en este trabajo, sin embargo, si se muestran algunos ejemplos donde se ve lo sencillo que es el programar una interfaz que permita utilizar el procesador de forma adecuada dependiendo de la aplicación.

3.1 Elemento Básico

El objetivo es diseñar una arquitectura para un procesador digital en el que se ejecuten distintos tipos de redes neuronales de forma eficiente. La forma ideal es que el procesamiento de datos se realice de la misma manera que las redes neuronales biológicas: de forma paralela. Sin embargo, antes se tienen que definir ciertas características que debe cumplir el procesador para poder analizar las posibles soluciones.

Como ya se mencionó, el elemento básico de una red neuronal artificial es el modelo de la neurona que se muestra en la figura 3.1.

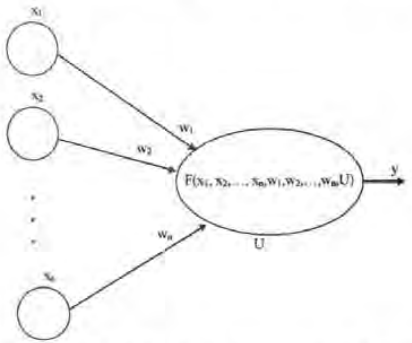


Fig. 3.1. Modelo de la Neurona Artificial.

Para este tipo específico de modelo de neurona, se realiza una suma ponderada de las entradas y se les aplica una función de activación o transferencia. Esta función puede ser de distinta índole dependiendo del tipo de red y no necesariamente se necesita de un umbral para todas las funciones de activación. En este modelo la salida es continua, por lo que no responde por pulsos y no se consideran retardos para generar la respuesta.

Por una parte se tienen los pesos que son números reales y por otra las entradas, que pueden tener valores fijos como 1 ó 0 pero en el caso más general, también son números reales así como la salida.

Así, el primer paso es decidir la naturaleza de la representación numérica del procesador. En este caso se ha elegido un procesador a punto fijo y con 16 bits de resolución, en este sentido, todos los valores serán representados en este formato, desde las entradas y pesos, hasta las salidas y por supuesto, la función de activación.

El siguiente aspecto a analizar es la topología de la red, existen muchos tipos de conexiones dependiendo del modelo que se trate. En un caso concreto se puede especificar una aplicación y una red específica que solucione el problema. En este caso se quiere dejar completa flexibilidad para la configuración de la topología de la red.

La figura 3.2 muestra solo algunas topologías de las redes neuronales. Se puede apreciar que no existe ningún patrón en común, por lo que tendrá que decidirse una forma de configuración en la que se especifiquen las conexiones o sinapsis de las neuronas, entradas y salidas, inclusive, se ha pensado una arquitectura que sea completamente flexible y que permita una topología completamente aleatoria de la red.

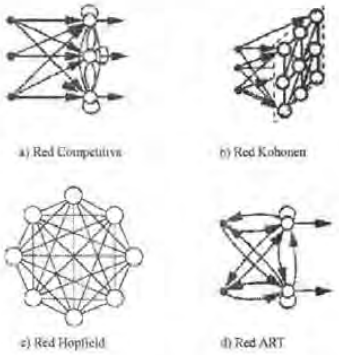


Fig. 3.2. Distintas topologías de las RNA.

Así mismo se debe pensar en la forma de implementar la función de activación, puesto que puede ser en forma general cualquier tipo de función. Existen muchas funciones comunes que se utilizan para los modelos de las neuronas, como las que se muestran en el apartado sobre redes neuronales. Por esta razón se decidió implementar la función de forma más general: con una tabla, donde la dirección pueda ser el dato de entrada y el valor sea el dato de salida.

Analizando estos puntos se puede pretender diseñar un modelo paralelo, sin embargo, cada neurona necesitaría de un número k_n de sumas de productos. Siendo k_n el número de entradas para la n-ésima neurona. Por lo tanto, se necesitarían de k_n multiplicadores por neurona. Para una red de tipo retropropagación, como la de la figura 1.14 se necesitan 10 multiplicadores, pero si se tiene una red, por ejemplo, del mismo tipo con 4 neuronas de entrada, 7 de una sola capa oculta, y 4 de salida; se necesitarían de 56 multiplicadores suponiendo que se tiene tan solo una entrada. Para una red tipo Hopfield que se pudiera utilizar en una aplicación real de reconocimiento de imágenes de 130x180 pixeles [4], se necesitarían de 547 536 600 multiplicadores. Inclusive para el caso de esta arquitectura que se ha pensado para un formato de

16 bits a punto fijo resultaría imposible implementar un procesador en un solo FPGA, de hecho, quizá no se pueda implementar ni siquiera en un gran conjunto de ellos.

Una solución es utilizar un bloque multiplicador por cada neurona, con lo cual ya no podría considerarse como una arquitectura paralela, más bien sería una arquitectura con un alto grado de paralelismo. Dependiendo de la topología de la red, podría ser una solución adecuada. Esta solución podría aplicarse a redes pequeñas, con algunas decenas de neuronas pero si volvemos al ejemplo de la red Hopfield, se necesitarían de 23 400 multiplicadores, lo cual sigue siendo imposible para implementar hasta en un arreglo de FPGA's.

Precisamente por este problema, se decidió por una arquitectura secuencial. La cual consta de tan solo un bloque multiplicador. Una solución diferente y más apropiada podría ser el diseño de una arquitectura con cierto número de multiplicadores, por ejemplo, los máximos que soporte cierto dispositivo. Si se tiene tan solo un bloque multiplicador, el número de ciclos que se necesitarán para procesar una sola vez una red será como mínimo de c_n*s, siendo c_n el número de ciclos que se necesitan para procesar una sinapsis y s el número total de sinapsis de la red en cuestión, también hay que contar los ciclos que se necesitan para traer los datos y almacenarlos.

Teniendo más bloques multiplicadores, se puede agilizar el ciclo proporcionalmente al número de bloques, si son dos, podría diseñarse una arquitectura dos veces más rápida que con un solo multiplicador, etc. En este aspecto entra otra problemática para los dispositivos FPGA's.

Como se vio en el apartado correspondiente, estos dispositivos básicamente cuentan con bloques lógicos, bloques de entrada y salida, bloques multiplicadores, bloques de memoria y una interconexión configurable. Se puede programar una memoria especial en estos dispositivos dentro de los bloques lógicos, pero la memoria es un componente lógico que consume una gran cantidad de recursos y más si se tratase de una memoria de acceso múltiple. En este caso es recomendable utilizar los bloques de memoria configurable aunque sean sólo de acceso doble o simple.

Por esta razón y las explicadas anteriormente se decidió una arquitectura secuencial con un solo bloque multiplicador-acumulador. La representación de este bloque se muestra en la figura 3.3.

A este bloque se le ha denominado NALU (*unidad aritmética y lógica neuronal*). Básicamente es una ALU que recibe como entrada el peso y la señal proveniente de otra neurona, realiza el producto y la acumulación, aplica la función neuronal y genera una salida. El bloque NALU se explica de forma más detallada en la sección 3.2.3.

Las señales de control indican cuando debe realizarse un producto o un producto-acumulación y cuando aplicar la función neuronal. Este bloque puede realizar las operaciones que se necesitan dentro de una red neuronal, sin embargo, se necesitan elementos exteriores que proporcionen de manera adecuada los datos a este bloque. La suma se realimenta internamente en el bloque, aquí se muestra con fines prácticos y para el entendimiento del funcionamiento del bloque.

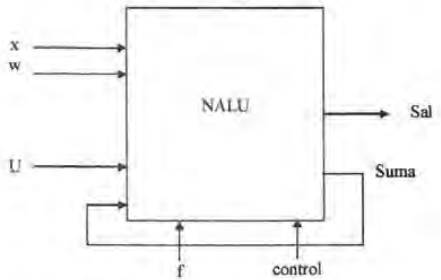


Fig. 3.3. Elemento básico de procesamiento.

3.2 Arquitectura propuesta

La arquitectura está basada en el funcionamiento de la NALU.

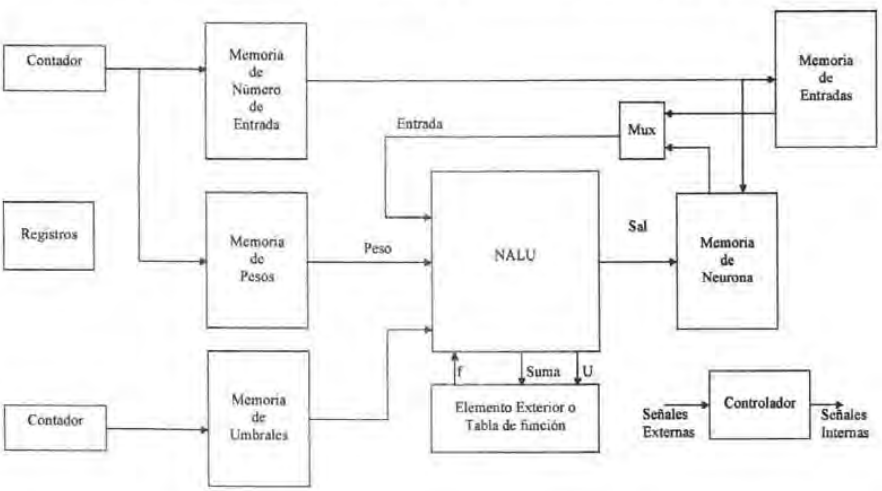


Fig. 3.4. Arquitectura del procesador neuronal.

En el diagrama a bloques de la figura 3.4 se puede explicar el funcionamiento de la arquitectura propuesta: Una vez configurada la red, se tiene un contador principal con el cual se van obteniendo directamente los pesos almacenados en una memoria e indirectamente la entrada. Como la topología de la red es configurable, cada conexión se guarda en una memoria (denominada memoria de número de entrada que especifica a qué neurona está conectada esa sinapsis o a qué entrada del exterior), teniendo la neurona o entrada a la cual está conectada la sinapsis, se recupera el valor el cual es almacenado en memorias específicas, de entrada o de neurona siendo esta última donde se guarda el estado actual de la red. Un multiplexor elige qué entrada es la correcta y ésta es alimentada al bloque NALU.

Este bloque ha recibido, por lo tanto, el peso y la entrada correspondiente a la sinapsis en evaluación. Un elemento de control es el que se encarga de manejar los contadores y las señales de control de la NALU dependiendo del estado actual y del valor de los registros. Finalmente, la NALU recibe la función de transferencia de una tabla exterior, que se puede configurar para utilizarse una memoria interna o un elemento externo al procesador, también existe un bloque específico con algunas funciones de activación preprogramadas.

Con todos estos elementos, se genera una salida que es almacenada en la memoria de neurona. El controlador decide qué operaciones realizar de acuerdo a las señales que recibe del exterior, se encarga de organizar las operaciones de toda la arquitectura.

Un aspecto importante de esta arquitectura es el número de memorias independientes utilizadas. En realidad la arquitectura utiliza 10 memorias RAM independientes con distintas características cada una. Esto es necesario por una razón fundamental: con este esquema la arquitectura procesa una sinapsis en un ciclo de instrucción y este parámetro es independiente de cualquier otro parámetro de la red. Siendo así, para una red con 20 millones de sinapsis, se necesitará de tan solo un segundo para actualizar el estado de la red por completo.

Más adelante se proporcionará el diagrama completo de la arquitectura. Antes se analizarán sus características y la forma de configurar las redes.

3.2.1 Características principales de la arquitectura

El procesador neuronal cuenta con las siguientes características:

- Secuencial: solo cuenta con un bloque NALU.
- Pipeline de 6 niveles.

- Número de neuronas a definir hasta 1023*.
- Número de entradas a la red a definir hasta 1023*.
- Número de entradas por neurona a definir.
- Número de sinapsis máximo de 8192*.
- Modelo de neurona de tipo Adaline y función de activación definida por el usuario.
- Memoria para almacenar una función de 16384 valores**
- Interconexiones entre neuronas definidas por el usuario.
- Calcula 1 sinapsis por ciclo de reloj***.
- Actualiza entradas sin detener la ejecución.
- Obtiene cualquier salida si detener la ejecución.
- Reinicia automáticamente el estado de la red.

3.2.2 Funcionamiento

La figura 3.5 muestra las señales de entrada y salida del procesador neuronal.

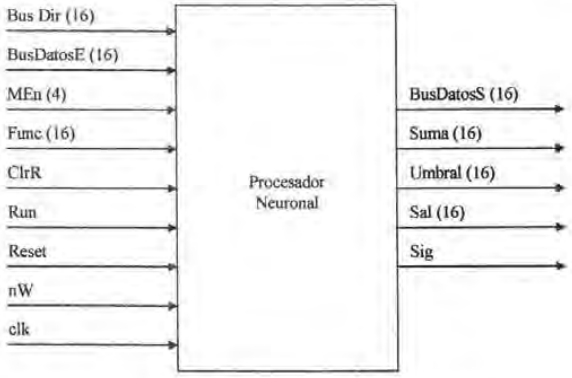


Fig. 3.5. Procesador Neuronal.

* El valor de los parámetros depende de la memoria RAM disponible (interna y externa).
** Si el número máximo de sinapsis se establece en 16384, la memoria de la tabla disminuye a 4096.
*** La frecuencia actual es de 20 MHz.

El número entre paréntesis que se encuentra después de algunas señales indica el número de bits que tiene el bus. Las señales que no cuentan con ninguna especificación son de 1 bit. A continuación se describe cada una de las señales.

- BusDir : Bus de direcciones de 16 bits. Indica la dirección donde se almacenará el dato, por ejemplo, peso o entrada.
- BusDatosE : Bus de datos de entrada de 16 bits.
- MEn : Memoria Enable. Bus de 4 bits que indica la memoria que se desea habilitar para operar sobre ella.
- Func : Función. Dato de entrada de 16 bits que se utiliza como función de activación para operar sobre el resultado.
- ClrR : Borra la red, inicializa los estados actuales de todas las neuronas.
- Run : Señal que indica la ejecución o pausa del proceso.
- Reset : Detiene inmediatamente cualquier operación que esté realizando el procesador.
- nW : Señal que indica si el dato ingresado se lee o se escribe. 1 escritura, 0 lectura.
- clk : Señal de reloj.
- BusDatosS : Bus de datos de 16 bits de salida de la memoria habilitada por MEn.
- Suma : Dato de 16 bits que indica la suma de productos calculada por el procesador para una neurona específica.
- Umbral : Dato de 16 bits. Se utiliza junto con Suma para obtener el dato final que será administrado a través de Func.
- Sal : Dato de 16 bits, indica la salida de la neurona deseada.
- Sig : Señal que indica cada vez que se ha completado el cálculo de la red entera.

Al configurar la red, se especifican los pesos, entradas, conexiones, umbrales si son necesarios, función de activación si es

necesaria, y registros como número de neuronas, número de entradas por neurona (este parámetro se toma como constante para todas las neuronas de la red), función de activación, corrimiento, neurona de salida, etc.

La siguiente tabla muestra las memorias internas del procesador y el valor de MEn para habilitar dicha memoria o registro.

MEn	Memoria o Registro
0	Pesos
1	Entradas de Neuronas
2	Umbrales
3	Entradas a la Red
4	Neurona 1
5	Neurona 2
6	Num. Entradas por Neurona
7	Num. de Entradas a la Red
8	Num. de Neuronas
9	Reg. Selecciona Función
10	Num. de Neurona de Salida
11	Reg. NALU
12	Memoria de Función
13	Reg. Dato 1*
14	Reg. Dato 2*
15	Memoria de Selección de Función
16	Reg. Dato de Función Lineal

Tabla 3.1. Memorias y Registros internos

Cuando se activa MEn con un valor válido de la tabla 3.1, los buses de entrada y salida son conectados a la entrada y salida de la memoria o registro dado, respectivamente. El bus de direcciones siempre es conectado a las memorias internas, así que siempre estará activo para el elemento seleccionado.

Para cargar un valor en una memoria específica solo basta con habilitarla con MEn, poner su dirección si es memoria u omitir el valor del bus de direcciones si se trata de un registro y dar el valor de entrada deseado. Cuando se da el flanco positivo del reloj y la señal nW es 0, el valor es escrito en el elemento de memoria. Si la señal nW es 1, el valor es leído y se encontrará en el bus de salida.

* Estos valores son para la función neuronal.

Se puede cargar un dato en cualquier memoria por ciclo de reloj. Los datos de las memorias de pesos, umbrales, número de entradas, estado de neuronas, selección de función y función sólo pueden ser cargados cuando las señales de Run y Reset se encuentran ambas en 0. Las demás memorias o registros se pueden modificar en cualquier momento. Esto permite que mientras la red se está actualizando se obtengan salidas válidas sin detener la ejecución y al mismo tiempo se puedan actualizar las entradas a la red.

La señal ClrR sólo es válida cuando las señales Run y Reset son ambas 0. Si cualquiera de estas señales está activa, no importa el valor que tenga ClrR, el circuito no dejará de realizar su proceso.

Cuando Run pasa del estado 0 a 1, comienza la ejecución del proceso. Se van obteniendo los valores de las sinapsis y se va calculando los valores de los nuevos estados de las neuronas. Si la señal regresa a 0, el estado actual de las neuronas no es borrado, se conserva, pero todos los cálculos realizados del estado nuevo incompleto son desechados, así que al volver a activar la señal, se calculará de nuevo toda la red. Para borrar el estado actual de la red es necesario desactivar la señal Run y activar la señal ClrR, así el estado actual de la red será inicializado a cero.

3.2.3. Configuración

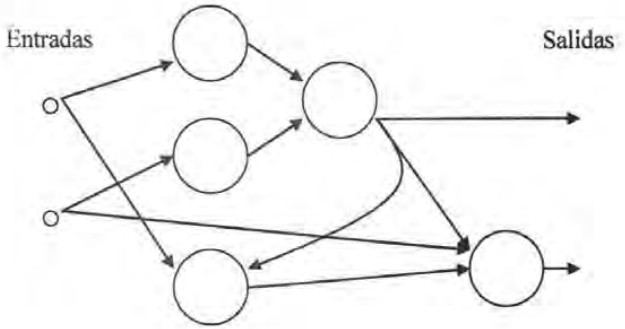
Se deben pasar muchos parámetros al procesador para configurar una red específica. No nada más se define el valor de los pesos y entradas, sino la topología de la red y la función de activación de las neuronas.

Primero se toma en cuenta la topología de la red, las neuronas se enumeran no importando el orden. Se toma la neurona con más sinapsis y éste será el número que se pasará como parámetro al procesador. Como este parámetro es constante para todas las neuronas, si existen neuronas con menos sinapsis, se deberá especificar que las conexiones restantes son nulas. Así, se enumeran todas las sinapsis incluyendo las nulas por toda la red, tomando en cuenta el orden de las neuronas, por ejemplo, para una red como la de la figura 3.6 se pueden enumerar las neuronas y las sinapsis como se muestra.

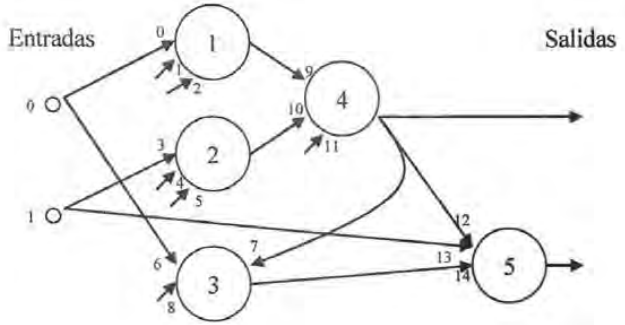
Las neuronas se enumeran comenzando desde el "1", la neurona cero si existe pero su valor es nulo y está reservada, ya que precisamente se utiliza para las conexiones nulas. Las sinapsis se enumeran desde cero. Una vez hecho esto, se almacenan los valores de los pesos, la dirección del peso está dada por la sinapsis correspondiente, para las conexiones nulas, el valor del peso puede ser cualquiera incluyendo 0. Después se procede a almacenar la topología de la red. En realidad estos pasos no tienen una secuencia fija, se pueden almacenar los elementos en cualquier orden incluyendo los registros.

La topología de la red es fácil de describir. Se habilita la memoria de número de entrada (1) y para cada valor de sinapsis se da un número que especifica la neurona o la entrada a la cual está conectado. Si la conexión proviene de una neurona, el valor es el número de neurona a la cual está conectado, si la sinapsis proviene de una entrada, el valor se obtiene del resultado de la suma del número de entrada más el número de neuronas totales más 1. Por ejemplo, para la red de la figura 3.6, se tienen 5 neuronas y las conexiones se especifican como sigue:

Sinapsis	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
valor	6	0	0	7	0	0	6	4	0	1	2	0	4	7	3



a)



b)

Fig. 3.6. Red neuronal a) topología y b) forma de enumerar sus parámetros.

Para la sinapsis 0, como el valor de entrada no proviene de una neurona sino de una entrada a la red, se suma el valor de la entrada, el valor del número de neuronas y se incrementa en 1, 0+5+1=6, así el procesador sabe que se trata de la entrada 0. Las conexiones siguientes son nulas, por lo tanto se ponen a 0. Este valor es importante ya que si se deja cualquier otro valor puede generar resultados incorrectos por una mala interpretación de la topología de la red.

Para la primer conexión de la neurona 2 se tiene la entrada 1, así el valor será 1+5+1=7. Las conexiones de la neurona 4 son muy fáciles de establecer ya que todas provienen de otras neuronas y por lo tanto sólo se pone el número de la neurona de la cual proviene la sinapsis.

Una desventaja de esta arquitectura es que las sinapsis nulas consumen ciclos de instrucción, por lo tanto el procesador pierde tiempo en considerar estas conexiones. El problema se podría arreglar introduciendo una memoria adicional donde se especifica el número de conexiones exacto por neurona, sin embargo, se necesitan de FPGA's más grandes o memorias externas.

Otra desventaja es que como se tiene una arquitectura de tipo Pipeline, cada vez que el procesador realiza el cálculo de la red, se necesitan llenar estos niveles, así que se pierden 5 ciclos cada vez que se actualiza el estado de la red. Para redes con muchas sinapsis este retraso es insignificante y para redes con pocas en realidad no importa por la velocidad del procesador. Para este caso, en vez de que el procesador actualice la red en 9 ciclos, lo realiza en 20 ciclos. Este parámetro depende mucho de la topología de la red, por ejemplo, para una red Hopfield de 10 neuronas, en vez de utilizar 100 ciclos, utiliza 105 ciclos.

Otro aspecto importante es que para redes que utilizan un valor de sesgo (*bias*) en las neuronas se puede introducir poniendo manualmente el valor requerido que es por lo general unitario en la neurona N+1, donde N es el número de neuronas de la red. Por ejemplo, si la red contiene 5 neuronas, el valor se ajusta en la neurona 6 y el procesador no modifica nunca ese valor ni siquiera al limpiar el estado de la red. En la memoria de número de neurona el valor que se pasa no es N+1, ya que este corresponde a la primer entrada o entrada cero de la red, el valor que se le pasa es N+L+1 donde L es el número de entradas a la red. Para el ejemplo en el que se tienen 5 neuronas y 2 entradas, si una neurona requiriera una conexión de sesgo, el parámetro de conexión al que se haría referencia sería 5+2+1=8 y la neurona que se utilizaría sería la 6 con el valor requerido.

Con este tipo de descripción se le puede especificar al procesador cualquier topología de red. Inclusive no importa si se tienen interconectadas más de una red neuronal, simplemente hay que especificar las conexiones entre ellas y las neuronas, y como se verá más adelante, se pueden tener distintos tipos de redes, por ejemplo, se podría tener una red de Retropropagación conectada a un Perceptrón o

a una red Adaline o de Hopfield, la topología depende de la aplicación del usuario. También se puede tener más de una red independiente que se esté ejecutando dentro del mismo ciclo, el procesador lo tomará como si se tratase de una sola red. Así, dependiendo de su tamaño, se pueden obtener resultados en tiempo real de ambas redes (o de todas las redes especificadas).

Los registros son importantes ya que en éstos se encuentran especificados parámetros importantes de la red. Así se debe cargar el registro de número de neuronas, obviamente con el número de neuronas de la red. El registro de número de entradas por neurona, el registro de número de entradas a la red. Con el registro de neurona de salida se puede obtener la salida deseada en el bus de salida (Sal), para el caso del ejemplo anterior, el valor estará fluctuando entre 4 y 5 ya que sólo se puede obtener una salida a la vez.

El registro NALU es muy importante, ya que especifica parámetros que utiliza el bloque de multiplicación y acumulación para realizar las operaciones aritméticas. Como ya se mencionó, la arquitectura utiliza formato de 16 bits en punto fijo, el uso del formato es responsabilidad del usuario. No hay ninguna especificación, se pueden realizar operaciones con o sin signo y utilizando números normalizados o absolutos.

Este registro especifica cómo se deben realizar las operaciones. La figura 3.7 muestra la forma en la que se encuentra implementado el bloque NALU.

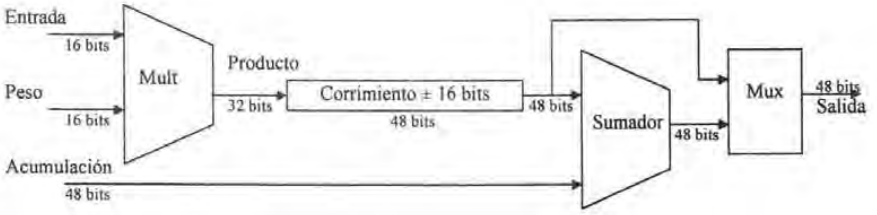


Fig. 3.7. Bloque NALU.

La entrada y peso se multiplican, como ambos datos son de 16 bits el resultado es de 32 bits. La multiplicación se puede realizar de dos maneras, tomando ambos valores como absolutos o tomándolos en complemento a 2. El resultado en ambos casos es diferente, así que el diseñador es quien decide qué tipo de operación utilizará de acuerdo a su conveniencia.

El resultado de 32 bits se pasa a un bloque de registro de corrimiento. Aquí se pueden realizar corrimientos a la izquierda o a la derecha dependiendo del valor almacenado en el registro NALU. Se

pueden realizar corrimientos de 0 a 15 bits en cualquier sentido. Si se realiza el corrimiento a la derecha, los bits menos significativos no se pierden, ya que el registro tiene la capacidad de almacenar 48 bits. Existen dos formas de realizar este corrimiento, tomando en cuenta que el número es signado, entonces si el bit más significativo es 1 todos los bits que se encuentren a la izquierda serán llenados con 1, en caso contrario o si el corrimiento se realiza considerando que el número es no signado, los bits se ponen a 0.

En el caso en que el corrimiento se haga hacia la izquierda, los bits más significativos se perderán y los bits de la derecha se llenarán siempre con 0. Estos corrimientos sirven para realizar ajustes debidos a la multiplicación y al formato que se esté utilizando, ya sea que se consideren números signados o no signados y normalizados o absolutos. La acumulación se realiza en un sumador de 48 bits, esto permite que la precisión no se pierda por los corrimientos al realizar varias sumas de productos.

El registro NALU especifica lo siguiente:

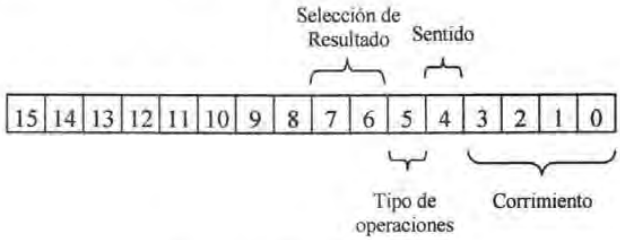


Fig. 3.8. Registro NALU.

Los bits 0-3 indican el corrimiento que se realizará, de 0 a 15. El bit 4 indica el sentido del corrimiento, 0 hacia la derecha o 1 hacia la izquierda. El bit 5 indica si las operaciones son signadas, operaciones con signo 0 y operaciones sin signo 1. Los bits 6 y 7 indican la parte que se tomará como salida final. La suma es de 48 bits pero se puede elegir entre tomar los bits 47 a 32 con "00", los bits 31 a 16 con "01" o los bits 15 a 0 con "10". La combinación "11" está especificada como cero así que no es una entrada válida. El resto de los bits (8 al 15) del registro NALU, hasta este momento no han sido definidos ni utilizados, por lo tanto no importa su valor.

Es muy importante el valor de este registro junto con la función de activación para obtener resultados satisfactorios, de lo contrario el procesador estará realizando de forma incorrecta las operaciones y por lo tanto los valores de las salidas se las neuronas también serán incorrectos.

Finalmente, antes de que se active la señal para que el procesador comience las operaciones, se debe configurar la función de activación de las neuronas. Existen algunas opciones dependiendo del modelo que se esté utilizando.

3.2.4 Función de activación

La arquitectura dispone de varios bloques específicos para la función de activación dentro de uno de los niveles del Pipeline. Se tiene la posibilidad de escoger alguna función ya implementada en el hardware del procesador, de utilizar la memoria de función o de utilizar las señales externas para que el usuario describa la función que necesite con hardware externo, que puede ser algún circuito combinacional, alguna memoria u otra función proveniente de cualquier fuente externa. Existen algunos tipos de redes como el de retropropagación que pueden utilizar más de una función de activación. Para estos casos se dispone de un registro que indica si las neuronas tienen una función en común o si cada neurona tiene una función distinta. Esto es útil cuando se configura el dispositivo para alguna red como Perceptrón, Adaline o cualquier red en la que todas sus neuronas tengan la misma función de activación, donde solamente se tiene que definir una vez y por otra parte, cuando se tienen redes en las que sus neuronas tienen distintas funciones de activación.

Cabe mencionar que si se utilizan solo los recursos del procesador no se puede describir cualquier tipo de red, utilizando hardware externo esto si es posible. Las funciones ya implementadas son escalón y lineal. El bloque encargado de realizar esta operación se muestra en la figura 3.9.

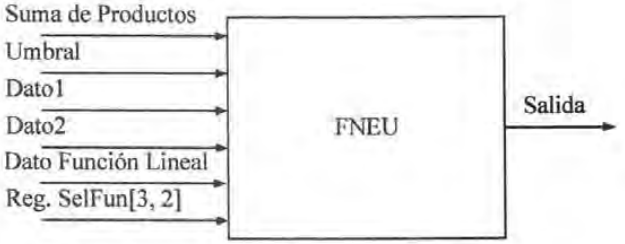


Fig. 3.9. Bloque de función de activación.

El registro encargado de la configuración de la función de activación es el registro de selecciona función. El bit 4 indica si todas las neuronas tienen la misma función de activación (cuando su valor es 0) o si cada una tiene su propia función de activación (cuando su valor es 1). Dependiendo de este valor se seleccionan los parámetros del mismo registro (bits 0 a 3) o de la memoria de selección de función. El

formato que se toma en ambos casos es el mismo, la memoria es de 4 bits, la única diferencia es que ésta contiene la información para cada una de las neuronas, mientras que el registro solo contiene un dato y es tomado para todas las neuronas. Cuando el bit 4 del registro es 0 se toman los parámetros indicados en el mismo registro. Si es 1, los parámetros se toman de la memoria y para este caso, se tiene qué introducir la tabla con los parámetros que se necesite por neurona, a continuación se explica la forma en la que opera el procesador y cómo toma los parámetros correspondientes a la función de activación.

En la figura 3.9 se pueden observar los parámetros que toma el bloque que se encarga de calcular la función de activación. La tabla 3.2 muestra los valores que se obtienen a la salida en función de los bits de configuración.

Control Bits 3 y2 (reg. o mem.)	Salida
00	Dato 1 si SumProd > Umbral Dato 2 si SumProd ≤ Umbral
01	Dato 1 si SumProd ≥ Umbral Dato 2 si SumProd < Umbral
10	Dato 1 si SumProd > Umbral Dato 2 si SumProd < Umbral Salida Anterior si SumProd = Umbral*
11	SumProd*Dato Función Lineal-Umbral**

Tabla 3.2. Funciones de activación.

En la tabla 3.2 se pueden observar las distintas funciones de activación. Las dos primeras son funciones de tipo escalón. Los valores binarios están dados en los registros de Dato 1 y Dato 2, esto es para que el usuario no tenga problemas para decidir el formato numérico a 16 bits que más le convenga. El tercer tipo de función se utiliza en redes como la de Hopfield, el bloque guarda el resultado anterior ya que éste no es modificado si la suma coincide con el valor del umbral.

La función lineal está ideada para que se obtenga el mismo resultado a la salida, es decir, que sea la función identidad. Sin embargo, se pueden tener algunos problemas con el formato numérico debido a ajustes de corrimiento a la salida. En los tipos de funciones anteriores no se tiene este problema debido a que los datos que se obtienen a la salida los define el usuario. Si se necesitan realizar algún ajuste extra para esta función se puede utilizar este dato. En caso contrario es suficiente con ajustar su valor a 1 para que el resultado de

* Esta función de activación la utilizan redes como Hopfield.
** El producto que se realiza es sin signo.

la multiplicación sea el mismo dato de entrada ya que la operación se realiza sin signo y se toma la parte menos significativa, el umbral se ajusta a 0 ya que es restado del producto. Esta operación permite realizar desplazamientos de la recta que define la función.

Puede parecer que debido al formato elegido de punto fijo se tengan muchos parámetros a definir para el funcionamiento correcto del circuito. Sin embargo se puede realizar un programa que genere todos estos parámetros de forma automática ya que sus valores están bien determinados de acuerdo a las operaciones que se requieran realizar y éstos dependen de la red que se quiera configurar. En todos los procesadores de punto fijo, como por ejemplo los DSP's, se tienen qué configurar parámetros para ajustar los resultados de las operaciones.

Finalmente, los bits 1 y 0 del registro o de la memoria de selección de función, indican la fuente de donde proviene el resultado, esto se ilustra en la figura 3.10.

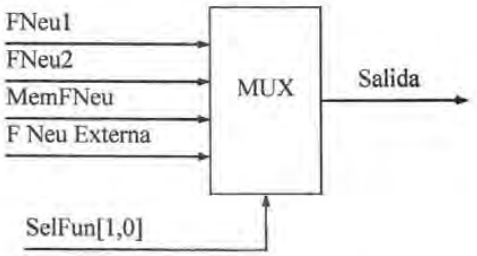


Fig. 3.10. Selección de la función de activación.

Las opciones que se tienen son 4: Bloque de función neuronal FNeu1, cuando el valor es "00"; bloque de función neuronal FNeu2, cuando el valor es "01"; memoria de función, cuando el valor es "10" o función externa, cuando el valor es "11". Los bloques FNeu1 y FNeu2 son prácticamente los mismos, la diferencia está en que el primero realiza las operaciones con números signados y el segundo con números absolutos.

La memoria de función es útil para el caso en que se tengan qué implementar funciones de tipo sigmoidal, tangente hiperbólica, gaussiana o cualquier tipo de función continua o simplemente que sea distinta a las descritas en los bloques de función neuronal. En este caso la idea es introducir la función como una tabla, el resultado de la suma de productos es introducido como la dirección del dato, por lo tanto, a la salida de la memoria se tendrá la función requerida.

En este caso la memoria es insuficiente, lo ideal es que se tuviese una memoria de 64kx16. Con eso se cubrirían todos los valores numéricos. Sin embargo, en la arquitectura final, la memoria es de 4kx16. Lo cual genera una pérdida de 4 bits a la entrada con lo que se

pierde precisión, si se reduce el número máximo de sinapsis a la mitad, se puede configurar la memoria de tal manera que se tengan 16kx16, con lo cual se pierden solo dos bits de precisión. Por desgracia los dispositivos FPGA no son elementos de memoria, por lo tanto, la memoria RAM disponible es muy limitada por lo que se tiene qué recurrir a arquitecturas de tarjetas con memoria externa. Si no se cuenta con memoria externa, las capacidades se ven muy limitadas.

Como nada más se cuenta con una memoria de función, solo se puede definir una función de activación a menos de que se haga uso de los buses externos. Es por esta razón que una red de tipo retropropagación con 2 funciones de activación distintas no se puede configurar en el circuito a menos que una de estas funciones sea lineal, ya que ésta si está definida en un bloque separado. Si se tiene una red cuya capa de entrada es tangente hiperbólica y en la capa oculta se tenga una función sigmoidal, sería imposible configurar el circuito de tal manera que se pueda operar la red sólo con los recursos internos, pero si la red tiene la misma función en todas sus capas o una combinación de alguna de ellas y la función lineal, si se puede configurar para que opere de manera adecuada.

Definiendo todos estos parámetros es como se configura una red neuronal en el procesador. Una vez que se cargaron todos los valores, solo resta activar la señal que inicializa la red (ClrR) y después activar la señal de ejecución (Run). Se pueden manipular las entradas y salidas al mismo tiempo que el procesador actualiza la red.

3.3 Interfaz

Es indispensable el diseño y uso de una interfaz adecuada para aprovechar al máximo las características del procesador. Esto es porque se ha diseñado de tal forma que no es necesario detener la ejecución para obtener las salidas deseadas ni para actualizar el valor de las entradas.

Esto se debe a que la memoria de neurona, que es donde se guarda el estado de cada neurona de la red, es de doble bus y se puede leer en cualquier momento. Así, mientras el procesador opera esta red con un bus, se pueden leer los resultados con el otro bus disponible. Por otro lado, las entradas se manejan de manera especial ya que éstas son almacenadas en una memoria primaria. El procesador no toma los valores directamente de esta memoria ya que está permitido cambiarla en cualquier momento. Si suponemos que dos o más neuronas están conectadas a una cierta entrada y la red toma el valor de esta entrada al evaluar una de estas neuronas y posteriormente el usuario actualiza el valor de la entrada a uno nuevo, cuando el procesador vuelva a necesitar el valor de la entrada, éste ya no será válido debido a que se ha modificado y no se ha terminado de actualizar la red. Para evitar este

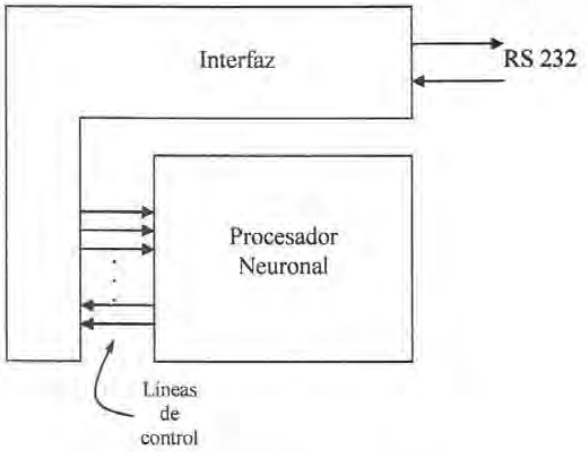
problema, el procesador toma los valores conforme los necesita la red y los almacena en otra memoria interna, así se segura que los valores de entrada no cambien hasta que la red se haya actualizado. Con cada actualización de la red toma los valores de inicio, de esta manera se pueden obtener las salidas o actualizar las entradas sin detener la ejecución y asegurando que los valores calculados sean correctos.

Se puede obtener una salida o actualizar una entrada por ciclo de instrucción pero no se puede obtener una salida y actualizar una entrada al mismo tiempo. Para redes que tengan un número de sinapsis mayor al número de entradas más salidas, se puede realizar todo el procesamiento continuo, sin detener la ejecución, pero para redes con mayor número de entradas-salidas que sinapsis, será necesario detener la ejecución hasta actualizar todos los valores.

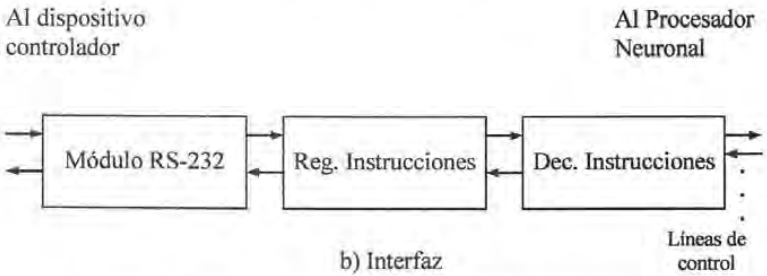
La arquitectura del procesador está completa, se puede controlar totalmente con las señales de entrada y salida pero se recomienda el diseño de una interfaz que permita el uso eficiente del procesador. Esto es para que el se pueda tener una comunicación adecuada con el exterior, ya sea con una PC, con otro sistema digital o directamente con sensores y actuadotes dependiendo del tipo de aplicación.

La comunicación con una computadora es importante ya que servirá de conducto para configurar las distintas redes. Una interfaz simple pensada para este propósito se muestra en la figura 3.11.

En realidad una interfaz de este tipo sirve para comunicar el procesador con cualquier otro dispositivo o sistema que tenga un puerto RS232. Básicamente se trata de enviar un parámetro por el puerto, la interfaz se encargará de decodificarlo para poner todas las señales y buses a sus estados correspondientes. Este parámetro se puede interpretar como una instrucción, dependiendo de su valor, el módulo espera otros datos u opera directamente sobre el procesador. El módulo implementado consta de tres bloques, uno se encarga de enviar o recibir datos a través del puerto serie RS-232. Este es un módulo genérico y no necesita de ninguna función especial, es por esto que el módulo utilizado es el mismo que se proporcionó por el fabricante de la tarjeta como código de ejemplo. Los siguientes dos módulos actúan en conjunto estrecho y utilizan el primero para enviar datos de respuesta. El registro de instrucciones se encarga de catalogar los datos recibidos, dependiendo de la instrucción que se trate llega a esperar uno o dos datos más. Finalmente, pasa un parámetro al decodificador el cual se encarga de llevar todas las líneas de control a sus respectivos estados. Por ejemplo, si se requiere escribir un dato, el registro de instrucciones espera tres bytes que definen el comando (la instrucción y el dato de 16 bits), estos parámetros son pasados al decodificador, el cual se encargará de poner el dato en el bus de datos y activar la señal de escritura en un ciclo de instrucción para que el dato sea almacenado.



a) Forma de operar el procesador
con una interfaz Serie.



b) Interfaz

Fig. 3.11. Interfaz básica para la comunicación con una PC.

En el apéndice B se muestra la forma de utilizar esta interfaz y las instrucciones que soporta. También se muestra un ejemplo de la configuración de una red neuronal y el código correspondiente.

3.4 Implementación

La arquitectura se programó completamente en VHDL, así que los módulos y la arquitectura completa en general, pueden ser transportados o programados en dispositivos lo suficientemente grandes como para soportar el diseño completo. En este caso se utilizó una tarjeta de evaluación con un FPGA de la compañía Xilinx de la familia Virtex II modelo XC2V1000FG256-4. Una característica importante del dispositivo es que cuenta con 1000 000 de compuertas programables.

Finalmente, cabe señalar que se utilizó el ambiente ISE versión 6.3 (fig. 3.12) provisto por la misma compañía Xilinx que soporta casi todas las familias de dispositivos programables que ofrece. Para aprender a manejar la interfaz se pueden consultar los documentos provistos por la empresa como ISE 6 In-Depth Tutorial o ISE Quick Start Tutorial. La interfaz en la PC, como se verá más adelante, está programada en un ambiente visual utilizando C++.

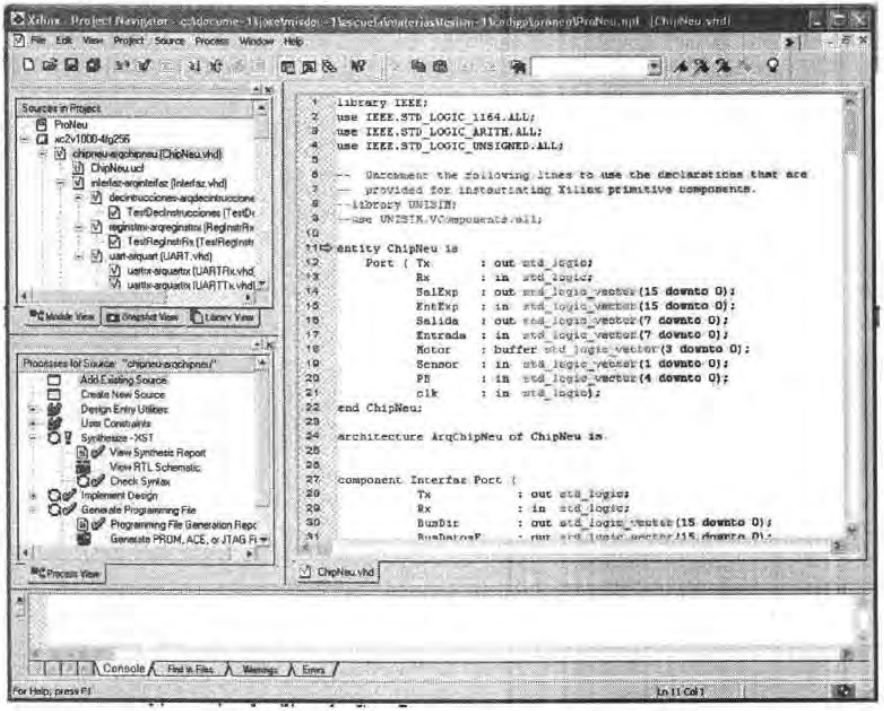


Fig. 3.12. Entorno de desarrollo ISE.

En el apéndice C se muestra el diagrama completo de la arquitectura donde pueden distinguirse todos los módulos antes mencionados, que en conjunto, logran que se ejecute una sinapsis por ciclo de instrucción sin importar el tipo de red que se trate y permiten el ajuste de entradas y la obtención de salidas sin que se detenga la ejecución.

CAPÍTULO 4

Resultados y conclusiones

En este capítulo se presentan algunas pruebas hechas con el circuito y algunas comparaciones de tiempos de ejecución con una computadora personal. Se describe una aplicación utilizando un sistema autónomo, en este caso un robot móvil donde se ilustra la potencialidad de las redes neuronales. Finalmente se mencionan algunas mejoras de fácil implementación para optimizar el desempeño del procesador.

4.1 Prueba del funcionamiento del procesador neuronal

El funcionamiento del procesador se ha probado configurando una red de tipo Hopfield, para reconocimiento de imágenes. La figura 4.1 muestra la topología de la red.

La red se entrenó en matlab para que pudieran ser reconocidos los patrones de entrada que se muestran en la figura 4.2. Los parámetros de esta red dependen de los patrones que se quieran reconocer, ya que el tamaño de éstos dicta el número de neuronas que se necesitan.

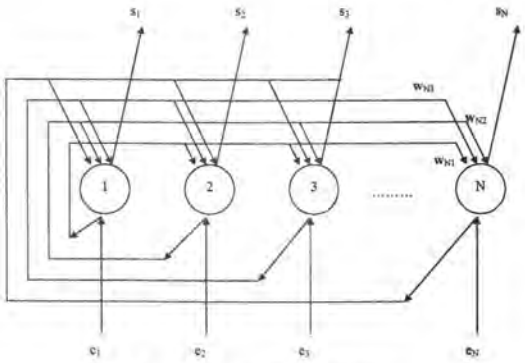


Fig. 4.1. Red Hopfield.

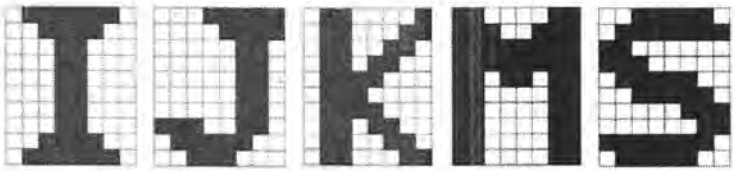


Fig. 4.2. Patrones de entrenamiento de la red.

La red consta de 80 neuronas, por lo que solo se pueden reconocer 4.5 patrones distintos con exactitud, pero se pueden reconocer hasta 11 de manera satisfactoria. Primero se verifica el nivel de similitud de los patrones con la ecuación:

$$NS = \sum_{i=1}^{N} e_i^{(k)} e_i^{(m)} \leq 0$$

donde: $e_i^{(k)}$ y $e_i^{(m)}$ son los valores binarios (+1, -1) de los *i*-ésimos componentes de los patrones de entrada distintos.

Los resultados obtenidos son los siguientes:

Entre *i* y *j* *NS* = 4

Entre *i* y *k* *NS* = 0

Entre *i* y *m* *NS* = -40

Entre *i* y *s* *NS* = 16

Entre *j* y *k* *NS* = -8

Entre *j* y *m* *NS* = -24

Entre *j* y *s* *NS* = 16

Entre *k* y *m* *NS* = 0

Entre *k* y *s* *NS* = -4

Entre *m* y *s* *NS* = -20

Como puede observarse, existen pares de entradas que no cumplen con la restricción de ortogonalidad, con lo cual no se garantiza que la red realice una asociación correcta, sin embargo, puede observarse que en estos casos el resultado es bajo, teniéndose un nivel de similitud máximo de 60%.

Para esta prueba se programó una aplicación que genera distintas entradas con un nivel de ruido que puede ser ajustado. La figura 4.3 muestra la interfaz de la aplicación y los principales controles.

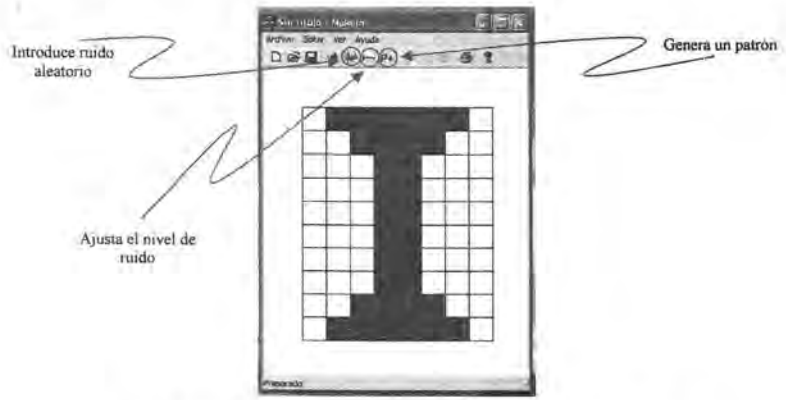


Fig. 4.3. Aplicación que genera patrones de entrada.

Esta aplicación también muestra el resultado obtenido una vez que se le presentó la entrada a la red. La aplicación no interactúa directamente con la tarjeta sino que genera y lee código que puede ser manejado por la interfaz que se explica en el apéndice B.

La figura 4.4 muestra las entradas aplicadas a la red con sus salidas correspondientes. Estas entradas se generaron a partir de los patrones originales añadiendo 10% de ruido.

Como ya se mencionó, la red consta de 80 neuronas y tiene 6400 sinapsis. La función de activación es escalón. Con una frecuencia de 20MHz la tarjeta procesa 20 millones de sinapsis por segundo, lo que

significa que la red es refrescada aproximadamente cada 320 µs. Las figuras se eligieron de 8x10 debido a las limitaciones de memoria del procesador.

Fig. 4.4. Entradas presentadas a la red y su salida correspondiente.

En la figura 4.5 se muestra una tabla con entradas y salidas de la red aumentando el nivel del ruido para cada patrón desde 15% hasta 35%. Se incluyen las salidas para cada caso ya que para algunas entradas no se obtuvo la salida esperada, siendo que ni siquiera se obtuvo un patrón parecido a los que se utilizaron para entrenar la red. Tal es el caso para las entradas *i* con 15% y 25% de nivel de ruido, las entradas *j* y *k* con 35% de ruido y las entradas *s* con 25% y 35% de ruido. Para el caso de la entrada *m* se tiene que en ningún caso se

obtuvo el patrón original, sin embargo, se obtuvo un patrón suficientemente parecido al original de tal forma que se puede establecer que se trata de éste. Esta cuestión se debe a las limitaciones de la red Hopfield que se explican en el capítulo 1.

Fig. 4.5. Entradas con distintos niveles de ruido y sus salidas.

Como puede observarse, la arquitectura trabaja de forma correcta. Esta prueba sirvió para comprobar el funcionamiento de la arquitectura a la frecuencia establecida de 20 MHz. La funcionalidad completa de los módulos así como las operaciones aritméticas se comprobaron simulando el comportamiento del circuito, utilizando diagramas de tiempos y verificando el valor real de los buses de datos internos en el FPGA.

4.2 Comparación de la tarjeta con otros dispositivos

El funcionamiento de esta arquitectura no se puede comparar con otros trabajos realizados utilizando PLD's y RNA como los citados en las referencias [13] a [18], debido a que fueron creados con distintos fines.

Tendrían qué tomarse en cuenta muchos parámetros como velocidad, funcionalidad, hardware utilizado, costo, facilidad de manejo, consumo de energía, tamaño, alcance, etc., y la comparación podría ser subjetiva ya que depende de la aplicación final el tipo de software y hardware a utilizar. Por esta razón la principal comparación se realiza con respecto a una computadora personal y a un microcontrolador PIC principalmente de la serie 18 que es la que cuenta con un multiplicador por hardware.

Hay que tomar en cuenta distintos parámetros, en el apéndice E se incluyen las especificaciones de la tarjeta utilizada que contiene un FPGA de la familia Virtex II de la compañía Xilinx. Hay que recalcar que el dispositivo incluido en la tarjeta de evaluación es el más lento de los disponibles (grado de velocidad de -4) y el que menos pines de E/S contiene.

Para comparar la velocidad de ejecución se realizaron unos programas en la computadora utilizando lenguaje de programación C/C++ y el compilador de Visual Studio 6. El programa fue ejecutado bajo ambiente Windows versión XP Pro SP 1.

La computadora cuenta con las siguientes características principales: Procesador Athlon XP de AMD modelo 2100 a 1.80 GHz, 1 GB de memoria RAM DDR a 333 MHz, tarjeta de video de 128 MB independiente (el procesador no se encarga de la salida de video ni sonido).

Una característica importante a tomar en cuenta es que no se puede programar exactamente el algoritmo que ejecuta el dispositivo FPGA debido a que en éste se pueden implementar rutinas o módulos en paralelo mientras que en una computadora no. Así que la primera prueba fue programar una red Hopfield de 120 neuronas con función de activación escalón. El ciclo principal programado es el siguiente (en pseudo código):

```
--Inicialización de variables, entradas y pesos
--...
--Ciclo principal
ContExt=0
Ciclo_Externo: Si ContExt = 120 SALTA Fin
Suma=0
ContInt=0
Ciclo_Interno: SI ContInt = 120 SALTA Fin_Ciclo_Int
SI ContInt=ContExt
Suma = Suma+Entrada[ContInt]*Pesos[120*ContExt+ContInt]
SI ContInt = ContExt
Suma = Suma+ Neurona[ContInt]*Pesos[120*ContExt+ContInt]
ContInt = ContInt+1
SALTA Ciclo_Interno
Fin_Ciclo_Int: SI Suma = Umbrales[ContExt] Salida = 1;
SI Suma < Umbrales[ContExt] Salida = -1
```

```
Si Suma = Umbrales[ContExt] Salida = Neurona[ContExt]
NeuronaNueva[ContExt] = Salida
ContExt = ContExt + 1
SALTA Ciclo_Externo
Fin:
```

Este código tan solo se encarga de la ejecución de la Red, mientras que la tarjeta puede actualizar las entradas y obtener salidas al mismo tiempo. Para una red como la especificada, la tarjeta ocupa 14405 ciclos, eso significa que se tarda 720.25 µs, mientras que la computadora se tarda aproximadamente 66 µs. Esto significa que el algoritmo en la PC es 10.9 veces más rápido que la tarjeta, sin embargo, hay que tomar en cuenta que el ciclo del procesador de la PC es de 1.8 GHz comparado con 20 MHz de la tarjeta, es decir, 90 veces más rápido.

También se programó un algoritmo para una red de tipo retropropagación con 3 capas y 5 neuronas por capa utilizando función tangente hiperbólica en todas las neuronas. El código principal (en pseudo código) es el siguiente:

```
--Inicialización de variables, entradas y pesos
--
--Ciclo principal
ContCapas=0
Ciclo_Capas: SI ContCapas = 3 SALTA Fin
ContNeuronas=0
Ciclo_Neuronas: SI ContNeuronas = 5 SALTA Fin_Ciclo_Neuronas
Suma=0
ContEntradas=0
Ciclo_Entradas: SI ContEntradas = 5 SALTA Fin_Ciclo_Entradas
SI ContCapas=0
Suma = Suma+Entradas[ContEntradas]*Pesos[5*ContNeuronas+ContEntradas]
SI ContCapas=0
Suma = Suma+Neuronas[ContEntradas+(5*(ContCapas-
1))]*Pesos[3*ContCapas+5*ContNeuronas+ContEntradas]
ContEntradas = ContEntradas + 1
SALTA Ciclo_Entradas
Fin_Ciclo_Entradas:NeuronaNueva[5*ContCapas+ContNeuronas] = tanh(Suma)
ContNeuronas = ContNeuronas+1
SALTA Ciclo_Neuronas
Fin_Ciclo_Entradas: ContCapas=ContCapas+1
SALTA Ciclo_Capas
Fin:
```

Como puede observarse, este código nada más ejecuta las operaciones sin tomar en cuenta las entradas y salidas. Para un caso

como este se tienen 75 sinapsis por lo que la tarjeta actualiza la red en 4 µs, mientras que la PC se tarda 1.6 µs, lo que significa que para una red de tipo retropropagación tan solo es 2.5 veces más rápida.

En el caso del PIC, se hizo la rutina en ensamblador que calcula una red de retropropagación como la descrita, el código para un PIC con multiplicador por hardware es el siguiente:

```
--Inicialización de variables, entradas y pesos
--...
--Ciclo principal
                    CLRF    ContCapas
CicloCapas:         MOVF    ContCapas, W
                    SUBLW   3
                    BZ      Fin
                    CLRF    ContNeur
CicloNeuronas:      MOVF    ContNeur, W
                    SUBLW   5
                    BZ      FinCicloNeu
                    LFSR    FSR1, Entradas
                    CLRF    Suma
                    CLRF    ContEntradas
CicloEntradas:      MOVF    ContEntradas, W
                    SUBWL   5
                    BZ      FinCicloEntradas
                    MOVF    POSTINC0, W
                    MULWF   POSTINC1
                    BTFSC   INDF1, SB
                    SUBWF   PRODH, PRODH
                    MOVF    INDF1, W
                    BTFSC   INDF0, SB
                    SUBWF   PRODH, PRODH
                    RLNCF   PRODH, W
                    ADDWF   Suma, Suma
                    MOVF    ContEntradas, W
                    ADDLW   1
                    MOVWF   ContEntradas
                    BRA     CicloEntradas
FinCicloEntradas:
                    MOVFF   Suma, PSFunc
                    MOVFF   PEFunc, Suma
                    MOVF    Suma, POSTINC3
                    MOVF    ContNeuronas, W
                    ADDLW   1
                    MOVWF   ContNeuronas
                    BRA     CicloNeuronas
FinCicloNeuronas:
                    MOVF    Entradas, W
                    ADDWL   5
                    MOVWF   Entradas
                    MOVF    ContCapas, W
                    ADDWL   1
                    MOVWF   ContCapas
                    BRA     CicloCapas
Fin:
```

Esta rutina toma aproximadamente 1585 ciclos de instrucción para ejecutarse, tomando en cuenta que solo se tiene un desplazamiento a la salida del multiplicador antes de realizar la acumulación. Utilizando la

frecuencia máxima para un PIC que es de 40 MHz, el ciclo toma 158.5 µs, que es 39.625 veces más lento que la arquitectura.

Si se utiliza un PIC que no cuenta con multiplicador por hardware, como por ejemplo la serie 16, la rutina es similar, sólo que en el ciclo principal se realiza la multiplicación por software. Los tiempos se pueden observar en la tabla 4.2. Así la rutina se realizaría en 7960 ciclos de instrucción, lo que equivaldría a 796 µs, que es 199 veces más lento que la tarjeta. La siguiente tabla muestra los resultados:

Red	PC (1.8 GHz)	PIC 18/16 (40 MHZ)	Arquitectura (20 MHz)
Hopfield	66 µs	31.944/154.344 ms	720.25 µs
Retropropagación	1.6 µs	158.5/796 µs	4 µs

Tabla 4.1. Comparación de tiempos de ejecución de la tarjeta contra una PC.

TABLE 8-1: PERFORMANCE COMPARISON

Routine	Multiply Method	Program Memory (Words)	Cycles (Max)	Time		
				@ 40 MHz	@ 10 MHz	@ 4 MHz
8 x 8 unsigned	Without hardware multiply	13	69	6.9 µs	27.6 µs	69 µs
	Hardware multiply	1	1	100 ns	400 ns	1 µs
8 x 8 signed	Without hardware multiply	33	91	9.1 µs	36.4 µs	91 µs
	Hardware multiply	6	6	600 ns	2.4 µs	6 µs
16 x 16 unsigned	Without hardware multiply	21	242	24.2 µs	96.8 µs	242 µs
	Hardware multiply	28	28	2.8 µs	11.2 µs	28 µs
16 x 16 signed	Without hardware multiply	52	254	25.4 µs	102.6 µs	254 µs
	Hardware multiply	35	40	4.0 µs	16.0 µs	40 µs

Tabla 4.2. Tiempos en los que se realiza la multiplicación en los PIC [22].

Cabe señalar que los algoritmos programados tan solo se dedican a actualizar los valores de los nuevos estados de las neuronas, como ya se mencionó, mientras que la arquitectura se encarga de las entradas y salidas sin detener la ejecución. Por otra parte los algoritmos son diferentes y su tiempo de ejecución depende de la red como se puede observar en la tabla, mientras que la arquitectura es independiente de la topología de la red y del tipo de neuronas que utiliza.

Otro aspecto a tomar en cuenta es el tamaño, peso, consumo y costo de ambos dispositivos. Una PC quizá no se pueda implementar en una solución donde se necesite algún sistema autónomo. Lo más importante, es que, como se verá más adelante, se puede mejorar el desempeño de la arquitectura. Un primer paso sencillo de implementar

es aumentar tan solo un nivel más de pipeline, con esto la velocidad de ejecución se podrá aumentar a 40 MHz con lo cual se tiene que para una red Hopfield, la PC es tan solo 5 veces más rápida y para una red de tipo retropropagación casi se llega a igualar el desempeño en cuanto a velocidad (la computadora sería 1.25 veces más rápida que el procesador neuronal).

4.3 Conclusiones y trabajo a futuro

A pesar de que la arquitectura ofrecida es secuencial, cuenta con un alto grado de paralelismo como se mostró en la sección de diseño y en los resultados obtenidos. Realizando algunas mejoras y modificaciones, se puede competir con sistemas tan rápidos como las PC. Por ejemplo, sin duda alguna la operación más lenta que existe en todo el proceso es el producto de dos números. Un producto simple en este dispositivo se realiza en aproximadamente 15 ns, considerando que se realiza en un solo paso. Así, ese sería el mayor retardo que se tendría, con lo cual se podrían ejecutar las operaciones a una frecuencia de 66 MHz. Al utilizar esta frecuencia, se logra una velocidad de ejecución mayor que una PC actual para ciertos tipos de redes.

Realizando el producto por pasos, con más de un nivel de pipeline, se puede ejecutar el algoritmo a frecuencias más altas, recordando que el dispositivo utilizado es el más lento de su familia, así, al configurar un dispositivo semejante pero más rápido, se podrá incrementar la frecuencia sin tener que realizar modificaciones a la arquitectura.

Como puede observarse, se tienen grandes ventajas al utilizar dispositivos FPGA. La primera es el tamaño y peso, la tarjeta de evaluación mide 7.5 x 14 cm y pesa algunos gramos. Como se vio en la aplicación, se puede incorporar a sistemas autónomos y de dimensiones similares y a cualquier otro dispositivo más grande. Esta tarjeta no tiene memoria extra pero existen otras en el mercado con dispositivos similares (o incluso el mismo) con memoria DDR y aunque el tamaño es más grande, dependiendo de las características, no doblan las dimensiones antes mencionadas. Por otra parte, un CPU tiene dimensiones mayores y pesa varios kg.

Una ventaja adicional es el consumo de energía. El fabricante recomienda un suministro de 5 Vdc a 1 A con lo cual se tiene un máximo consumo de 5 W, mientras que una computadora como la descrita (contra la que se comparó el desempeño) consume alrededor de 350 W sin considerar el dispositivo de salida de video.

Las principales características son las siguientes:

- Dispositivo portable, el cual se puede incorporar a sistemas autónomos por su tamaño, peso y consumo.
- Autónomo, solo se necesita de un dispositivo externo para configurar la red que se requiera ejecutar, pero se puede utilizar en una aplicación sin que ningún otro sistema lo controle (como en robots móviles).
- Bajo consumo de energía.
- Flexible, se pueden configurar distintas topologías de red, no fue creado para una especifica y el desempeño es independiente del tipo de red y de las funciones de activación de las neuronas.
- Procesa grandes cantidades de datos. Está diseñado para procesar miles de neuronas y sinapsis dependiendo de la memoria disponible.
- Expandible, se puede incrementar súbitamente el número de sinapsis máximo dependiendo de la memoria externa que se le añada, pudiendo ser de hasta millones de conexiones.
- Velocidad, el desempeño se puede comparar con el de una PC e incluso puede ser superado realizando algunas modificaciones a la arquitectura.
- Reconfigurable, al ser implementado en un dispositivo lógico programable, se pueden realizar cambios a la arquitectura con el fin de mejorar el desempeño, aumentar la funcionalidad y la velocidad sin tener que cambiar de dispositivo.

Estas caracteristicas son las que hacen al sistema descrito útil para distintas aplicaciones. Se pueden realizar muchas mejoras importantes, la primera es la adición de memoria externa que permita configurar redes o conjuntos de redes de miles de neuronas y millones de conexiones, a diferencia de diseños donde el número de neuronas es limitado por la arquitectura o inclusive se tienen topologías fijas e invariantes. Al permitir tener un gran número de neuronas y sinapsis se pueden realizar tareas más complejas y completas.

Uno de los aspectos más importantes es que el diseño permite configurar redes con topologías distintas e incluso, es posible configurar internamente conjuntos de redes que pueden ser distintos, ya sea que estén interactuando entre sí o sean independientes. El desempeño no está en función de la configuración de la red, a diferencia de un programa en el que la ejecución si depende de este parámetro.

Otra mejora importante es el cambio de formato numérico interno, se puede pasar de punto fijo a punto flotante, sin embargo, esta

característica arrojaría cambios importantes en la arquitectura aumentando por mucho los niveles de pipeline para lograr realizar un producto en punto flotante en un ciclo de instrucción y de forma eficiente. Sin embargo, con esta modificación, ya no tendrían qué configurarse tantos parámetros debidos al punto fijo, las operaciones siempre se realizarían de forma correcta. Aunado a esto, al mismo tiempo se podría aumentar la precisión numérica aumentando el tamaño de los datos utilizados por el procesador, en vez de ser una arquitectura a 16 bits, podría aumentarse a 32, o incluso, 64 bits teniendo una gran precisión al realizar las operaciones aritméticas.

Para lograr el objetivo de procesar una sinapsis por ciclo de instrucción teniendo más de 16 bits y utilizando punto flotante, no nada más es necesario aumentar los niveles de pipeline en el bloque NALU, sino en los bloques de función de activación neuronal. Al tener 32 o 64 bits de precisión, se requerirían de tablas muy grandes para almacenar los datos de la función teniendo más desventajas que ventajas.

Para resolver este problema se podrían tener bloques de aproximación de funciones, donde la función de activación se divide en intervalos y se utilizan funciones lineales para obtener el valor deseado. Mientras más divisiones se tengan, más preciso será el cálculo de la función requerida.

Como ya se mencionó, una parte importante que no pertenece a la arquitectura pero ayuda a facilitar su manejo, es el desarrollo de una interfaz adecuada. Se ha pensado en el desarrollo de un procesador que funcione en forma paralela, con el cual se puedan manejar los puertos de entrada y salida ya que la forma en la que se introducen y obtienen los datos en el procesador neuronal es serial. Esto se diseñó así debido a que siempre se tiene un número limitado de pines de E/S en cualquier dispositivo.

Todas estas modificaciones requieren de tiempo de desarrollo por lo que se han dejado como trabajo a futuro. Lo importante es mencionar que los objetivos planteados al principio del proyecto fueron alcanzados, se logró diseñar un procesador neuronal que si bien no es la solución más rápida para todos los tipos de redes, si presenta grandes ventajas en otros aspectos, como la flexibilidad de configurar cualquier tipo de red, la capacidad para procesar redes de gran tamaño, elegir distintos tipos de funciones de activación para las neuronas, etc., que son aspectos que no son tomados en cuenta por otro tipo de arquitecturas. Al probar el procesador se vio la gran potencialidad que se puede alcanzar siguiendo esta misma idea de diseño.

APÉNDICE A

Palabras reservadas, tipos de datos y operadores de VHDL

Las palabras reservadas por VHDL son:

abs	else	nand	return
access	elsif	new	select
after	end	next	severity
alias	entity	nor	signal
all	exit	not	subtype
and	file	null	then
architecture	for	of	to
array	function	on	transoprt
asser	generate	open	type
attribute	generic	or	units
begin	guarded	others	until
block	if	out	use
body	in	package	variable
buffer	inout	port	wait
bus	is	procedure	when
case	label	process	while
component	library	range	with
configuration	linkage	record	xor

constant	loop	register
disconnect	map	rem
downto	mod	report

VHDL posee un número limitado de tipos de datos con los que se puede diseñar:

- BIT, {0,1}
- BOOLEAN, {false, true}
- CHARACTER, {la tabla ASCII desde 0 a 127}
- INTEGER, {-2147483647, 2147483647}
- NATURAL, {0, 2147483647}
- POSITIVE, {1, 2147483647}
- STRING, array {POSITIVE range <>} de CHARACTER
- BIT_VECTOR, array {NATURAL range <>}de BIT
- Tipos físicos, como TIME
- REAL, {-1E38, 1E28}
- POINTER, para accesos indirectos a datos.
- FILE, para accesos a ficheros y pilas de datos del disco duro.

Muchos de estos tipos de datos corresponden a estructuras del lenguaje no sintetizables. De hecho, durante la síntesis de bloques de VHDL, casi nunca se utilizan de manera explícita y completa estos tipos de datos. Los tipos de datos más utilizados se acercan más al funcionamiento normal de un circuito, se trata del paquete *std_logic_1164* de la biblioteca IEEE.

En esta biblioteca se define un nuevo tipo de dato enumerado denominado *std_logic*, que contempla los siguientes valores:

- 'U', no inicializado, es el valor por defecto.
- 'X', desconocido. Es el valor que toma en casos de conflicto.

- '0', el valor lógico bajo.
- '1', el valor lógico alto.
- 'Z', representa el estado de alta impedancia.
- 'W', estado desconocido débil.
- 'L', valor lógico bajo débil.
- 'H', valor lógico alto débil.
- '-', valor que no importa

Las situaciones tipo débil suelen representar situaciones de tipo resistivo. Por ejemplo, una señal con valor 'L' se conecta directamente a una con valor '1' el resultado será una señal de valor '1'.

De la misma manera están definidos los tipos VECTOR *std_logic_vector*, para soportar conjuntos ordenados de del tipo *std_logic*. Dentro del paquete también se encuentran definidas las operaciones boleanas y los comparadores = y /=.

VHDL admite la definición de nuevos tipos de datos de una forma muy simple. Dentro de la parte declarativa es posible escribir de manera explícita todos los posibles valores que puede tomar una señal de un determinado tipo. Se le conoce como tipo enumerado:

TYPE nombre_tipo_dato IS (Valor1, ..., ValorN);

Una matriz se define:

TYPE matriz_nombre_tipo IS ARRAY (intervalo de enteros) OF nombre_tipo;

Atributos

Los atributos son valores asociados a una señal o variable que proporcionan información adicional, independientemente del valor que la señal o variable tenga en un instante dado. Son de gran utilidad en la elaboración de código que depende intensivamente de parámetros. Algunos atributos definidos en VHDL son:

- LEFT. Identifica el valor definido a la izquierda del intervalo de índices.
- RIGTH. Identifica el valor definido a la derecha del intervalo de índices.
- HIGH. Identifica el valor máximo del intervalo de índices.
- LOW. Identifica el valor mínimo del intervalo de índices.

- LENGTH. Identifica el valor total del intervalo de índices.
- RANGE. Copia el intervalo de índices.
- REVERSE_RANGE. Representa el mismo intervalo de índices pero en sentido inverso.
- POS(x). Posición de x dentro del dato.
- PRED(x). Elemento que está delante de x dentro del dato.
- SUCC(x). Elemento que está detrás de x dentro del dato.
- EVENT. Indica si se ha producido algún cambio en la señal.
- STABLE(t). Indica si la señal estuvo estable durante el último periodo t.

Operadores y expresiones

Las expresiones en VHDL son muy similares a las expresiones de otros lenguajes de programación. Una expresión funcional representa una función bolenana o aritmética que es una combinación de señales. Dentro de las expresiones se incluyen operadores de distintos tipos:

Operadores Aritméticos.

- ** Exponencial. Eleva un número a un a potencia, X**Y es X^Y. El operador de la izquierda puede ser entero o real, mientras que el de la derecha sólo puede ser entero.
- ABS Valor absoluto. Devuelve el valor absoluto del parámetro.
- * Multiplicación.
- / División.
- MOD Módulo. Calcula el módulo de dos números. Los operadores solo pueden ser enteros.
- REM Resto. Calcula el resto de la división entera.
- + Suma.
- - Resta.

Operadores de desplazamiento.

- SLL Desplazamiento lógico a la izquierda. El desplazamiento se realiza llenando con ceros los huecos.
- SRL Desplazamiento lógico a la derecha.
- SLA Desplazamiento aritmético a la izquierda.
- SRA Desplazamiento aritmético a la derecha.
- ROL Rotación a la izquierda.
- ROR Rotación a la derecha.

Operadores relacionales.

- = Igual
- /= No igual.
- < Menor que.
- > Mayor que.
- <= Menor igual.
- >= Mayor igual.

Operadores lógicos.

- AND
- OR
- NOT
- NAND
- NOR
- XOR

Operadores varios.

- & Concatenación. Concatena matrices o vectores.
- <= Asignación. Solamente para asignar valores a las señales.
- := Asignación. Solamente para asignar valores a las variables.

Precedencia de operadores.

**	ABS	NOT				Máxima precedencia
*	/	MOD	REM			
+(signo)		-(signo)				
+	-	&				
=	/=	<	<=	>	>=	
AND	OR	NAND	NOR	XOR		Mínima precedencia

APÉNDICE B

Interfaz y operación

La interfaz que permite la comunicación entre la tarjeta y una computadora consta de dos partes. La correspondiente a la tarjeta que se encarga de recibir datos por el puerto serie RS232 y la parte correspondiente a la PC (o cualquier otro dispositivo) que se encarga de enviar las instrucciones.

Como ya se mencionó en el capítulo 3, se necesita de una interfaz adecuada que permita el máximo aprovechamiento de la arquitectura o simplemente que permita su fácil uso. En este caso, se programó una interfaz simple (fig. 3.11) que recibe un dato del puerto serie y lo pasa a un decodificador, el cual se encarga de manejar adecuadamente las señales de control y los buses de E/S del procesador. El decodificador también decide si el siguiente dato será tomado como instrucción o como parámetro.

Así se define un protocolo sencillo con 10 instrucciones que son suficientes para controlar todas las características del procesador desde cualquier dispositivo con puerto serie programable. La tabla B.1 muestra las instrucciones definidas y su función.

Instrucción	Descripción
RUN0	Pone la entrada Run a '0'
RUN1	Pone la entrada Run a '1'
MEN *dato*	Escribe *dato* en el bus MEn
DIR *dato*	Escribe *dato* en el bus de direcciones

Tabla B.1. Instrucciones de la interfaz del procesador.

WDT *dato*	Escribe *dato* en el bus de datos de entrada y da un pulso a la señal nW
RDT	Lee el bus de datos de salida
ROUT	Lee el bus de salida
CLRR	Da un pulso a la señal ClrR
RSTP	Activa Run hasta que cambia la señal Sig.
NOP	No operación

Tabla B.1 (continuación). Instrucciones de la interfaz del procesador.

Cada instrucción tiene un valor binario asociado que es el que acepta la interfaz del procesador. Pero en este caso se tiene un programa en la PC que acepta directamente la entrada de estas instrucciones. Esta aplicación se muestra en la figura B.1.

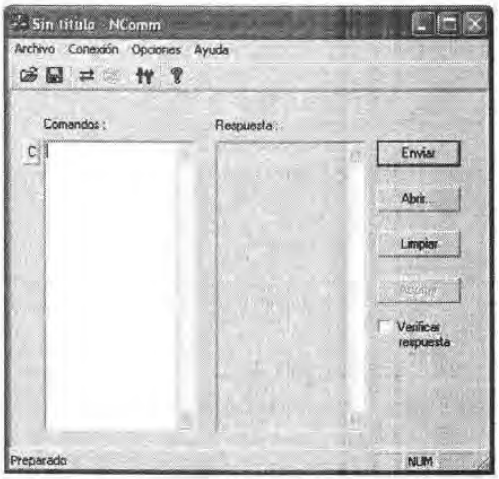


Fig. B.1. Interfaz de usuario.

Fig. B.2. Controles de la interfaz de usuario.

La interfaz permite manejar todas las señales de E/S con las instrucciones que pueden provenir de un archivo de texto plano. Así se puede hacer un programa que configure redes específicas, traduciendo la topología de la red y los datos de pesos y entradas a instrucciones que se pueden enviar al procesador.

Aunque esta interfaz es básica, se puede configurar por completo una red. La sintaxis que acepta es la siguiente:

Instrucción [*#parámetro*]

En el campo de *Instrucción* se puede escribir cualquier palabra de la tabla B.1. El campo de *parámetro* es opcional ya que no todas las instrucciones requieren parámetros. Se pueden escribir números en formato decimal o hexadecimal, para este último es necesario añadir un signo "#" antes del parámetro para indicar que la base numérica es 16.

Por ejemplo, si se requiere activar la memoria RAM de pesos se escribe la siguiente línea.

MEN #0

O en su defecto

MEN 0

Las instrucciones se pueden escribir con mayúsculas o minúsculas. Cualquier línea que no contenga una entrada válida no será tomada en cuenta. Para escribir un valor en una localidad de memoria

hay que realizar varios pasos. Primero se activa la memoria a utilizar, se escribe la dirección y finalmente se escribe el dato. Por ejemplo, el siguiente código escribe 4000h en la localidad 0 de la memoria de pesos.

```
MEN     #00
DIR     #0000
WDT     #4000
```

No es necesario especificar todos los bits de los buses, en el código anterior se puede escribir tan solo DIR #0 ó DIR 0. Para seguir escribiendo datos en la misma memoria no es necesario habilitarla cada vez que se requiera enviar un dato. La interfaz del procesador tiene un registro donde guarda el valor de la memoria que se habilita, por lo tanto, sólo se tiene qué escribir la memoria y el dato. También se tiene un registro del bus de direcciones.

La interfaz de usuario indica si la transmisión se realizó correctamente o si hubo algún error. Existen dos tipos de errores: aquellos provocados por la pérdida de la comunicación, en la que la tarjeta no responde al comando enviado y aquellos provocados por la transmisión de un dato incorrecto. La figura B.3 muestra la respuesta que obtiene la aplicación al enviar instrucciones a la tarjeta.

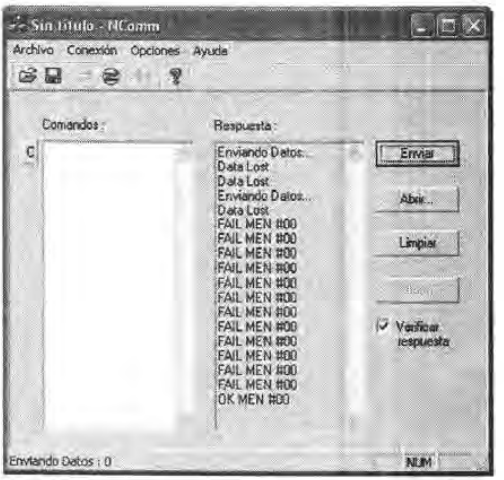


Fig. B.3. Respuesta y errores de comunicación.

Si existió un error por pérdida de dato, se envía un mensaje al usuario para continuar con los intentos de transmisión o para dejar de transmitir las instrucciones. Si existió un error por debido a un dato

incorrecto, éste se muestra en pantalla y automáticamente se reenvía la instrucción.

Si la instrucción se envía correctamente no se muestra nada en pantalla a menos se que tenga activada la casilla de verificar respuesta, con lo cual se muestra la instrucción que se ha enviado correctamente. No es bueno tener esta casilla activada cuando se tienen grandes cantidades de datos a enviar, ya que se consumen recursos de la computadora y se vuelve más lenta.

En la barra de estado se puede observar los datos que se están enviando a la tarjeta. Se puede configurar el puerto por el cual se quiere realizar la transmisión así como la velocidad, aunque la tarjeta tiene velocidad fija de 19200 bps.

El siguiente listado es un ejemplo de un fragmento de código producido por un programa que lee datos de pesos generados por matlab para configurar una red neuronal.

```
--Registros
RUN0
MEN #06
WDT #0008
MEN #07
WDT #0005
MEN #08
WDT #0012
MEN #09
WDT #0010
MEN #0A
WDT #0001
MEN #0B
WDT #0013
MEN #04
DIR #0013
WDT #1000
MEN #05
WDT #1000
MEN #10
WDT #0002

--Entradas
MEN #03
DIR #0000
WDT #0000
DIR #0001
WDT #0000
DIR #0002
```

```
WDT =0000
DIR =0003
WDT =0000
DIR =0004
WDT =0000

--Umbrales
MEN =02
DIR =0000
```

Como puede observarse, es bastante sencillo automatizar el procedimiento para configurar cualquier tipo de red neuronal con esta interfaz, sin embargo, como también puede observarse se tienen grandes desventajas. La primera es que mientras el procesador permite leer o introducir un dato por ciclo de reloj y en un paso, la interfaz utiliza de uno a tres pasos para realizar una tarea tan sencilla y además entre estos pasos existen muchos ciclos de instrucción perdidos que el procesador simplemente espera a que el dato se haya transmitido por el puerto serie. Así, la configuración de una red neuronal no se lleva fracciones de segundos, sino algunos minutos dependiendo de la cantidad de datos y de la calidad de la transmisión serial.

Otra desventaja es que esta interfaz es pasiva, es decir, su forma de operar es esperar una instrucción y ejecutarla. En realidad, la operación del procesador originalmente fue pensada de tal forma que tuviera una gran interacción con el exterior y para esto se necesita de una interfaz más flexible, de una arquitectura completamente paralela que permita realizar operaciones generales como lectura y escritura en puertos de E/S y transmisión de datos.

Una ventaja de los dispositivos más grandes FPGA, es que contienen procesadores embebidos, para esta aplicación solamente se tendría que configurar una interconexión entre el procesador embebido y el neuronal.

Para casos específicos, el procesador embebido tendría la tarea de actualizar las entradas tomando lecturas de los puertos y de obtener las salidas del procesador neuronal para enviarlas a los buses o dispositivos correspondientes. Con una arquitectura de este tipo se podría programar el tipo de acción que se requiera en vez de configurar un circuito secuencial que se encargue de realizar las mismas tareas como el que se programó para la aplicación del robot móvil.

Una interfaz de este tipo se muestra en la figura B.4.

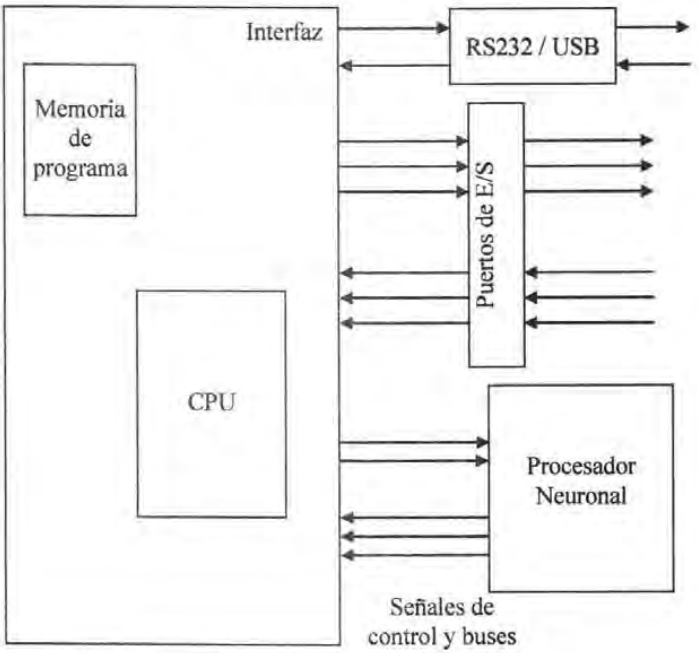


Fig. B.4. Interfaz programable.

Este tipo de interfaz es más completa, ya que se puede transmitir primeramente un programa que indique las operaciones a realizar y posteriormente los datos de forma más eficiente. Por otra parte, este esquema es adecuado para aplicaciones en tiempo real donde se requieran actualizar entradas de forma eficiente así como leer salidas.

APÉNDICE C

Diagrama de la arquitectura del procesador neuronal

La arquitectura consta de 6 niveles de pipeline que son indispensables para realizar todas las tareas de forma paralela de tal modo que se pueda ejecutar una sinapsis en un ciclo de reloj. Estas tareas son:

- Nivel 1: Contadores principales, generan las direcciones para obtener los datos de las memorias de pesos, umbrales y conexión de la sinapsis.
- Nivel 2: Obtención del peso, umbral y sinapsis. Decodificación de la entrada a la que está conectada la sinapsis. Decodificación de la memoria de neurona en uso actual.
- Nivel 3: Obtención de los datos de la memoria de neurona y de la memoria de entrada del usuario. Decodificación de la memoria de neurona de uso actual. Almacenamiento del resultado y obtención de la salida.
- Nivel 4: Obtención y/o almacenamiento de la entrada en uso a la red.
- Nivel 5: Obtención del valor de dato de entrada a la sinapsis en evaluación. Obtención del producto, corrimiento del resultado y acumulación con el resultado anterior.
- Nivel 6: Obtención del resultado final de la función de activación elegida.

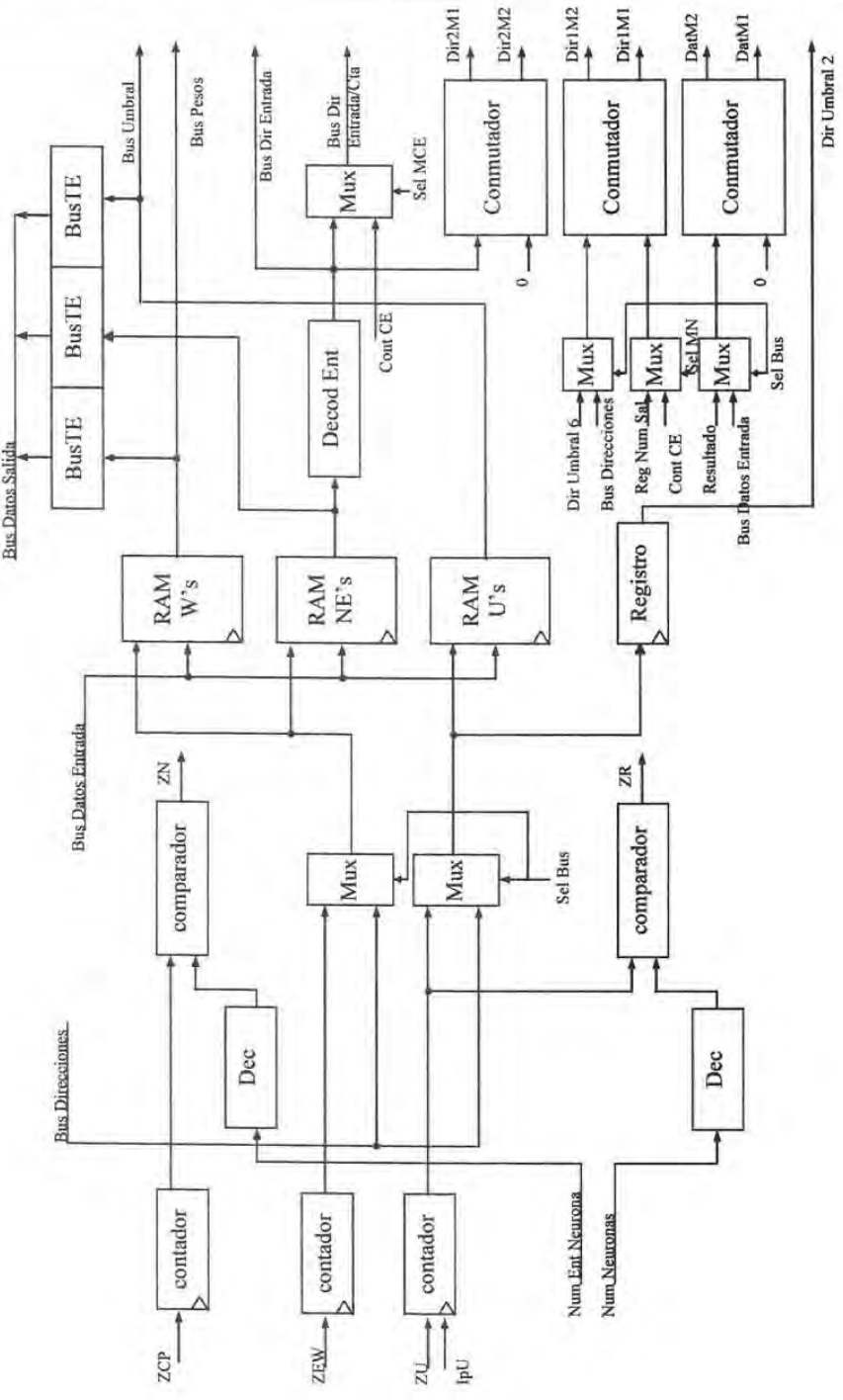
El número de niveles de pipeline se puede incrementar para añadir funciones nuevas al procesador o para incrementar la velocidad de ejecución. Sin embargo, no se puede disminuir debido a que el ciclo no podría completarse y se necesitaría de más de un ciclo de reloj para procesar una sinapsis.

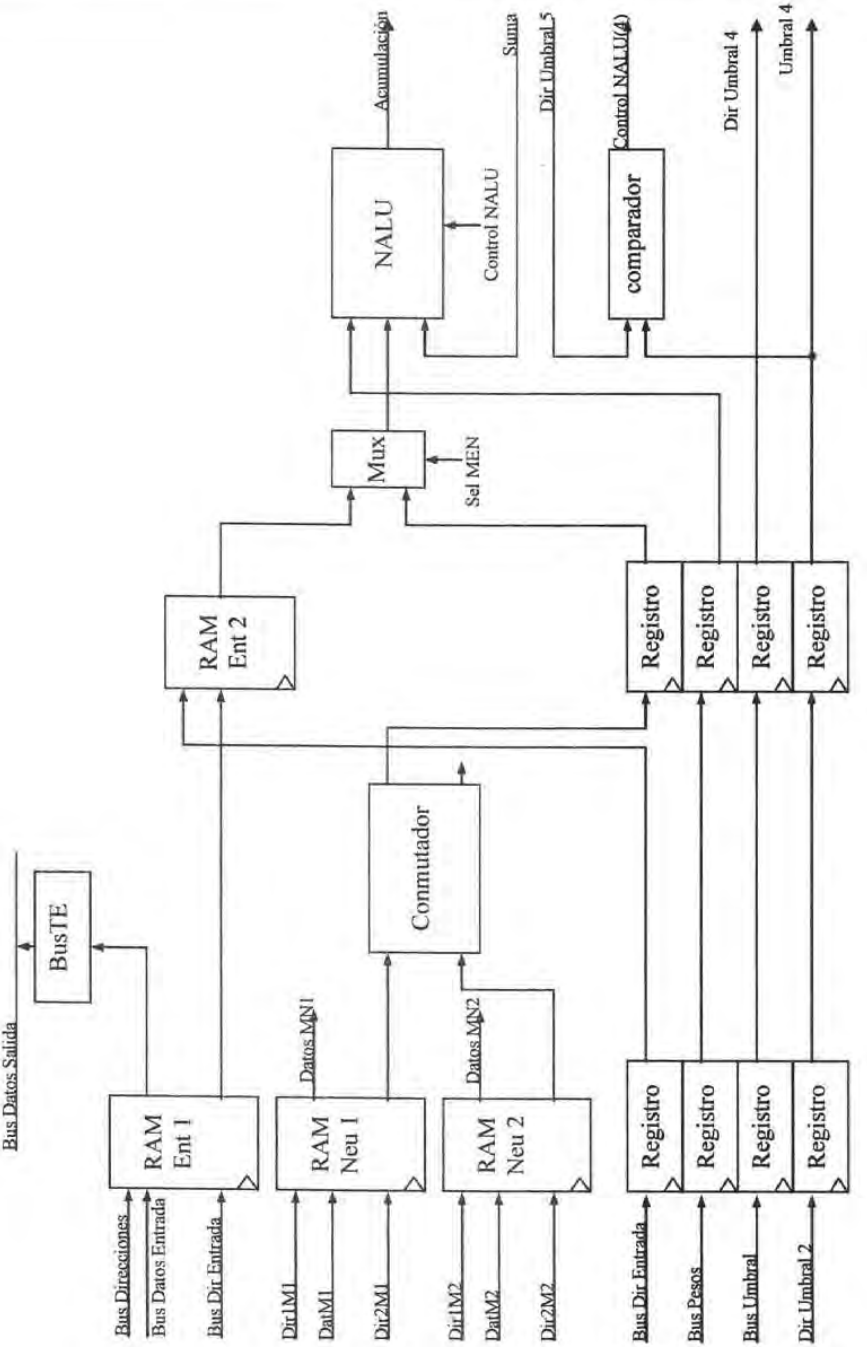
Existen algunos elementos que no pertenecen al ciclo de pipeline pero son indispensables en la arquitectura, de hecho, uno de estos elementos es el bloque de control del procesador, el cual es independiente de estos niveles y su función es controlar las señales de bloques como contadores y señales de selección de multiplexores principalmente.

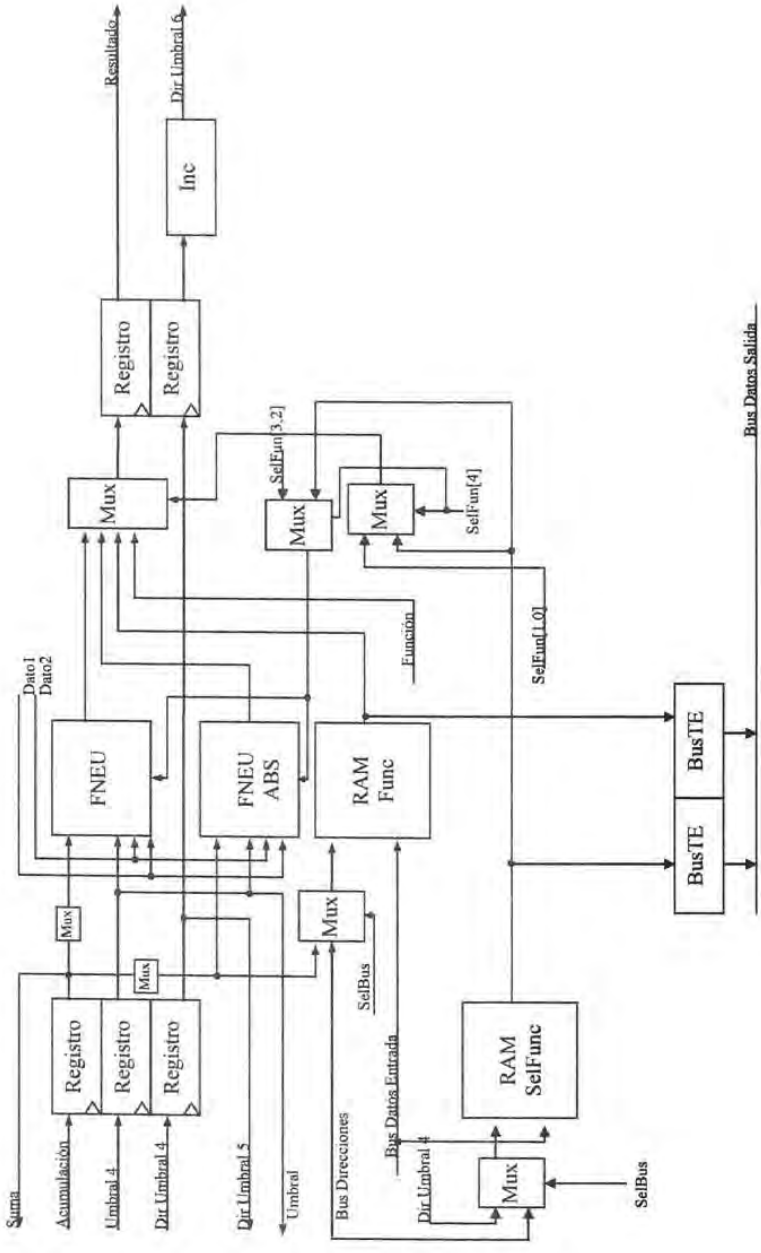
Otro bloque independiente es el decodificador de memorias, que es un decodificador común que solo activa una señal de salida que es dependiente de la entrada, finalmente, se tienen los registros donde se guarda información que utiliza el procesador para realizar las operaciones de forma adecuada.

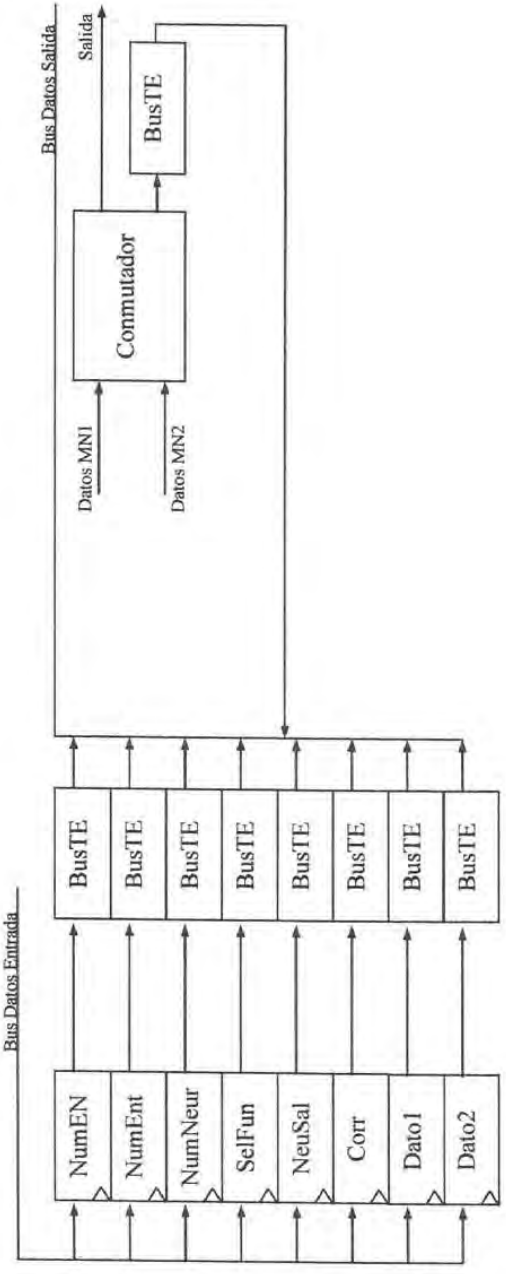
El código VHDL de cada uno de los bloques no se presenta aquí por su gran extensión, pero se incluye en el CD.

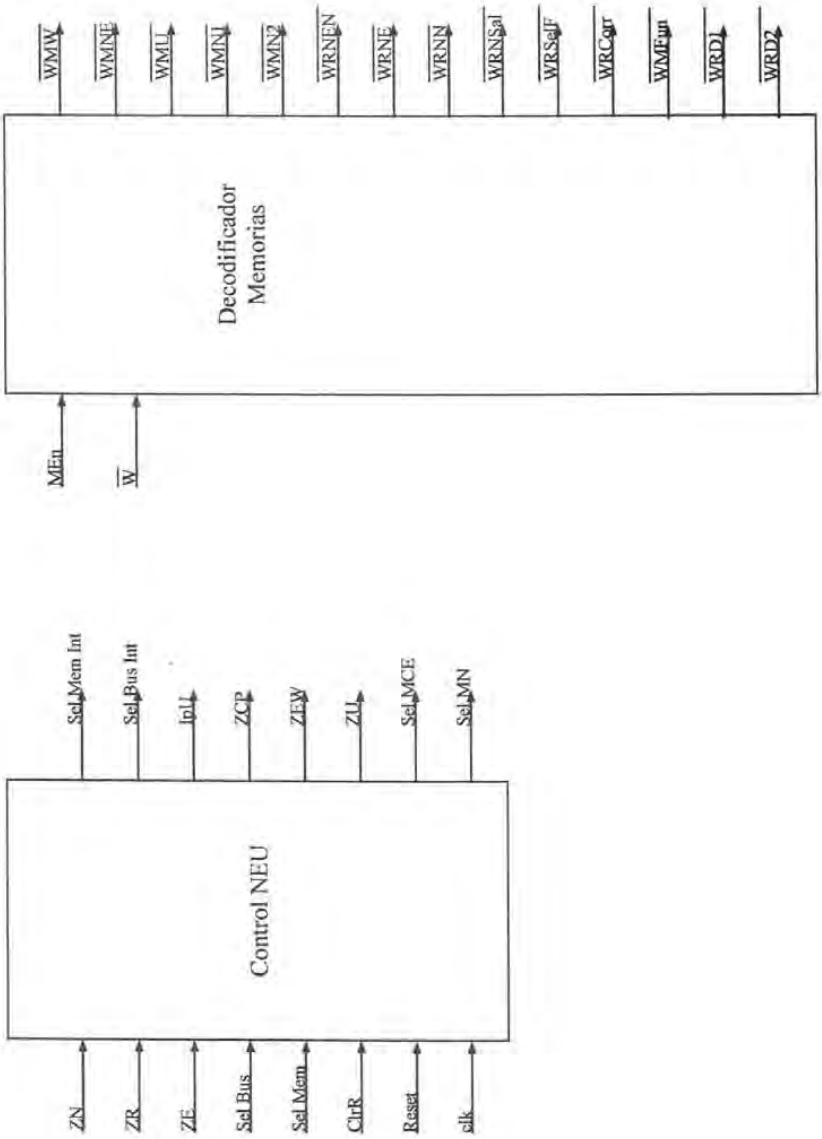
Los siguientes diagramas muestran la arquitectura del procesador neuronal.


ZCP
ZEW
ZU
IpU
contador
contador
contador
Num Ent Neurona
Num Neuronas
Bus Direcciones
Dec
Dec
comparador
comparador
ZN
ZR
Bus Datos Entrada
Mux
Mux
Sel Bus
RAM W's
RAM NE's
RAM U's
Registro
Bus Datos Salida
BusTE
BusTE
BusTE
Bus Umbral
Bus Pesos
Bus Dir Entrada
Decod Ent
Cont CE
Mux
Bus Dir Entrada/Cta
Sel MCE
Conmutador
Conmutador
Conmutador
Dir2M1
Dir2M2
Dir1M2
Dir1M1
DatM2
DatM1
Dir Umbral 2
0
0
Mux
Mux
Mux
Dir Umbral 0
Bus Direcciones
Reg Num Sal
Cont CE
Sel MN
Resultado
Bus Datos Entrada
Sel Bus


Bus Datos Salida
BusTE
RAM Ent 1
RAM Neu 1
RAM Neu 2
RAM Ent 2
Conmutador
Mux
Sel MEN
NALU
Control NALU
Acumulación
Suma
comparador
Control NALU(4)
Dir Umbral 5
Dir Umbral 4
Umbral 4
Registro
Registro
Registro
Registro
Registro
Registro
Registro
Registro
Bus Direcciones
Bus Datos Entrada
Bus Dir Entrada
Dir1M1
DatM1
Dir2M1
Datos MN1
Dir1M2
DatM2
Dir2M2
Datos MN2
Bus Dir Entrada
Bus Pesos
Bus Umbral
Dir Umbral 2


Resultado
Dir Umbral 6
Inc
Registro
Registro
Mux
SelFun[3.2]
Mux
Mux
SelFun[4]
Bus Datos Salida
Función
SelFun[1.0]
Dato1
Dato2
FNEU
FNEU ABS
RAM Func
BusTE
BusTE
Mux
Mux
Mux
SelBus
Registro
Registro
Registro
RAM SelFunc
Suma
Acumulación
Umbral 4
Dir Umbral 4
Dir Umbral 5
Umbral
Bus Direcciones
Bus Datos Entrada
Dir Umbral 4
Mux
SelBus


Bus Datos Salida
Salida
BusTE
Conmutador
Datos MN1
Datos MN2
BusTE
BusTE
BusTE
BusTE
BusTE
BusTE
BusTE
BusTE
Bus Datos Entrada
NumEN
NumEnt
NumNeur
SelFun
NeuSal
Corr
Dato1
Dato2


WMW
WMNE
WML
WMN1
WMN2
WRNEN
WRNE
WRNN
WRNSal
WRSelF
WRCorr
WMFun
WRD1
WRD2
Decodificador Memorias
MEn
W
SelMem Int
SelBus Int
IpLL
ZCP
ZEW
ZU
SelMCE
SelMN
Control NEU
ZN
ZR
ZE
Sel Bus
Sel Mem
ClrR
Reset
clk

APÉNDICE D

Aplicación utilizando un Robot Móvil

Un tipo de aplicaciones donde la arquitectura puede resultar muy útil es en sistemas autónomos, para este caso se utilizó la tarjeta para controlar el movimiento de un robot móvil. El problema consiste en entrenar una red neuronal que permita que el robot avance y a la vez no choque contra algún obstáculo que se le presente en su trayectoria como se muestra en la figura D.1.

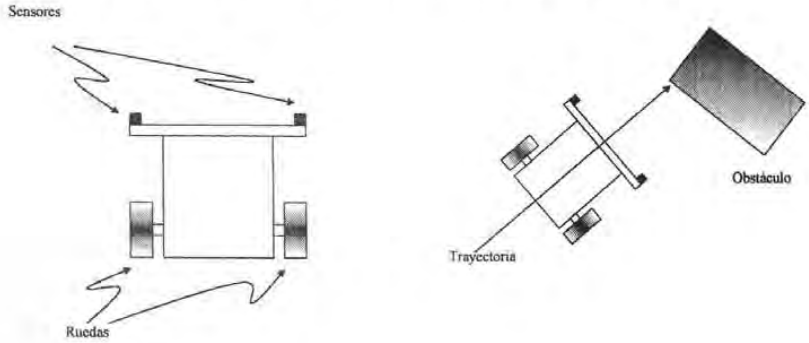


Fig. D.1. Robot Móvil

La aplicación consiste en que la red realice todo el procesamiento y no resuelva tan solo una parte del problema, esto es, que las entradas se conecten directamente a una red neuronal y las salidas de la red al sistema. En este caso las entradas provienen de las lecturas de los sensores y las salidas se conectan a los motores.

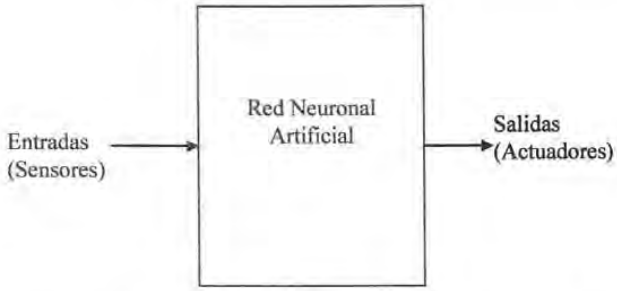


Fig. D.2. Red neuronal para controlar el robot móvil.

La red debe ser capaz de generar distintos comportamientos, en este caso el problema consiste en evadir obstáculos, por lo tanto, cuando uno o ambos sensores detecten un objeto frente al robot, la salida debe modificarse de tal manera que el sistema actúe y evite una colisión con dicho objeto.

Una forma de resolver el problema, es asignar una salida distinta para cada combinación de entrada de los sensores. Por, ejemplo, si las lecturas de los sensores son "1" indicando la presencia de un objeto y "0" indicando la ausencia; la lectura "00" indicará que no tiene ningún obstáculo en frente y por lo tanto puede asignar las salidas de tal manera que el robot avance hacia delante. Se pueden asignar valores binarios a las salidas, de tal manera que indiquen avance (con 1) o paro (con 0) y otro par de valores indicarán el sentido de cada motor. Así se pueden tener 4 salidas indicando la forma en que el robot debe moverse.

Para este problema se asignaron los siguientes valores para la salida de los motores:

Entradas		Acción
0	0	Motor parado
0	1	Giro hacia enfrente
1	0	Giro hacia atrás
1	1	Motor parado

Un problema que se tiene con los sensores, es que su alcance es limitado ya que su principio de funcionamiento es fotosensible: detectan luz infrarroja. La distancia a la que se puede detectar un obstáculo depende de las condiciones del ambiente y de los parámetros del objeto como su color y su superficie (como textura, reflexión, etc.), pero en la práctica para los objetos claros se tiene un intervalo de detección de aproximadamente 3 cm.

Este problema quizá se pueda resolver con una red de tipo Perceptrón, pero se elegirá una red de retropropagación, por el hecho de que de antemano sabemos que al tener al menos una capa de

neuronas ocultas, se puede entrenar la red con patrones que pueden no ser linealmente separables. Por otra parte, se propone realizar una máquina de estados para resolver el problema. La misma red será la encargada de generar el estado correspondiente en función de los sensores de entrada, el estado presente y una señal auxiliar proveniente de un contador que indicará un tiempo *t*. Este parámetro será utilizado para que cuando la red detecte algún objeto, pase a un estado de evasión y salga de este estado al transcurrir el tiempo *t*. La figura D.3 muestra la red propuesta y en la figura D.4 se muestra la solución propuesta.

Las entradas a la red son las dos lecturas de los sensores, el estado en el que se encuentra el robot y un valor binario proveniente de un contador que indica si ha transcurrido un tiempo *t*, también se ha considerado una entrada que le indica al robot cuando caminar o si éste debe quedarse parado. La red genera salidas para los motores, el estado futuro y una variable (*zt*) que inicializa el contador de tiempo. Así, mientras que *zt* esté activo, el contador se establece a cero.

En la figura D.4 se puede ver el propósito de estas salidas: Originalmente, si la entrada *camina* es 0, el robot se mantiene en su posición, en este punto no hay ningún problema ya que aunque los sensores detecten la presencia de un objeto, el robot nunca chocará. Los estados presente y futuro para esta entrada son 0. Cuando *camina* se activa, el estado presente del robot sigue siendo 0 y éste ahora depende de las entradas de los sensores. Si ambos sensores no detectan algún objeto, las salidas se ambos motores indican giro hacia delante con lo que el robot avanza en línea recta.

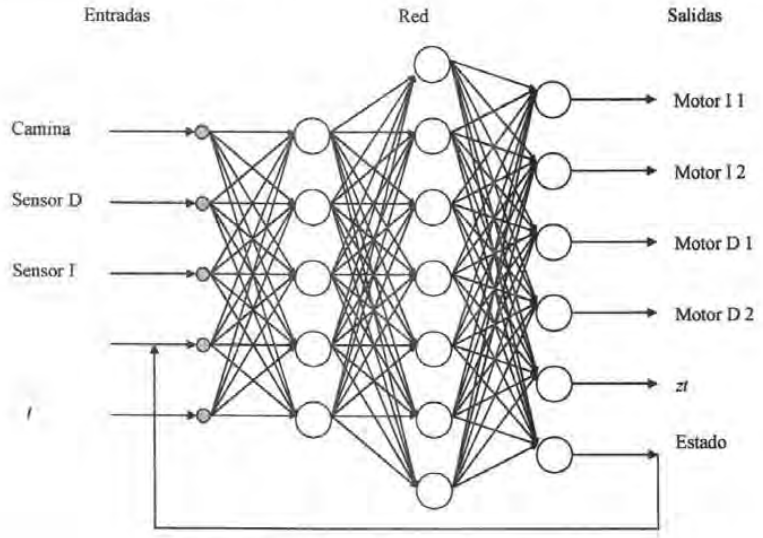


Fig. D.3. Red propuesta para resolver el problema.

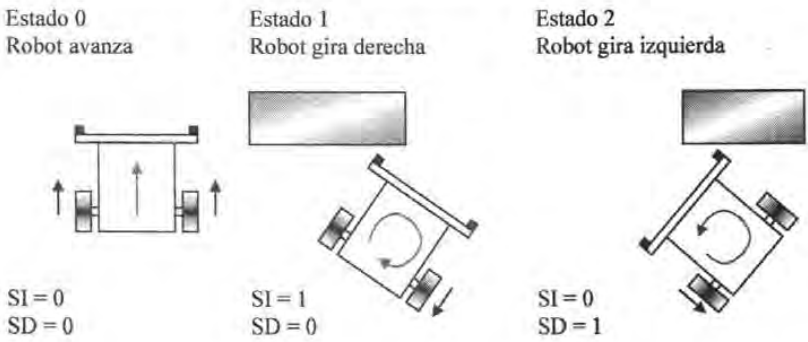


Fig. D.4. Solución al problema.

Cuando un objeto es detectado, el estado del robot cambia inmediatamente, dependiendo del sensor que se activó. Si se activó el de la izquierda, el robot pasa al estado 1 donde girará hacia la derecha, en caso de haberse activado el sensor de la derecha, el robot girará hacia la izquierda estando en el estado 2. En la figura D.4 se muestra el sentido de giro de las llantas para que el robot logre evitar el objeto. La salida de los estados 1 y 2 está determinada por la entrada *t*. El tiempo en el que el robot se mantiene en uno de estos estados depende de la construcción física de éste, así como de su dinámica. Sin embargo, este valor puede ajustarse en forma práctica ya que existe un intervalo de valores para el cual se tienen resultados satisfactorios.

Para entrenar a la red se generan patrones de entrada y salida. La red fue entrenada en matlab y se probaron distintas configuraciones variando el número de neuronas de la capa oculta. Se obtuvieron mejores resultados (menor error) con 7 neuronas en esta capa. Los patrones utilizados para entrenar a la red se muestran en la siguiente tabla.

Camina	SI	SD	Edo P	*t*	MI1	MI2	MD1	MD2	*zt*	Edo S
0	0	0	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0	1	0
0	1	1	0	0	0	0	0	0	1	0
1	0	0	0	0	1	0	1	0	1	0
1	1	0	0	0	0	0	0	1	0	1
1	1	0	1	0	0	0	0	1	0	1
1	0	1	1	0	0	0	0	1	0	1
1	1	1	1	0	0	0	0	1	0	1
1	0	0	1	0	0	0	0	1	0	1
1	0	0	1	1	1	0	1	0	1	0
1	1	0	1	1	1	0	1	0	1	0
1	0	1	1	1	1	0	1	0	1	0
1	1	1	1	1	1	0	1	0	1	0
1	0	1	0	0	0	1	0	0	0	2

1	0	1	2	0	0	1	0	0	0	2
1	0	0	2	0	0	1	0	0	0	2
1	1	0	2	0	0	1	0	0	0	2
1	1	1	2	0	0	1	0	0	0	2
1	0	0	2	1	1	0	1	0	1	0
1	1	0	2	1	1	0	1	0	1	0
1	0	1	2	1	1	0	1	0	1	0
1	1	1	2	1	1	0	1	0	1	0
1	1	1	0	0	0	0	0	1	0	1
1	1	1	1	0	0	0	0	1	0	1

Tabla D.1. Patrones de entrenamiento de la red.

Las siguientes instrucciones se utilizaron en matlab para entrenar a la red:

```
net = newff([0 1; 0 1; 0 1; 0 2; 0 1], [5 7 6], {'tansig' 'tansig' 'purelin'});

P=[[0; 0; 0; 0; 0] [0; 1; 0; 0; 0] [0; 0; 1; 0; 0] [0; 1; 1; 0; 0] [1; 0; 0; 0; 0]
[1; 1; 0; 0; 0] [1; 1; 0; 1; 0] [1; 0; 0; 1; 0] [1; 0; 0; 1; 1] [1; 1; 0; 1; 1] [1; 0; 1; 1; 1]
[1; 1; 1; 1; 1] [1; 0; 1; 0; 0] [1; 0; 1; 2; 0] [1; 0; 0; 2; 0] [1; 0; 0; 2; 1] [1; 1; 0; 2; 1]
[1; 0; 1; 2; 1] [1; 1; 1; 2; 1] [1; 1; 1; 0; 0] [1; 1; 1; 1; 0] [1; 0; 1; 1; 0] [1; 1; 1; 1; 0]
[1; 1; 0; 2; 0] [1; 1; 1; 2; 0]];

T = [[0; 0; 0; 0; 1; 0] [0; 0; 0; 0; 1; 0] [0; 0; 0; 0; 1; 0] [0; 0; 0; 0; 1; 0] [1; 0; 1; 0; 1; 0]
[0; 0; 0; 1; 0; 1] [0; 0; 0; 1; 0; 1] [0; 0; 0; 1; 0; 1] [1; 0; 1; 0; 1; 0] [1; 0; 1; 0; 1; 0]
[1; 0; 1; 0; 1; 0] [1; 0; 1; 0; 1; 0] [1; 0; 1; 0; 1; 0] [0; 1; 0; 0; 0; 2] [0; 1; 0; 0; 0; 2] [0;
1; 0; 0; 0; 2] [1; 0; 1; 0; 1; 0] [1; 0; 1; 0; 1; 0] [1; 0; 1; 0; 1; 0] [1; 0; 1; 0; 1; 0] [0;
0; 0; 1; 0; 1] [0; 0; 0; 1; 0; 1] [0; 0; 0; 1; 0; 1] [0; 0; 0; 1; 0; 1] [0; 1; 0; 0; 0; 2] [0;
1; 0; 0; 0; 2]];

net.trainParam.epochs = 1000;
net = train(net,P,T);
```

La primer linea define una red neuronal de tipo retropropagación (*newff*). La primera serie de parámetros definen los intervalos de valores que toman las entradas. Todas las entradas toman valores binarios de '0' y '1' excepto el estado que define valores hasta 2. El siguiente parámetro (*[5 7 6]*) define el número de neuronas por capa. Para la capa de entrada se tienen 5 neuronas, mientras que para la capa de salida se tienen 6. En este caso, como ya se mencionó, el número de neuronas de la capa oculta es 7. Finalmente se definen las funciones de transferencia por capa de neuronas. Para las dos primeras se definió la función tangente hiperbólica mientras que para la capa de salida se optó por una función lineal.

Los parámetros *P* y *T* definen los valores de entrada y salida deseados, es decir, en conjunto forman los patrones con los que la red será entrenada y los cuales se tomaron de la tabla D.1. Finalmente, se definen los parámetros de entrenamiento de la red, definiendo 1000 ciclos (ver algoritmo de entrenamiento de la red retropropagación). La última sentencia *train*, se utiliza para entrenar la red, especificándola con el parámetro *net* y los patrones definidos por *P* y *T*.

Hay que señalar que para configurar una red de este tipo en la tarjeta no es necesario modificar el programa, sin embargo, para esta aplicación, es necesario proveer algunos parámetros a las entradas y leer las salidas. Por esta razón, se añadieron algunos módulos extra a la interfaz del procesador. Estos módulos se encargan de leer valores directamente de puertos de entrada y obtener las salidas que genera el procesador para enviarlas a los pines correspondientes. Así mismo se encarga de realimentar el valor del estado que genera la red a la entrada respectiva.

Con estas modificaciones se pudo probar el algoritmo en un robot móvil, el cual se muestra en la figura D.5.

Una ventaja de utilizar redes neuronales para el control de robots móviles, es que al cambiar los valores de los pesos de las sinapsis, se tienen comportamientos distintos. No es necesario cambiar la red neuronal en todos los casos para que el robot realice otras actividades. Otra ventaja es que se puede enseñar al robot a realizar una labor específica sin tener que cambiar el programa que lo maneja, de hecho, no se tiene que saber programar para hacer que el robot efectúe las tareas que el usuario requiera, el robot aprende a partir de ejemplos.

Una ventaja adicional que se tiene específicamente con esta tarjeta, es que se pueden implementar máquinas de estados que controlen el comportamiento de la red. Se pueden tener distintos niveles de comportamiento y neuronas de interconexión de salidas de cada máquina de estados. Algo parecido a lo que se realiza con máquinas de estados aumentadas, aquí los nodos de supresión serían neuronas o redes y cada máquina estaría conformada por una RNA. Así tan solo sería necesario entrenar cada máquina para que realice una función específica. Con la tarjeta utilizada se pueden configurar varias redes con miles de sinapsis cada una y se tendría una respuesta en tiempo real.

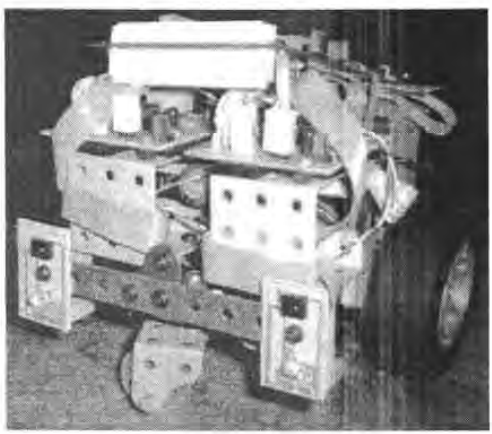


Fig. D.5. Robot móvil utilizado para probar el algoritmo de evasión de obstáculos.

APÉNDICE E

Especificaciones de la tarjeta de evaluación Virtex II

Tarjeta de evaluación Virtex II:

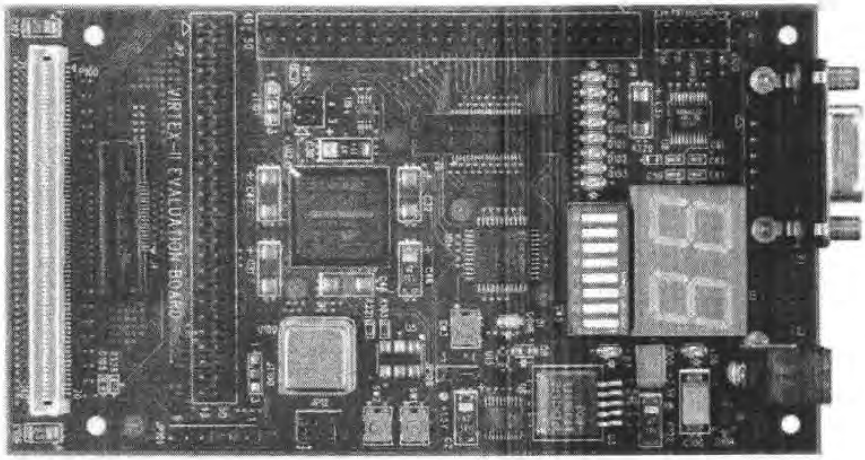


Fig. D.1. Tarjeta con FPGA Virtex II XC2V1000.

Características:

- FPGA Xilinx XC2V1000-4FG256
- EEPROM de configuración XC18V04VQ44C
- Conexiones:
  - Dos headers de 50 pines.
  - Conectores Mictor
  - Conector de expansión AvBus.
- Tensiones de trabajo:
  - 5 V Externa
  - 3.3 V Interna
  - 1.5 V Interna
- Puerto serial RS 232
- Conector JTAG para configuración.
- Dip switch
- 8 Leds
- 2 Push Buttons
- Display de 7 segmentos doble
- Oscilador de 40 MHz.
- Alimentación de 5V, 1A.

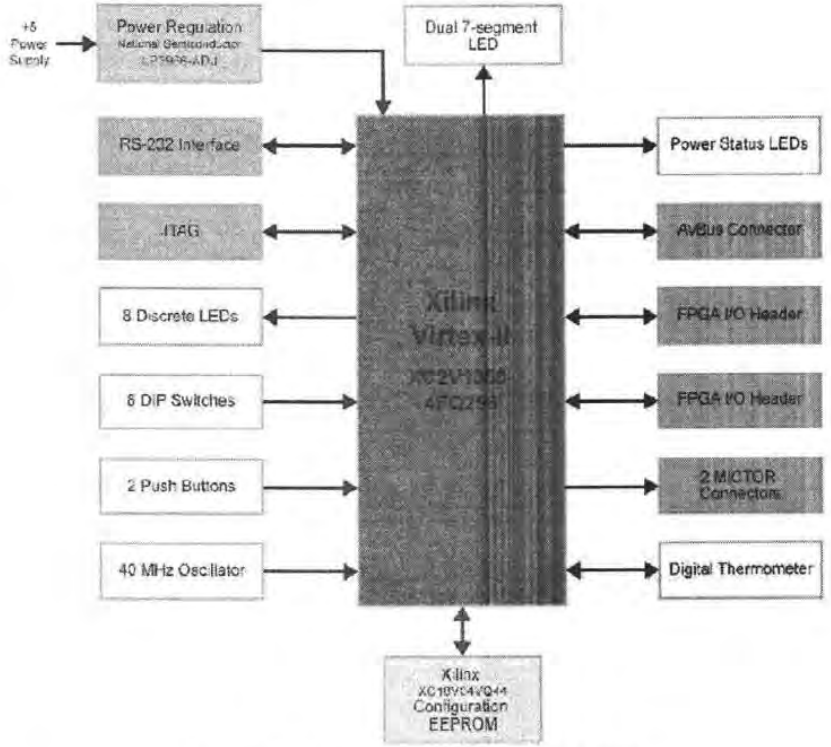


Fig. D.2. Diagrama a bloques del sistema.

Características de la familia Virtex II:

Device	System Gates	CLB (1 CLB = 4 slices = Max 128 bits)			Multiplier Blocks	SelectRAM Blocks		DCMs	Max I/O Pads[1]
		Array Row x Col.	Slices	Maximum Distributed RAM Kbits		18 Kbit Blocks	Max RAM (Kbits)		
XC2V40	40K	8 x 8	256	8	4	4	72	4	88
XC2V80	80K	16 x 8	512	16	8	8	144	4	120
XC2V250	250K	24 x 16	1,536	48	24	24	432	8	200
XC2V500	500K	32 x 24	3,072	96	32	32	576	8	264
XC2V1000	1M	40 x 32	5,120	160	40	40	720	8	432
XC2V1500	1.5M	48 x 40	7,680	240	48	48	864	8	528
XC2V2000	2M	56 x 48	10,752	336	56	56	1,008	8	624
XC2V3000	3M	64 x 56	14,336	448	96	96	1,728	12	720
XC2V4000	4M	80 x 72	23,040	720	120	120	2,160	12	912
XC2V6000	6M	96 x 88	33,792	1,056	144	144	2,592	12	1,104
XC2V8000	8M	112 x 104	46,592	1,456	168	168	3,024	12	1,108

BIBLIOGRAFÍA

[1] Andrés Pérez. Structure-Adaptable Digital Neural Networks. École Polytechnique Fédérale De Lausanne. 1999.

[2] Ma Del Pilar Gómez. Fundamentos En Redes Neuronales Artificiales. Universidad de las Américas. Puebla.

[3] María I. Acosta, Camilo A. Zuluaga. Tutorial Sobre Redes Neuronales Aplicadas En Ingeniería Eléctrica. Universidad Tecnológica de Pereira.
Pereira. 2000.

[4] José R. Hilera, Víctor J. Martínez. Redes Neuronales Artificiales. Fundamentos, modelos y aplicaciones. Alfaomega. 2000.

[5] Anil K. Jain, Jiangchang Mao. Artificial Neural Networks: A Tutorial. IEEE 1996.

[6] Herminia Pasantes. De Neuronas, Emociones y Motivaciones. Fondo de cultura económica. 1997.

[7] David G. Maxinez, Jessica Alcalá. VHDL El Arte De Programar Sistemas
Digitales. CECSA. 2003.

[8] Fernando Prado Carpio. VHDL Lenguaje Para Descripción y Modelado
De Circuitos. Universidad de Valencia. 1997.

[9] Miguel A. Aguirre, Jonathan N. Tombs, Fernando M. Chavero. Lenguajes De Alto Nivel Para Diseño De Circuitos Integrados Digitales.
Dep. De Ingeniería Electrónica. Universidad de Sevilla.

[10] Altera Corporation. www.altera.com

[11] Xilinx Inc. www.xilinx.com

[12] Jesús Savage, Gabriel Vázquez. Diseño de Microprocesadores. Facultad de Ingeniería. UNAM.

[13] J. Valeriano, G. Calva. Implantación Digital De La Neurona Artificial Perceptrón Utilizando CPLDs. Centro de Instrumentos. UNAM.

[14] A. Chohra, P. Schöll. Neural Networks (NN) Based Learning of Elementary Behaviors and their Integration in FPGA Architectures for a Fast Moving Robot Team (RoboCup). GMD-German National Research Center for Information Technology. Germany.

[15] J. L. Ayala, A. G. Lomeña, M. López, A. Fernández. Design Of a Pipelined Hardware Architecture For Real-Time Neural Network Computations. Departamento de Ingeniería Electrónica. UPM.

[16] Denis F. Wolf, Roselia F. Romero, Eduardo Marques. Using Embedded Processors In Hardware Models Of Artificial Neural Networks. Universidade de Sao Paulo. Brasil.

[17] Geert Deconinck. A Massively Parallel Architecture for Hopfield-type Neural Network Computers. Katholieke Universiteit Leuven. Electrical Engineering Department. Belgium.

[18] Jean-Luc Beuchat, Jaques-Oliver Haenni and Eduardo Sanchez Hardware Reconfigurable Neural Networks. Swiss Federal Institute of Technology, Logic Systems Laboratory.

[19] Mario Gómez Martinez. Teoria y simulación de Redes Neuronales Artificiales. Instituto de Investigación en Inteligencia Artificial (IIIA). Bellaterra, SPAIN. 1999.

[20] Savage Carmona Jesús. Navegación de Robots Móviles Usando Sonares. Facultad de Ingeniería. UNAM. 2000.

[21] Ronald C. Arkin. Behavior-Based Robotics. The MIT Press. Cambridge, Massachusetts.

[22] Microchip. PIC 18F6520/8520/6620/8620/6720/8720 Data Sheet. 2004 Microchip Technology Inc.