ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN
UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Proyecto Fin de Carrera

Estrategias para el desarrollo de aceleradores hardware de algoritmos basados en Redes Neuronales

AUTOR: José Carlos Fernández Conesa
DIRECTOR: Javier Garrigós Guerrero

Febrero / 2006

Autor	José Carlos Fernández Conesa
Correo-e del autor	zirition@gmail.com
Director	Javier Garrigós Guerrero
Correo-e del director	j.garrigos@upct.es

Título del PFC	Estrategias para el desarrollo de aceleradores hardware de algoritmos basados en Redes Neuronales
Descriptores	Redes neuronales artificiales, Sistemas en un Chip, procesadores de propósito específico

Resumen

La utilización de coprocesadores hardware para acelerar la ejecución de algoritmos de cómputo es una técnica comúnmente utilizada en las arquitecturas de las computadoras comerciales. Estos coprocesadores sin embargo son de propósito general, significando que aceleran las operaciones más comunes en los sistemas de cómputo convencionales. En ciertas ocasiones, no obstante, se hace conveniente disponer de aceleración por hardware de algoritmos de cómputo no convencionales, como los basados en técnicas de soft-computing (Redes Neuronales, Lógica Borrosa y Algoritmos Genéticos). En estos casos, se hace necesario el desarrollo de procesadores de propósito específico, especialmente adaptados a la naturaleza del problema y de los cálculos a realizar.

Dentro de este marco, el objetivo principal del presente proyecto es el desarrollo de una herramienta que ayude en el diseño de aceleradores hardware de Redes Neuronales Artificiales (ANNs) de tipo Perceptrón Multicapa (MLP) y similares.

Titulación	Ingeniero de telecomunicación
Intensificación	Sistemas y Redes de Telecomunicación
Departamento	Electrónica, Tecnología de Computadoras y Proyectos
Fecha de presentación	Febrero 2006

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Planteamiento inicial

La utilización de coprocesadores hardware para acelerar la ejecución de algoritmos de cómputo es una técnica comúnmente utilizada en las arquitecturas de las computadoras comerciales. Estos coprocesadores sin embargo son de propósito general, significando que aceleran las operaciones más comunes en los sistemas de cómputo convencionales. En ciertas ocasiones, no obstante, se hace conveniente disponer de aceleración por hardware de algoritmos de cómputo no convencionales, como los basados en técnicas de soft-computing (Redes Neuronales, Lógica Borrosa y Algoritmos Genéticos). En estos casos, se hace necesario el desarrollo de procesadores de propósito específico, especialmente adaptados a la naturaleza del problema y de los cálculos a realizar.

Dentro de este marco, el objetivo principal del presente proyecto es el desarrollo de una herramienta que ayude en el diseño de aceleradores hardware de Redes Neuronales Artificiales (ANNs) de tipo Perceptrón Multicapa (MLP) y similares.

1.2. Historia y panorama actual de las redes neuronales artificiales

Las redes neuronales artificiales son sistemas adaptativos universales, ya que se puede demostrar que tienen la capacidad de aproximar cualquier función continua no lineal, en un dominio compacto, con una precisión arbitraria. Estas redes neuronales están formadas por la inteconexión de un número de unidades funcionales idénticas, o *neuronas*. Cada interconexión entre dos neuronas tiene asignado un parámetro, o *peso*, el cual indica la fuerza de dicha conexión, de forma que el comportamiento de la red neuronal reside en dichos parámetros.

1.2.1. Historia

El punto de vista actual de las redes neuronales se generó en los años 40, gracias al trabajo de McCulloch y Pitts, que mostraron que, en principio, las redes neuronales pueden realizar cualquier operación aritmética o lógica.

Después apareció el trabajo de Donald Hebb, donde se propuso un mecanismo para el aprendizaje de las neuronas biológicas en su obra *The Organization of Behavior*. Su idea sugiere cómo la fuerza de la conexión entre dos neuronas debe modificarse de acuerdo a si, en un determinado momento, la conexión está activada. Así, si una neurona está estimulando a otra, al mismo tiempo que la neurona receptora está activa, entonces la fuerza de la conexión debe incrementarse, y viceversa — si la conexión se activa, pero la salida no se activa, entonces la fuerza de la conexión decrece —.

La primera aplicación práctica vino a finales de los años 50 de mano de Rosenblatt, con la invención del Perceptrón y de su regla de aprendizaje. En su trabajo, Rosenblatt construyó un Perceptrón y mostró su capacidad para el reconocimiento de patrones.

Al mismo tiempo, Widrow y Hoff desarrollaron un nuevo algoritmo de aprendizaje, el cual utilizaron para entrenar redes neuronales muy similares al Perceptrón de Rosenblatt.

Sin embargo, ambas arquitecturas sufren de la mismas limitaciones, que fueron ampliamente mostradas en el trabajo de Minsky y Papert. La principal de estas limitaciones era la imposibilidad de resolver cualquier problema que no fuera linealmente separable, lo cual limita enormemente su campo de aplicación. Minsky y Papert además diseñaron una nueva red que sería capaz de solucionar estas limitaciones, pero sin embargo no fueron capaces de adaptar las reglas de aprendizaje a la nueva arquitectura.

Durante los años 70, el desarrollo de las redes neuronales artificiales estuvo prácticamente estancado, debido a que el trabajo de Minsky y Papert parecía mostrar el final de estas redes, y a la falta de potencia de cálculo en los ordenadores para poder experimentar con ellas. No obstante, algunos trabajos, como los de Kohonen o el de Anderson demostraron que las redes neuronales podían trabajar también como memorias.

Durante los 80, los problemas de potencia de cálculo se fueron superando debido a los nuevos ordenadores personales y a las estaciones de trabajo. Además surgieron nuevos conceptos, como el uso de la estadística para explicar las operaciones de cierto tipo de redes neuronales recurrentes, que pueden ser usadas como memorias asociativas, tal y como describe Hopfield.

Otro desarrollo clave fue la invención del algoritmo Backpropagation para entrenar Perceptrones Multicapa, diseñado simultáneamente por varios investigadores. La publicación más influyente fue la de Rosenblatt y McClelland, que fueron capaces de solucionar los problemas mostrados por Minsky y Papert.

Estos nuevos desarrollos dieron un gran empuje al desarrollo de este campo. En los últimos años, se han escrito miles de trabajos y se han desarrollado cientos de aplicaciones y posibles usos para las redes neuronales.

1.2.2. Introducción a las redes neuronales

En la actualidad, las redes neuronales se pueden dividir en tres arquitecturas principales. En primer lugar destacaremos la arquitectura en capas, en la cual, como su nombre indica, las neuronas se encuentran dispuestas en varias capas, de forma que la salida de las neuronas de una capa son las entradas de las neuronas de la capa siguiente. Las capas se dividen en tres clases:

- Capa de entrada, que dependiendo del autor se refiere a la primera capa de neuronas, cuyas entradas son las entradas de la red, o a una capa ficticia que no realiza ningún procesamiento y únicamente recibe las entradas de la red. En esta Memoria se supondrá que la capa de entrada es la primera capa de neuronas de la red.
- Capas ocultas, que son aquellas invisibles desde el exterior de la red.
- Capa de salida, formada por las neuronas cuyas salidas son las salidas de la red.

En la figura 1.1 se observa un ejemplo de arquitectura en capas. Esta arquitectura es la más empleada y estudiada, siendo su principal exponente el Perceptrón Multicapa.

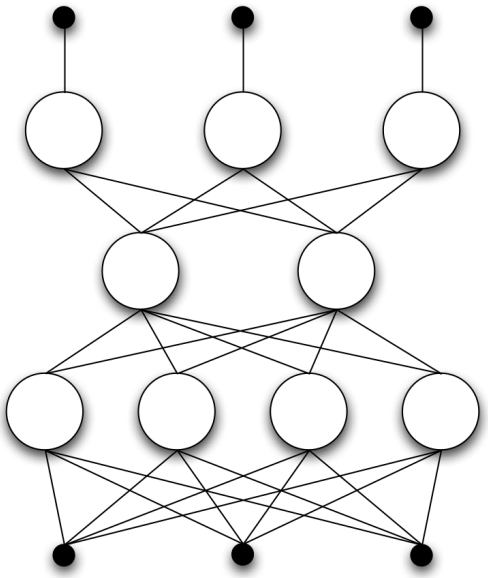


Figura 1.1. Estructura de una red multicapa

Como segunda arquitectura destacada se encuentra la arquitectura recurrente, mostrada en la figura 1.2, donde las neuronas incluyen bloques de realimentación, de forma que su salida actual dependa de sus salidas anteriores. Generalmente todas sus neuronas están conectadas entre sí, lo que dificulta su manejo y aumenta su complejidad. En esta arquitectura totalmente conectada, cualquier neurona puede ser entrada, salida de la red, o una unidad oculta. Las redes ART son un ejemplo de este tipo de arquitectura.

Finalmente se encuentran las redes mixtas, redes en capas con características de las redes recurrentes. Un ejemplo son las redes BAM.

Desde el punto de vista funcional, en las redes neuronales se observan dos fases diferenciadas. En primer lugar aparece la fase de aprendizaje, en la cual la red es entrenada para que responda de forma adecuada a la aplicación en la que se va a emplear. El aprendizaje consiste en la modificación de los parámetros o *pesos* de las neuronas de

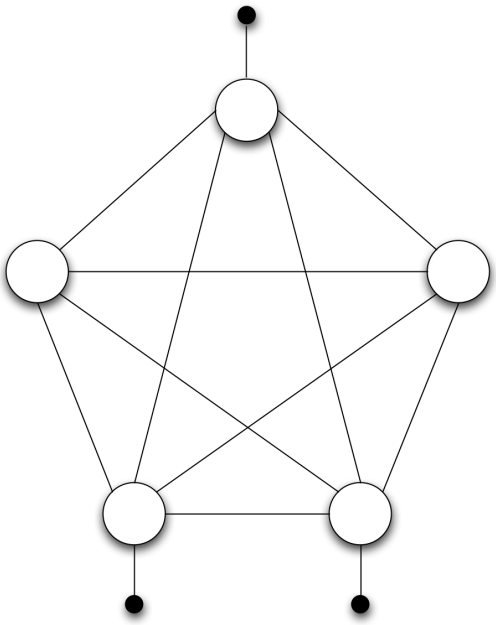


Figura 1.2. Estructura de una red recurrente

la red. Dichos pesos representan la fuerza de la conexión entre dos neuronas. El aprendizaje puede ser de dos tipos: aprendizaje supervisado o aprendizaje no supervisado.

En el aprendizaje supervisado, a la red se le suministra tanto un patrón de entrada como la salida que se desea, de forma que se le indica cuando actúa de forma correcta. Tras el entrenamiento, la red debe ser capaz de suministrar una salida correcta para un conjunto de entradas. Este tipo de entrenamiento se emplea para labores de clasificación de patrones y aproximación o predicción de funciones.

En el aprendizaje no supervisado, a la red únicamente se le suministran los patrones de entrada, de forma que la red no es capaz de conocer si actúa correctamente. El propósito de este tipo de aprendizaje es que la red sea capaz de deducir categorías o características de los patrones de entrada. Los campos principales de aplicación son la autoorganización de patrones y el descubrimiento de rasgos.

Por otro lado aparece la fase de presentación, en la cual, tras entrenar la red, ésta se utiliza para la aplicación deseada. En ella no se modifican de ninguna manera los pesos

1.3. Objetivos del proyecto

Como se ha comentado, las redes neuronales son sistemas altamente paralelizables, ya que están formadas por una red de neuronas, que realizan sus cálculos de manera independiente. Por tanto su ejecución como un algoritmo *software* en un microprocesador de propósito general es lenta, debido a la falta de aprovechamiento del inherente

paralelismo. Lo más parecido al paralelismo en máquinas de propósito general es el multihilo, que realmente es un paralelismo ficticio, o los sistemas multiprocesadores. Actualmente los sistemas multiprocesadores de propósito general tienen un número muy reducido de procesadores, por lo que ésta tampoco es una aproximación efectiva.

Una lógica implementación paralela del algoritmo sería su desarrollo en *hardware*, ya que cada neurona puede funcionar de manera independiente al resto. Esto, además puede descargar al procesador principal de los cálculos de la red neuronal, si se utiliza como coprocesador, lo que redunda en una mejora del rendimiento del sistema.

Por tanto, los objetivos de este Proyecto, a grandes rasgos son el diseño de una arquitectura hardware genérica para tipos comunes de ANNs, y el desarrollo de herramientas de generación automática de redes específicas utilizando esta arquitectura.

Estos objetivos requieren el desarrollo de un conjunto de objetivos parciales, detallados a continuación.

- En primer lugar se debe realizar un estudio de distintos algoritmos de redes neuronales, con el objetivo de decidir cuál es el más idóneo para su implementación como acelerador hardware. Finalmente se optó por que el diseño final de la arquitectura hardware se basase en las redes multicapa, más concretamente en el tipo Perceptrón Multicapa, debido a su extenso uso y su simplicidad, tanto conceptual como de implementación.

- Tras la realización del diseño de la arquitectura, se debe de desarrollar una metodología para la creación de ANNs con la arquitectura diseñada. Este paso es necesario para el desarrollo de una herramienta que permita la generación automática de ANNs autónomas, mediante la implementación de dicha metodología. Esta herramienta tratará de mostrar la validez tanto de la arquitectura diseñada, como de la estrategia de la automatización.

- Finamente, se procede a la integración de la arquitectura de ANN en un sistema mayor, de forma que se pueda emplear la ANN diseñada como periférico de un procesador de propósito general, mediante el uso de las estrategias de automatización desarrolladas, dado que esta implementación permite una mayor flexibilidad y potencia.

Esta Memoria está estructurada de manera que profundiza en estos objetivos parciales de forma individual. Así, tanto la arquitectura de la red, como de los componentes que la forman se estudian en el capítulo 2. En el capítulo 3 se profundiza sobre el desarrollo de una herramienta capaz de automatizar la generación de ANNs autónomas para aplicaciones específicas, denominada HANNA (*Hardware ANN Architect*). En el capítulo 4 se desarrolla la implementación de la ANN como periférico, junto a todos los detalles necesarios para la implementación en un sistema empotrado. En el capítulo 5 se realiza una implementación de una aplicación específica empleando las dos aproximaciones realizadas, tanto de la red autónoma generada por HANNA, como la red desarrollada como periférico. Finalmente, esta Memoria concluye con un capítulo de conclusiones, y posibilidades de ampliación y futuros trabajos derivados.

Capítulo 2

La red neuronal implementada

En este capítulo nos vamos a centrar en la explicación, elección y justificación del tipo de red neuronal que se va a implementar. Se abordarán detalles concretos de diseño, así como la adaptación del algoritmo de ejecución de red para adecuarlo a una implementación hardware.

2.1. Notación empleada

En primer lugar se va a explicar la notación empleada en adelante para la descripción de distintos algoritmos.

La capa se indica con el superíndice, mientras que la neurona se indica como subíndice. Así, la salida de las neurona j-ésima de la capa i-ésima se indica como o_j^i. En el caso de los pesos, el segundo subíndice indica el peso dentro de la neurona, de forma que el peso k-ésimo de la neurona j-ésima en la capa i-ésima se muestra como w_{jk}^i.

Esta notación también se extiende para todos los aspectos de los distintos algoritmos de redes neuronales, como funciones de activación.

En general y salvo que se diga lo contrario, la primera capa será la que tiene como entradas de sus neuronas las entradas de la red, y la última capa la que su salida corresponde con la de la red.

Como variables, se tomará w como la encargada de representar los pesos, o se referirá a la salida de las diferentes neuronas, y f simbolizará la función de activación de las distintas neuronas.

2.2. Algoritmos para la implementación de redes neuronales tipo Perceptrón

Como ya se explicó, las redes neuronales son sistemas complejos formado por la interconexión de unidades más simples, llamadas neuronas. Dichas neuronas calculan la suma ponderada de sus entradas, esto es, cada entrada se multiplica por un valor, llamado peso, donde cada peso indica la fuerza de la sinapsis entre neuronas. Además, a esta suma se suele añadir un valor constante, o *bias*, que proporciona un umbral

para la función de activación que se aplicará al resultado de la suma. Esta función generalmente es de carácter no lineal.

En una neurona, tanto los pesos como la *bias* son parámetros ajustables, que normalmente se obtienen mediante una regla de aprendizaje.

Dentro de de esta arquitectura de redes neuronales, podemos distinguir tres grandes familias:

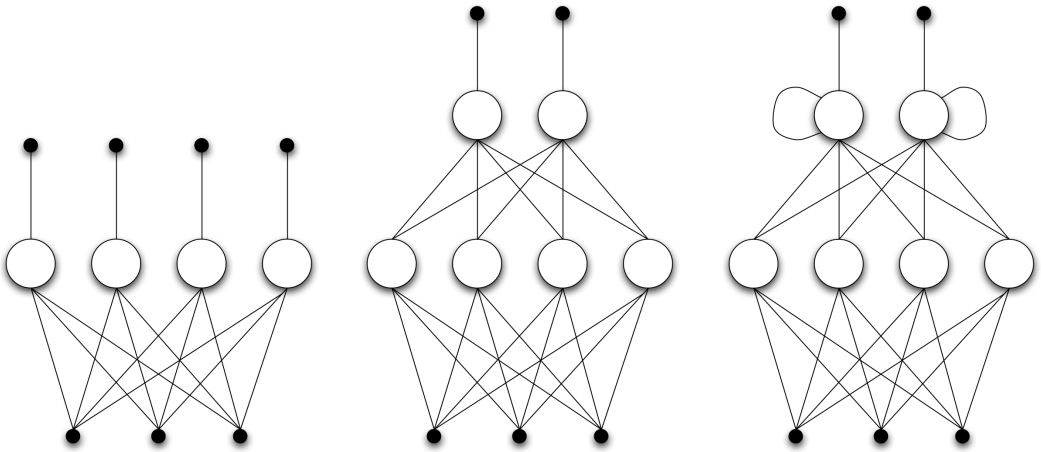
- **Redes de una sola capa de neuronas:** únicamente son capaces de resolver problemas linealmente separables. Son las primeras redes neuronales que se diseñaron, sobre las que recaen todas las críticas indicadas en el trabajo de Minsky y Papert. Dentro de esto sistemas podemos destacar el Perceptrón original diseñado por Rosenblatt, formado por neuronas cuya función de activación es el signo, o el ADALINE cuya función de transferencia es lineal. En la figura 2.2(a) se puede ver un ejemplo de este tipo de redes.

- **Redes con varias capas de neuronas:** en estas redes, las salidas de las neuronas de una capa son las entradas de las neuronas de la siguiente capa. Esta estructura es la más habitual en sistemas con aplicaciones reales, y tienen el aspecto que se muestra en la figura 2.2(b). Entre estas estructuras podemos destacar el Perceptrón Multicapa, sin lugar a dudas el sistema más utilizado en la actualidad.

- **Redes recurrentes:** realmente se podrían encuadrar en las familias anteriores, pero por su especial topología las vamos a separar. Son redes neuronales cuya característica novedosa es la retroalimentación, como se puede observar en la figura 2.2(c), lo que permite que sean sistemas teóricamente más potentes que las anteriores, y les permite mostrar comportamiento temporal. Como ejemplo podemos nombrar las redes Hopfield.

Cada una de estos tipos de redes no proporcionan el método para el cálculo de los pesos de la red, por lo que es necesario explicar las reglas más importantes de aprendizaje de redes neuronales. Para las redes monocapa, básicamente se utilizan técnicas de descenso de gradiente, como LMS o similares.

Para redes multicapa se suele emplear el algoritmo Backpropagation, o regla delta generalizada, el cual es una extensión de la regla de aprendizaje de redes monocapa. Este algoritmo, junto a todas sus variantes, es la regla fundamental para el aprendizaje de redes del tipo Perceptrón Multicapa. Es un algoritmo de aprendizaje supervisado, de forma que es necesario para el entrenamiento aportar, para cada patrón de entrenamiento, tanto la entrada de la red como la salida que se desea que calcule dicho patrón. El objetivo del algoritmo es ir propagando el coste producido debido al error cometido por la red, definiendo el error como la divergencia del valor calculado por la red respecto al valor que se desea que la red produzca, para todas las neuronas de la red, no únicamente a las neuronas de la última capa. El coste se define mediante una función de coste, la cual es elegida por el usuario, con la única restricción de que sea derivable respecto a la salida de la red. Este algoritmo *no* garantiza la convergencia, y el aprendizaje puede ser bastante lento.



(a) Red de una sola capa **(b)** Red de varias capas **(c)** Red recurrente

Figura 2.1. Ejemplos de las distintas arquitecturas de redes

Las fórmulas que describen el Backpropagation tienen la siguiente forma:

$$\frac{\partial C}{\partial w_{jk}^i} = \sum_{m=0}^{N_{i-1}} \left(\Delta_m^{i-1} w_{mj}^{i-1}\right) \frac{\partial f_j^i}{\partial y} \left(\sum_{m=0}^{N_{i-1}} o_m^{i+1} w_{jm}^i\right) o_k^{i+1} = \Delta_j^i \; o_k^{i+1} \text{ para } i > 1$$

$$\frac{\partial C}{\partial w_{ij}^1} = \frac{\partial C}{\partial o_i^1} \frac{\partial f_i^1}{\partial y} \left(\sum_{j=0}^{N_0} o_j^2 w_{ij}^1\right) o_j^2 = \Delta_i^1 o_j^2$$

N_i indica el número de neuronas de la capa i-ésima, y C es la función de coste definida por el usuario. En los sumatorios, la iteración referida a la entrada 0 indica la bias.

Cabe indicar que en este caso la numeración de las capas es al revés, por lo que N_1 es el número de neuronas en la capa de salida, tal y como se muestra en la figura 2.2. Esta nomenclatura facilita la demostración, si bien, como ya se ha comentado, no es la norma en esta Memoria.

Han surgido multitud de modificaciones del Backpropagation, pero esencialmente el algoritmo es idéntico, con variaciones como el empleo del gradiente conjugado, parámetro de paso de tamaño variable, etc.

En esta Memoria nos decantaremos por el uso del Perceptrón Multicapa debido a que es el más empleado en la actualidad, el más estudiado y su modelo matemático tiene una estructura lo suficientemente simple como para hacer atractiva su implementación sobre hardware específico. Como función de activación, en primera instancia, se utilizará la tangente hiperbólica.

Para una explicación detallada de los distintos algoritmos de redes neuronales y de algoritmos de aprendizaje, se puede consultar [2], [3], [7] y [8].

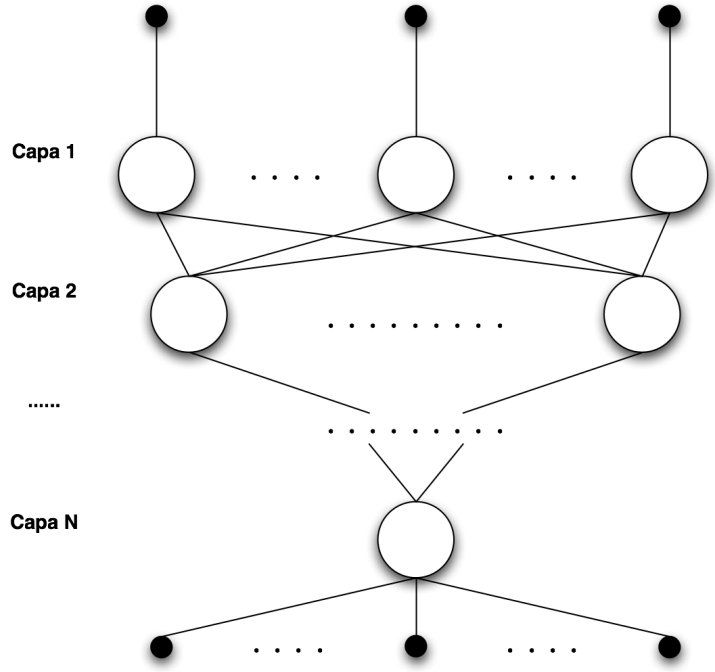


Figura 2.2. Esquema de la red con la nomenclatura empleada para el Backpropagation

2.3. Arquitectura hardware del Perceptrón Multicapa

La arquitectura de la red será diseñada para optimizar su implementación en un circuito de propósito específico. Para la implementación del sistema se utilizará un lenguaje de descripción hardware, como VHDL.

La otra alternativa principal de metodología de diseño sería el diseño basado en esquemáticos, el cual consiste en modelar el sistema mediante la interconexión de componentes. Estos componentes se ven, desde el punto de vista funcional, como una caja negra con unas determinadas entradas y salidas, abstrayéndonos de su funcionamiento interno. Dichos componentes pueden ser los componentes básicos suministrados por la herramienta o pueden ser creados como la interconexión de otros componentes.

A partir de la creación de componentes y su interconexión, que darán lugar a la generación de nuevos componentes de mayor nivel en la estructura jerárquica del diseño, se puede ir abstrayendo el diseño completo del sistema, por lo que se puede afirmar que esta metodología impone una estrategia de diseño ascendente.

La principal ventaja de los diagramas de esquemas es la facilidad para generar el circuito, ya que es una plasmación directa del esquema del sistema, mediante la interconexión de cada uno de los componentes que lo forman. Además, su comprensión y análisis es muy rápida y sencilla, más si cabe en sistemas modulares, como las redes neuronales, formados principalmente por la interconexión de componentes similares — las neuronas —. Sin embargo, esta metodología imposibilita, de forma simple, la

generalización en el diseño de componentes y de la arquitectura.

Este inconveniente es prácticamente insalvable, ya que requeriría que cada posible neurona, vista como componente, fuera realizada previamente. Además, cada nueva red debe ser implementada de nuevo, añadiendo las neuronas y conectándolas. Por tanto, se puede concluir que este acercamiento no es viable.

El diseño basado en lenguajes de descripción hardware describe el circuito proyectado mediante la especificación funcional del comportamiento del sistema. Esta especificación es traducida por la herramienta de síntesis en una serie de ecuaciones que se disponen sobre los distintos componentes básicos definidos para la tecnología electrónica en la que se esté trabajando. Esta forma de especificar el diseño permite un nivel de abstracción mayor, ya que definimos su funcionamiento, no su implementación; no obstante, también es posible inspeccionar, e incluso modificar, a bajo nivel las construcciones en las que se sintetizará el diseño.

Además, estos lenguajes poseen recursos para la generalización y reutilización de los diseños, ya que permiten el uso de parámetros, de forma que el comportamiento final de un módulo determinado dependerá de los valores que se le asignen a dichos parámetros. Esto permite desarrollar una arquitectura completamente genérica, cuyas implementaciones particulares se obtienen mediante el uso de tales parámetros.

Dentro de los lenguajes de descripción hardware se ha decidido utilizar VHDL, por ser uno de los de mayor aceptación y el más estandarizado. VHDL fue patrocinado en primera instancia por el Departamento de Defensa de los Estados Unidos en 1982, con el fin de diseñar una herramienta estándar e independiente para el modelado, documentación y simulación de los sistemas electrónicos digitales, placas de circuito y componentes durante todas las fases de diseño. La primera revisión fue desarrollada por IBM y Texas Instruments en 1985. Posteriormente, la IEEE Computer Society decidió adoptarlo y estandarizarlo al ver su enorme potencial. Desde entonces, se han desarrollado por este organismo tres revisiones del lenguaje, en 1987, 1993 y 2001.

La arquitectura básica del modelo hardware de la red será la que se puede esperar de un Perceptrón Multicapa: una interconexión de neuronas, con la única diferencia de que en este caso además se hace necesario una unidad de control para la gestión de las neuronas, tal y como se muestra en la figura 2.3. Ambos bloques se estudian en profundidad a continuación. Los detalles de su implementación se darán en capítulos posteriores, donde se desarrollan aplicaciones para la automatización de la construcción de este tipo de redes. En cuanto a su rendimiento, en las secciones 5.1 y 5.2 se incluyen medidas concretas de diferentes redes aplicadas a ejemplos prácticos, e implementadas sobre FPGAs reales.

2.3.1. La neurona

La neurona realiza los cálculos habituales para un Perceptrón Multicapa: la suma compensada con los pesos de las distintas entradas, a las que se le suma el bias y se le aplica una función de activación.

Las entradas se introducen todas juntas en forma de un único bus, incluida una entrada que será empleada como bias. Posteriormente, cada una de las entradas se

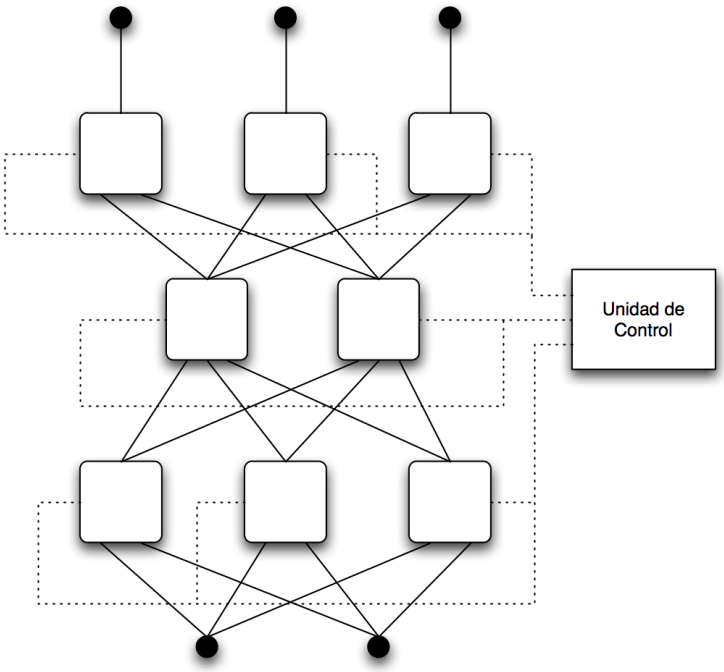


Figura 2.3. Arquitectura de la red hardware

extraerá de forma interna mediante el uso de un multiplexor. En principio, como convención se tomará que el bit más significativo del bus es el bit más significativo del bias, que será la entrada que se extraiga del multiplexor al asignar el valor lógico 0 a las señales de control de dicho multiplexor. Los pesos se introducen como una entrada adicional — no se encuentran predefinidos como constantes en el interior de la neurona — de la misma manera que las entradas, es decir, en forma de bus.

Hay una entrada para el bias tanto en los pesos como en las entradas. Por tanto, para calcular el bias, se realiza el producto de ambas entradas. Lo normal es especificar la entrada con el símbolo que represente un *1* e introducir el valor del bias como el peso. No obstante cualquier cálculo que dé como resultado el mismo bias es válido.

Esta estrategia para codificar entradas y pesos en dos únicos buses se justifica porque esta estructura simplifica su implementación electrónica, ya que el número de entradas no depende de la neurona, sólo varía la anchura de esas entradas, parámetro fácilmente parametrizable en lenguajes de descripción hardware.

La implementación de la suma se realiza mediante un único multiplicador y un acumulador, tal y como se observa en la figura 2.4. La selección se hace mediante un multiplexor en las entradas y otro en los pesos, controlados por las mismas señales de control, de forma que a la salida de ambos multiplexores se obtenga el peso y la entrada correspondiente. Estas señales de control se introducen en la neurona por un puerto, siendo éstas generadas en la unidad de control, que se encargan de gobernar la ejecución sincronizada de la red.

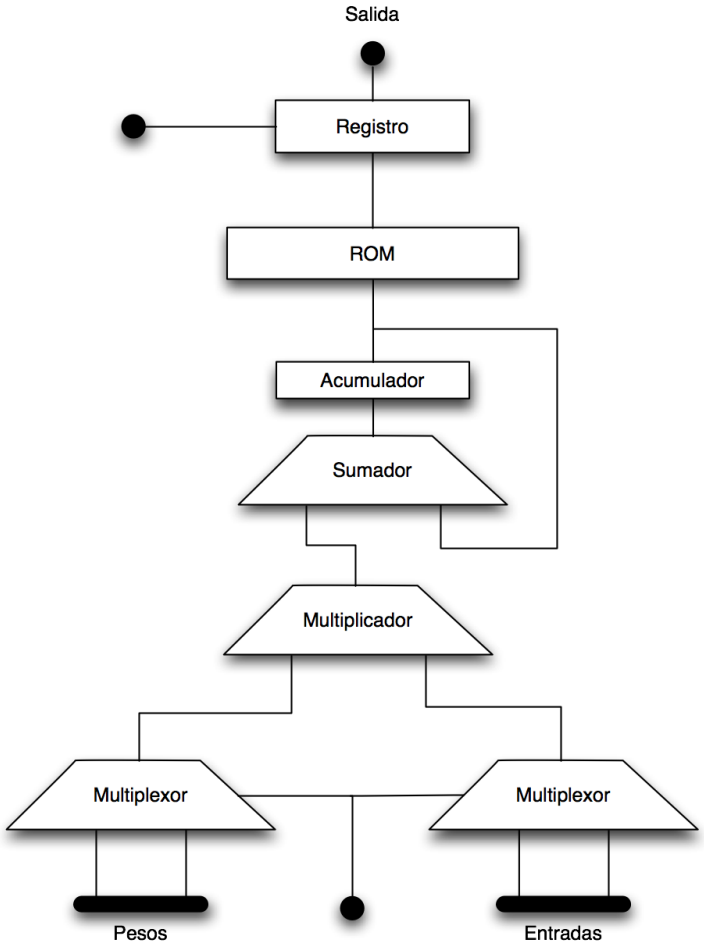


Figura 2.4. Arquitectura de la neurona diseñada

A continuación ambas señales entran al multiplicador, cuya salida se conecta al acumulador. La salida del acumulador se introduce en la función de activación, que generalmente estará implementada en forma de ROM. Además, la salida de las neuronas tendrá una señal de habilitación, controlada por las señales de control, de forma que mantenga la misma salida hasta que la red termine de calcular la nueva iteración, asegurando de este modo que a su salida siempre haya un resultado correcto. El esquema de la arquitectura, como se ha indicado, se muestra en la figura 2.4. En ella se observan lineas que comienzan o terminan en un punto o linea gruesa negra, que indica que es una entrada o salida de la neurona. Comentar además que se han obviado las señales de reloj y reset por simplicidad.

Esta forma de implementar la neurona, con un único multiplicador, se debe a que la implementación completamente paralela implicaría un multiplicador por cada entrada de la neurona, lo que dispararía la necesidad de recursos de la red.

Teniendo en cuenta lo anterior, se deduce que cada capa genera una salida correcta

cada $N_i + 1$ pulsos de reloj, donde N_i es el número de entradas, incluido el bias, de las neuronas de esa capa. El pulso adicional es necesario para la lectura de la ROM que almacena la función de activación

Por tanto, la latencia, o número de ciclos necesarios para obtener la salida de la red neuronal, en el caso general de N capas, será de

$$L = \sum_{i=1}^{N} (N_i + 1) = \sum_{i=1}^{N} (N_i) + N \text{ ciclos de reloj.} \tag{2.1}$$

2.3.2. La unidad de control

La unidad de control se encarga de enviar las señales de control a las neuronas de las distintas capas. Estas señales controlarán los multiplexores de entradas y pesos, y la señal de habilitación de la salida de las neuronas.

El procedimiento seguido consiste en primer lugar en el envío de las señales de control que controlan los multiplexores, de forma que vayan recorriendo las entradas y los pesos de las neuronas de una capa, y en segundo lugar, tras terminar de recorrer las entradas y los pesos, se envía el bit de activación de la salida de dichas neuronas, por lo que en el siguiente ciclo se pueden empezar los cálculos de la siguiente capa. Como se ha indicado, las señales de control serán las mismas para las neuronas de una misma capa, ya que todas tienen el mismo número de entradas, y sus cálculos se pueden realizar en paralelo.

El número de señales de control dependerá del número de capas de la red neuronal, de forma que al aumentar el número de capas, aumenta el número de señales de control necesarias, ya que neuronas de diferentes capas no pueden compartir señales de control.

Desde el punto de vista de su función, para una capa determinada, las señales de control se dividen en dos: por un lado están las que se encargan de controlar los multiplexores — cuya longitud varía de una capa a otra, dependiendo del número de entradas de esa capa —, y por el otro se encuentra la habilitación de la salida de las neuronas de esa capa — que siempre es de un bit —. Las señales de control que gobiernan los multiplexores tienen, para cada capa, la longitud de líneas mínima para direccionar las entradas de las neuronas de una capa. Así, si una capa determinada tiene 4 entradas, más el bias — esto es, 5 entradas —, necesita 3 bits para direccionar todas las entradas.

Un ejemplo de la salida de la unidad de control en función del tiempo, para una red de dos capas, con 5 entradas, y 3 neuronas en la primera capa— las neuronas de la capa de salida no influyen —, será la siguiente:

```
000  0   00  0
001  0   00  0
010  0   00  0
      ...
101  0   00  0
101  1   00  0
```

101 1 01 0
...
101 1 11 0
101 1 11 1

Notar que cada línea corresponde con un ciclo de reloj. Los dos primeros bloques se encargan de el control de la primera capa, mientras los dos últimos se encargan de la segunda capa. El primer bloque de números controla el multiplexor de las neuronas de la primera capa. Se observa que va de 0 a 5, mientras se va incrementando en cada palabra de la unidad de control, hasta llegar al número de entradas, donde se incluye el bias. Tras finalizar, el siguiente campo conmuta a 1; este campo controla la salida de las neuronas de esa capa.

Al terminar la primera capa sus cálculos, empieza a variar el tercer campo, encargado del control del multiplexor de las neuronas de la capa de salida. El proceso para esta capa es idéntico al de la primera capa. Al finalizar el recorrido, el proceso comienza de nuevo.

En principio, la unidad de control para este algoritmo se implementará como una ROM que contiene las salidas de la unidad de control, junto a un acumulador que irá incrementándose para ir recorriendo toda la ROM. La ROM debe contener tantas palabras como ciclos de reloj necesite la red para generar una salida válida, cuyo número se puede calcular en la ecuación 2.1. Cada palabra tendrá, por tanto, una anchura de

$$d = \sum_{i=1}^{N} (\lceil \log_2(N_i) \rceil + 1) = \sum_{i=1}^{N} (\lceil \log_2(N_i) \rceil) + N \text{ bits.}$$

Como se puede observar, la unidad de control es muy simple, ya que el funcionamiento de las neuronas es muy regular, no hay estados ni excepciones, toda la ejecución es secuencial y repetitiva.

Capítulo 3

Herramienta software para el prototipado rápido de redes neuronales

HANNA — *Hardware Artificial Neural Network Architect* — es una aplicación para la generación de descripciones de redes neuronales optimizadas para su implementación hardware, a partir de modelos algorítmicos de dichas redes ya diseñadas y entrenadas, de forma que puedan crearse un prototipo de la red sobre hardware específico sin requerir conocimientos avanzados de sistemas electrónicos digitales y arquitectura de computadoras.

La proyección sobre microprocesadores especialmente diseñados tiene generalmente como objetivo el incremento de velocidad de cómputo, aunque no deben olvidarse la reducción de tamaño, consumo o costo del dispositivo, factores determinantes en algunas aplicaciones, como robótica, juguetería o control empotrado.

De esta forma, una vez determinado y ajustado un modelo algorítmico adecuado para la resolución de un problema concreto, es posible crear con poco esfuerzo adicional un circuito de aplicación específica que implementa dicho modelo, pudiéndose beneficiar del consiguiente incremento en la velocidad de procesamiento de la red. Por tanto, HANNA no es una herramienta pensada para el diseño integral de redes neuronales. No obstante, sí puede ser empleada para realizar una exploración del espacio de diseño de las redes al ser proyectadas en hardware, para el estudio de sus prestaciones al modificar determinados parámetros, como número de bits, la función de activación, etc.

Por otro lado, la complejidad del diseño a nivel de transferencia de registros haría inviable, por costosa, la implementación desde cero de un número suficiente de arquitecturas similares, hasta determinar los parámetros óptimos para la síntesis del modelo hardware de la red.

HANNA está implementado sobre Matlab, de forma que las redes neuronales generadas por la herramienta son subsistemas de Simulink, lo que permite su utilización en el flujo de trabajo habitual de esta popular herramienta, pudiendo comparar sus prestaciones con implementaciones de la red de utilizando distintas aproximaciones (software genérico, microcontroladores, procesadores digitales de señales, y FPGAs).

En la figura 3.2 se muestra un ejemplo de un sistema completo que nos permite ilustrar las ventajas de este flujo de diseño. Se trata de un Perceptrón con dos en-

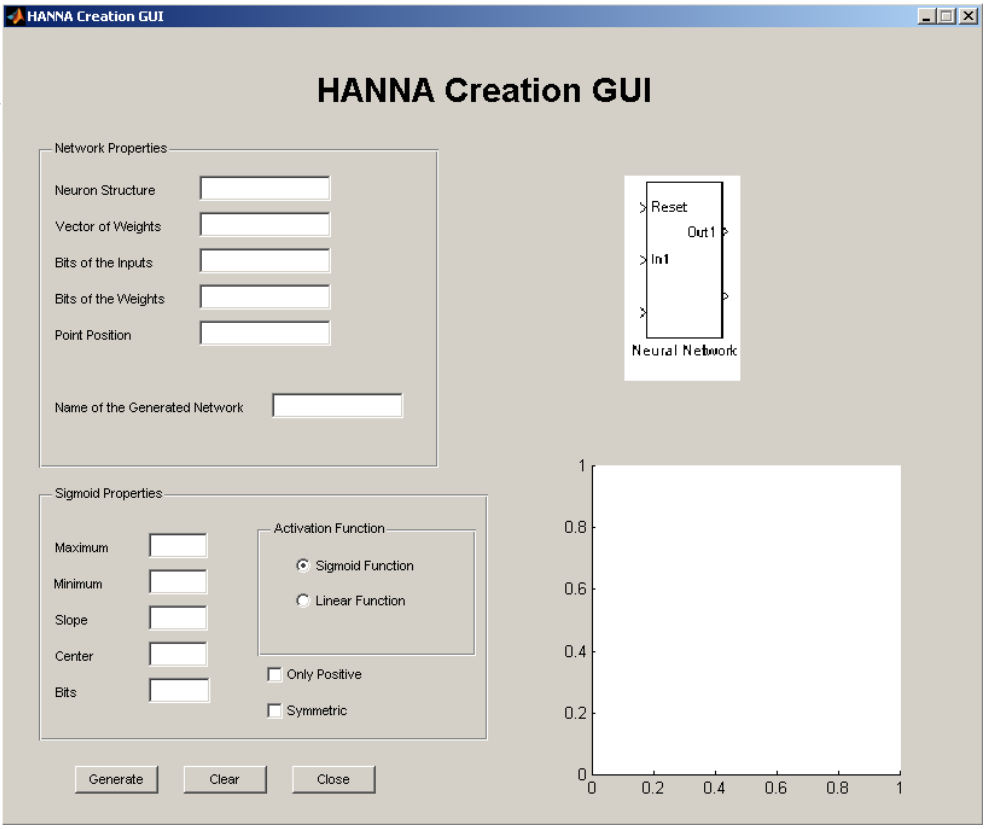


Figura 3.1. Aspecto de la GUI de HANNA

tradas, dos neuronas en la capa oculta y una en la capa de salida, cuya objetivo es modelar la función lógica XOR. Para comparar prestaciones y resultados, esta función se implementa de tres formas distintas, como se aprecia en la figura: mediante el propio operador XOR — verde —, mediante un Perceptrón software tradicional — azul — y mediante un Perceptrón hardware — naranja —. Como se puede observar, se incluyen los bloques necesarios para generar los estímulos para las entradas y para la monitorización de las salidas. Los bloques *Gateway In* y *Gateway Out* delimitan el espacio software del hardware dentro del sistema. Al estar integrados dentro de Simulink, es posible realizar una simulación conjunta de los tres sistemas y observar en tiempo real la evolución y precisión de las salidas en cada caso.

El objetivo de HANNA es la creación de una herramienta que sea un paso intermedio al objetivo final, demostrar que es posible automatizar la generación de redes neuronales con la arquitectura diseñada.

3.1. Arquitectura de la aplicación

Como ya se ha indicado, HANNA está implementado como un *script* en Matlab con una interfaz gráfica independiente, lo que permite unificar facilidad de uso, por el em-

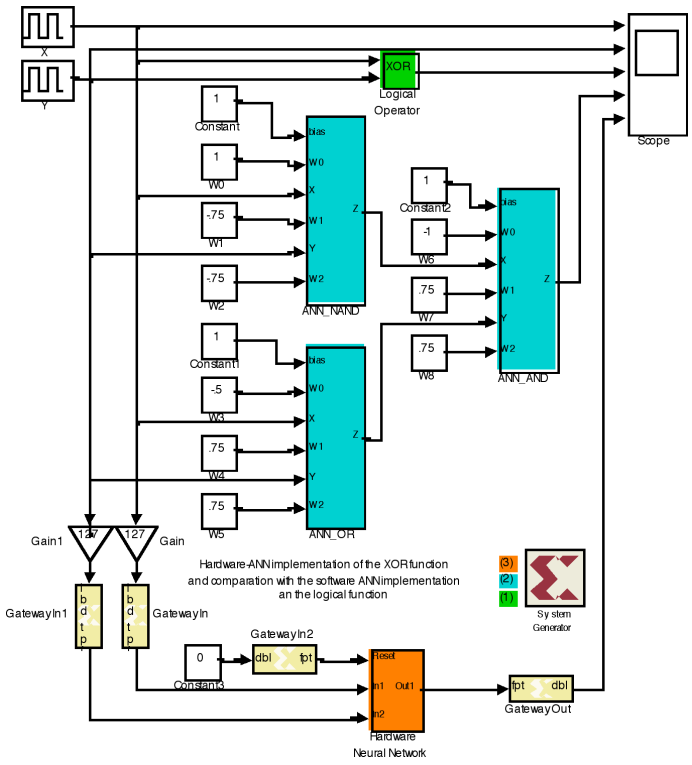


Figura 3.2. Ejemplo de un sistema implementado de tres formas diferentes

pleo de la interfaz, con la potencia del *script*, ya que su utilización permite automatizar la generación de redes, o el paso de parámetros complejos.

HANNA requiere para su funcionamiento de *Xilinx System Generator*, una herramienta para el modelado a nivel de sistemas que facilita el diseño en FPGAs. Para ello System Generator extiende Simulink de forma que proporcione un entorno apropiado para el diseño hardware, incluyendo además tanto un conjunto de bloques que representan sistemas de alto nivel que pueden ser compilados directamente en FPGA, como un conjunto de bloques de bajo nivel, que permite acceso a los recursos de la FPGA para obtener sistemas muy optimizados.

Una de las ventajas incorporadas es una interfaz de *co-simulación hardware*, que permite, al simular el diseño en Simulink, ejecutar la parte implementada en System Generator en hardware, lo que permite aumentar drásticamente la velocidad de la simulación.

System Generator también provee de interfaces para importar módulos implementados en lenguajes de descripción hardware, como VHDL o Verilog, a diseños de System Generator, lo que permite extender la funcionalidad de los bloques suministrados con los implementados por el usuario. Además estos bloques son simulados de forma transparente al usuario, y permite la utilización de la interfaz de co-simulación también de forma transparente.

La compilación de diseños en System Generator permite generar distintas imple-

mentaciones, como la generación de código de descripción hardware o la exportación para otras herramientas de Xilinx, como Xilinx ISE o Xilinx EDK.

Los bloques suministrados por la herramienta se dividen en varias librerías:

- Bloques destinados a las comunicaciones, que engloban bloques usados comúnmente en sistemas de comunicaciones digitales, como moduladores.
- Bloques utilizados como lógica de control, como máquinas de estados.
- Bloques que se encargan de realizar conversiones entre datos, entre los que se encuentran los *Gateway*, bloques que conectan la parte de bloques estándar de Simulink con los bloques de Xilinx.
- Bloques para el procesado digital de señal.
- Bloques que calculan funciones matemáticas.
- Bloques que implementan memorias.
- Bloques que permiten el acceso a los objetos de Xilinx con memoria compartida.
- Bloques que sirven de utilidades del diseño, como el bloque encargado de generar el código que describe el circuito correspondiente, o el bloque encargado de la co-simulación.

La estructura de nuestra aplicación, como se ha comentado, se divide en dos partes fundamentales, el *script*, parte principal de la aplicación y la interfaz gráfica, mero traductor para el *script*.

El *script* es el corazón de la herramienta, encargada de la generación de la red y todos los ficheros necesarios para su funcionamiento. La herramienta devuelve un fichero de Simulink con la red generada lista para ser conectada a un banco de pruebas diseñado por el usuario, así como los bloques necesarios para su correcta simulación y su posterior síntesis sobre una FPGA. Además, se genera un fichero llamado *nombreDeLaRed*Vars.mat, donde *nombreDeLaRed* es el nombre de la red que se genera, que contiene las variables de entorno necesarias para el funcionamiento de la red en Simulink.

La herramienta requiere para el correcto funcionamiento del *script* los siguientes archivos: HANNA.m, Control_config.m, neuron_wcte_config.m y Plantilla.mdl. HANNA.m es el fichero del *script*, mientras que la utilidad de todos los demás se explicará más adelante, al explicar el funcionamiento interno de la herramienta.

La cabecera de la función tiene la forma siguiente:

```
1 function HANNA(neurons, weights, bits, wbits, sigbits, point, nameNet, a,
      c, maxi, mini, positiva, lineal, simetrica)
```

Cada uno de estos parámetros tiene la siguiente función:

- `neurons` especifica la estructura de la red neuronal, en la forma de un vector, en el que el primer valor indica el número de entradas de la red. Así `[5 4 6]` indica que la red tiene 5 entradas, con 4 neuronas en la primera capa, y 6 neuronas en la capa de salida.
- `weights` contiene los pesos de la red neuronal en forma de array tridimensional, en el que la primera dimensión indica la capa, la segunda dimensión la neurona y la tercera indica el peso. De esta forma, `peso(x,y,z)` hace referencia al peso z de la neurona y de la capa x.
- `bits` indica el número de bits utilizados para la cuantización de las entradas y salidas de la red (generalmente tomando valores entre 8 y 16 bits).
- `wbits` indica el número de bits con los que se representan los pesos. Los valores adecuados usualmente también oscilan entre 8 y 16 bits.
- `sigbits` especifica el número de bits empleados para discretizar la función de activación, lo que determina el tamaño de la memoria ROM que se utilizará para su implementación. Para muchas aplicaciones, sus valores se encuentran entre el intervalo [4, 12].
- `point` indica la posición del punto decimal. `0` especifica que no hay punto decimal.
- `nameNet` es una cadena de texto que indica el nombre con el que se guardará la red.
- `a` es el parámentro con el que se indica la pendiente de la función de activación, en caso de emplear la tangente hiperbólica.
- `c` ídem para el centro.
- `maxi` establece el máximo de la función de activación que se mapeará.
- `mini` ídem para el mínimo.
- `positiva` indica con `1` si se desea que la función de activación sea positiva, convirtiéndose en una sigmoide, o `0` si se desea positiva y negativa.
- `lineal` indica si se desea una función de activación específica — `0` — o se desea una salida lineal — `1` —.
- `simetrica` especifica si se desea utilizar la mitad de la ROM únicamente, con la condición de que la función de activación sea simétrica. Esto se implementa eliminando la parte negativa de la ROM y mediante la conversión de las entradas negativas de la ROM a positivas, y a continuación realizando el complemento a 2 de la salida de la función.

La interfaz se diseñó pensando en facilitar la tarea de la creación de redes neuronales sin tener que introducir la larga lista de parámetros a mano, de un modo más visual y simple. La interfaz la componen los ficheros siguientes:

- HANNAGUI.fig: es la interfaz, generada con *GUIDE*.
- HANNAGUI.m: contiene las funciones de la interfaz.
- icono.bmp: imagen que contiene el icono que muestra la forma que tendrá la red neuronal.

El uso de la interfaz implica unas series de restricciones respecto al uso directo del *script*, consistentes en:

- La estructura de la red tiene que tener la forma de vector en Matlab, no pudiendo utilizar una variable, así una red de dos entradas, dos neuronas en la capa oculta y una en la capa de salida tiene que escrirse como `[2 2 1]`.
- El parámetro de los pesos ha de ser una variable, la cual ha de ser accesible desde el *workspace* de la interfaz. Para ello, el modo más simple es que la variable esté definida como *global*.
- El resto de parámetros numéricos han de ser números, o en el caso del nombre de la red una cadena de texto; no pueden utilizarse variables.

Estas limitaciones se impusieron por facilitar el desarrollo de la interfaz. Obviamente, el uso del *script* no presenta estas restricciones, si bien dichas condiciones no se consideran debilidades de la interfaz, ya que no restan funcionalidad a la herramienta. Además no suponen dificultar el uso de la aplicación, únicamente no permiten el uso de formas complejas en los parámetros, lo que no es muy apropiado en una interfaz, si bien es lógico su uso en el *script*.

El modelo de neurona empleado utiliza la estructura mostrada en el capítulo anterior, escrita totalmente en VHDL. Cada peso es implementado como constante de *System Generator*, mediante el uso del bloque *constant*. En el caso de la unidad de control, ésta se implementa con la estructura comentada en el capítulo anterior, también en VHDL. Ambos ficheros son generados por la herramienta con las opciones suministradas por el diseñador.

3.2. Funcionamiento interno

Solo se va a comentar cómo el *script* genera las redes neuronales, pero no se va a comentar nada sobre la interfaz gráfica, ya que como se ha dicho, su única función es llamar al *script*.

Las redes son generadas del mismo modo que si se hicieran a mano, es decir, se generan mediante la colocación de bloques y de su interconexión mediante las funciones que Matlab provee para este fin.

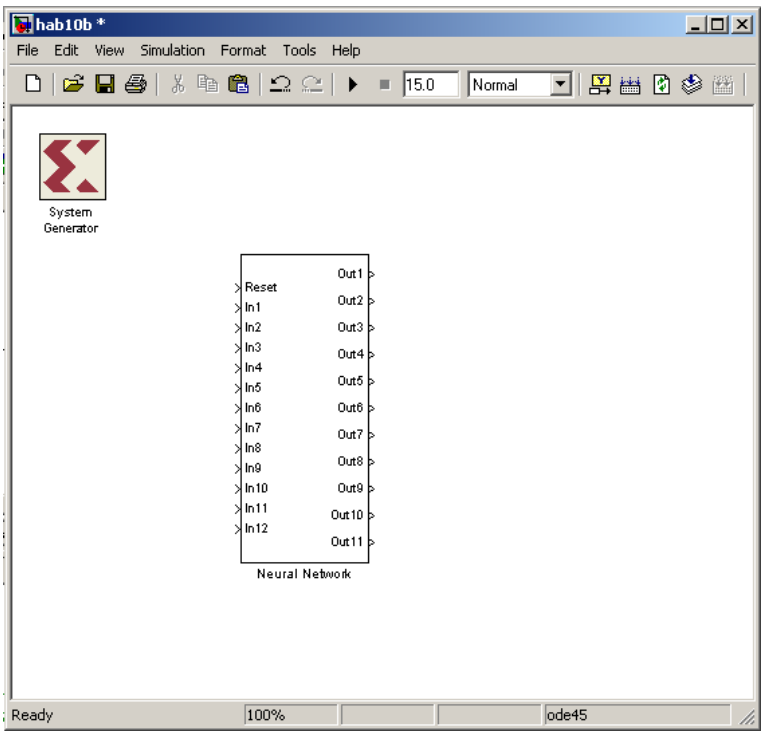


Figura 3.3. Red generada por HANNA

En la figura 3.3 se observa el aspecto de una red generada. El bloque Neural Network contiene la red propiamente dicha, mientras que el bloque denominado System Generator se utiliza para configurar los parámetros que habrán de utilizarse para la extracción del modelo hardware de la red, durante el proceso de síntesis del circuito equivalente.

La herramienta genera la red siguiendo los siguientes pasos:

- En primer lugar crea las variables globales necesarias por la aplicación, genera la ROM de la unidad de control y de la neurona según los parámetros introducidos, escribiendo a continuación el fichero VHDL tanto de la unidad de control como de la neurona con las ROMs generadas, llamados Control.vhd y neuron_wcte.vhd respectivamente.

  En el caso de utilizar como función de activación de la neurona la función lineal, no es necesario utilizar una ROM, simplemente se conectan los bits más significativos del acumulador a la salida de la neurona.

  Para construir la ROM de la función de activación, se calculan los valores de la función de activación en las posiciones que tiene dicha ROM entre el máximo y el mínimo definido por el usuario, para a continuación convertirlos a binario, y finalmente almacenar en disco el fichero de la neurona.

  Para la generación de la ROM de la unidad de control, se realiza un bucle que se

ejecuta tantas iteraciones como palabras tiene la ROM. En cada iteración se ejecuta un segundo bucle, que se encarga de recorrer tantas iteraciones como número de capas tiene la red. En cada iteración de este segundo bucle se van escribiendo los distintos bloques de señales de control que componen cada palabra, es decir, primero se escribe el bloque que escribe las señales de control de la primera capa, a continuación el bloque que contiene las señales de control de la segunda capa, y así sucesivamente. Cada bloque contiene dos subbloques, consistentes en un contador que se va incrementando y en la señal de activación de la salida de las neuronas de esa capa, tal y como se indicó al explicar la arquitectura de la unidad de control, en el apartado 2.3.2.

En las primeras iteraciones del primer bucle, el cual se encarga de recorrer las palabras de la ROM, el contador del primer bloque se va incrementando hasta llegar al número de entradas de las neuronas de esa primera capa, dejando el resto de señales a cero, para, a continuación, en la siguiente palabra, poner la señal de control de la salida de las neuronas a 1, tal y como se puede ver en el ejemplo del apartado 2.3.2. Tras ello, en la siguiente palabra empieza a incrementarse el contador de la capa siguiente, manteniendo el valor de las señales de las capas anteriores. Este proceso se va repitiendo en todos los bloques de señales de control.

Al finalizar cada iteración del primer bucle, se convierte la palabra generada a binario y se concatena a las ya generadas, hasta obtener finalmente la ROM completa.

- A continuación se procede a la generación de la red. En primer lugar, se crea el sistema en Simulink y se implementa el subsistema que corresponderá con la red neuronal, el cual se muestra en la figura 3.4.

- Tras ello, se empieza a generar todas los puntos de conexión con el exterior del subsistema, y a recorrer la primera capa de neuronas.

  Los pasos para desarrollar una neurona es crear un subsistema, que la encapsulará, para a continuación realizar un bucle que se ejecutará tantas veces como el número de entradas de la neurona más uno. En cada iteración de dicho bucle se coloca una interconexión con el subsistema de la red neuronal, que corresponderá con esa entrada, y una constante de *System Generator* correspondiente a su peso, al que se le asigna su valor correspondiente. Para el caso del bias, se colocará una constante como entrada, con el valor definido en `bias`. Todos estos bloques son puestos en el interior del subsistema que formará la neurona.

  Para concatenar los buses de las entradas y los pesos, se utilizan concatenadores, *concat*, de dos entradas, que se van poniendo en cascada, debido a que ello facilitaba la programación sin incrementar la lógica, ya que juntar señales en un bus no incrementa el tamaño del circuito.

  Tras colocar toda la lógica necesaria para las entradas y los pesos, se instancian las interconexiones de las señales de control y la salida. A continuación se instancia la neurona y se conectan todas sus entradas y la salida.

- Este último paso se repite con todas las capas hasta completar la red. En la iteración de la última capa, se generan las interconexiones de salida de la red y se conecta la salida de los subsistemas neuronas de la última capa a estas interconexiones. El resultado es el que aparece en la figura 3.5.
- Al terminar, se crean las entradas de la red neuronal, y se conectan a las neuronas de la primera capa.
- A continuación se coloca la unidad de control, y se divide su salida mediante el bloque *slice* en las distintas señales de control de las distintas capas.
- Como último paso relativo a la contrucción de la red, se interconectan las entradas de todos los subsistemas neuronas de las capas intermedias a las salidas de las anteriores, a la vez que se conectan las señales de control.
- Finalmente, se guarda el sistema generado, se copian las variables globales que son necesarias para la síntesis de la red en el fichero *NombreDeLaRed*Vars.mat, para que puedan ser cargadas si se limpia el *workspace*, y finalmente se reabre el sistema para poder empezar a trabajar con él.

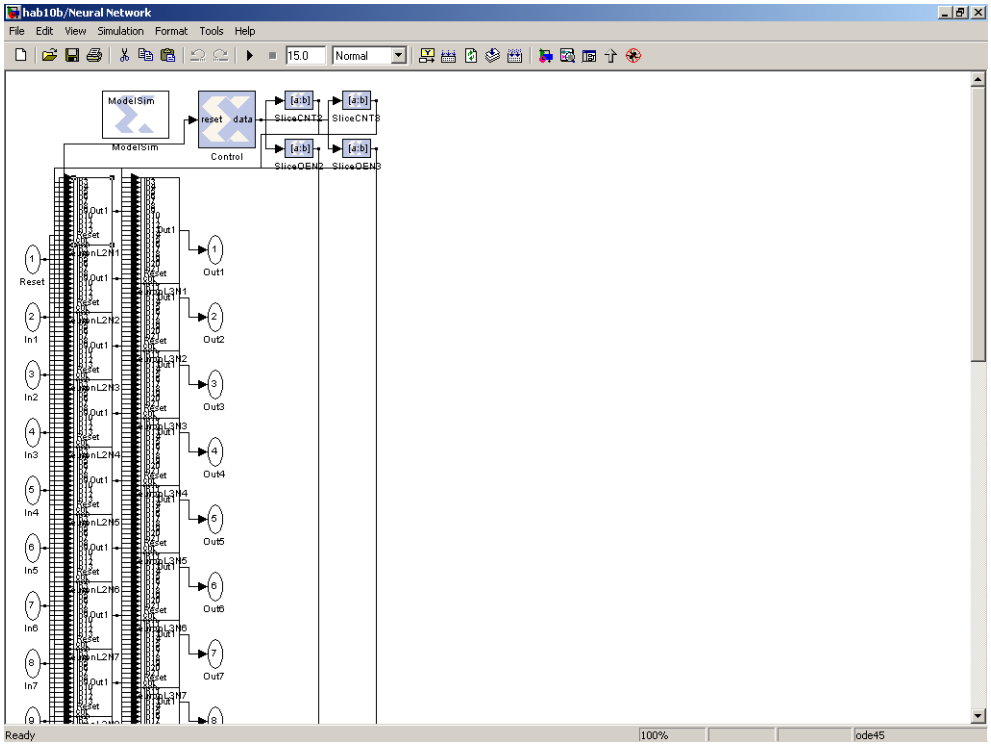


Figura 3.4. Aspecto del interior de la red neuronal generada

Cabe comentar que la neurona y la unidad de control son utilizados en la forma de *Black Box*, cuyos ficheros VHDL son generados por el *script*, como ya se comentó,

debido a que ambos tienen partes que dependen de la topología de cada red. En el caso de la neurona, la función de activación, y en el caso de la unidad de control, la ROM que la modela.

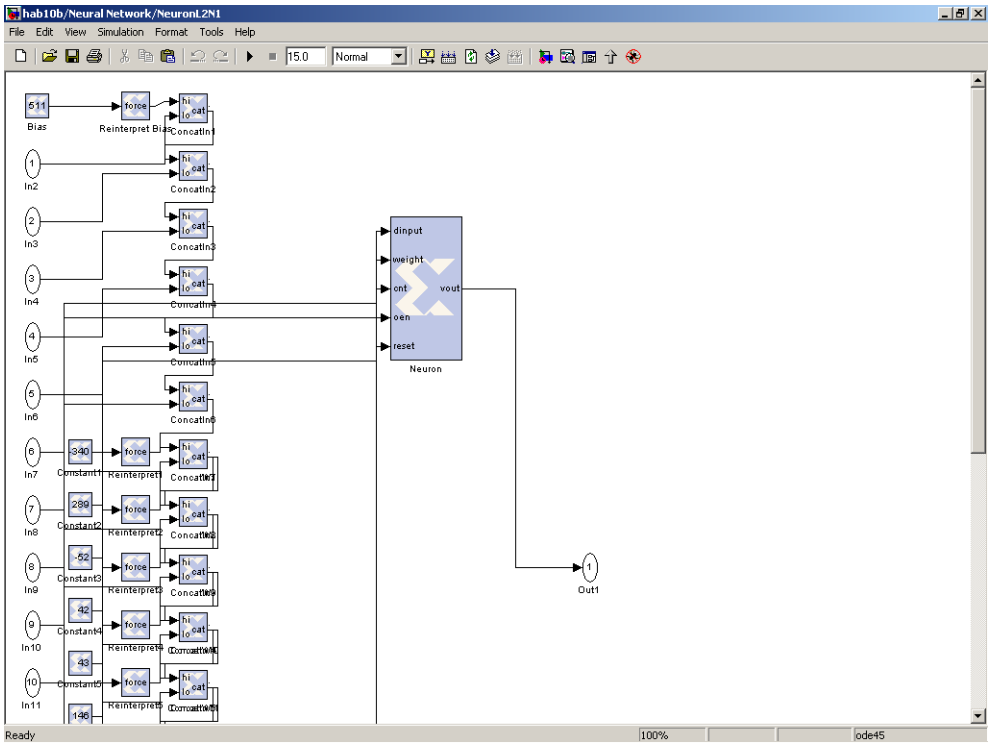


Figura 3.5. Aspecto del interior de cada neurona

Sin embargo, son necesarios además los ficheros Control_config.m, y neuron_wcte_config.m. El primero es el fichero de configuración de la *Black Box* de la unidad de control, y el segundo cumple idéntica función para la neurona. Los parámetros de las *Black Box*, declarados en sus ficheros de configuración, se han definido como variables en lugar de como valores concretos, de forma que no haya que generar estos ficheros en cada implementación, pero se requiera la existencia de dichas variables. Las variables utilizadas por los ficheros son:

- `lenNN`. Especifica la longitud en palabras de la ROM de la unidad de control.
- `sigmoidbits`. Indica los bits de resolución de la ROM de la función de activación.
- `vinputbits`. Indica el número de bits de cada entrada de la neurona y de su salida, la cual corresponde con el ancho de palabra de la ROM de la función de activación.
- `vweightbits`. Contiene el ancho en bits de cada peso.

- `widNN`. Especifica la anchura en bits de cada palabra de la ROM de la unidad de control.

Comentar también que los bloques de *System Generator*, al menos en la versión y sistema que se utilizaron para el desarrollo de la herramienta — Windows 2000 SP4, Matlab 7.0.1, System Generator 6.3 — no podían ser instanciados de la biblioteca de bloques; únicamente se podían tomar de otros sistemas de Simulink. Para solucionarlo, se creó un sistema que contiene todos los bloques necesarios por la herramienta. Este fichero es Plantilla.mdl.

Finalmente hay que tener en cuenta que la red espera señales en punto fijo, por lo que para usarla hay que realizar la conversión del formato tradicional de coma flotante de Simulink, mediante los bloques suministrados por *System Generator*, llamados *Gateway In* y *Gateway Out*.

En la figura 3.3 se observa el resultado final. En la figura 3.4 se observa en la parte superior la unidad de control con los *slices* encargados de separar las señales de control, y debajo se observan una interconexión de subsistemas. Cada subsistema contiene una neurona con sus pesos. El interior de cada subsistema se muestra en la figura 3.5, donde se observan los pesos — indicados como constantes — y los concatenadores, así como la *Black Box* de la neurona en sí.

Capítulo 4

Implementación de redes neuronales para sistemas empotrados

Un sistema empotrado es un sistema de computación de propósito específico, el cual está encapsulado completamente en el dispositivo que controla. Dichos sistemas tienen una gran importancia en la actualidad, ya que prácticamente cualquier sistema electrónico tiene uno o varios sistemas empotrados. En la figura 4.1 muestra la estructura básica de un sistema empotrado.

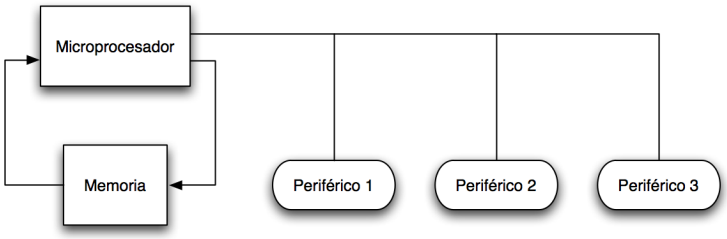


Figura 4.1. Estructura básica de un sistema empotrado

Los sistemas empotrados tienen algunas características que los hacen muy diferentes a otros sistemas de cómputo de propósito general, entre los que se puede destacar:

- El coste puede llegar a ser muy importante en estos sistemas, ya que algunos son fabricados en un número enorme, debido a que se utilizan en dispositivos muy comunes. Por eso, generalmente los sistemas empotrados tienen importantes restricciones tanto en la velocidad del procesador como en la cantidad de memoria, de forma que abaraten sus costes.

- El espacio y el consumo de energía suelen ser factores críticos en la mayoría de los sistemas empotrados. Esto es así debido a que se suelen utilizar para miniaturizar características de los productos finales, por lo que generalmente no es posible alojar sistemas de disipación de calor, sin contar con que, en un número considerable de ocasiones, son empleados para construir dispositivos a baterías.

- Otra característica es que son específicos para una determinada tarea, aunque el procesador empleado sea de propósito general. El software que utilizan estos sistemas se suele denominar *firmware*. Este software suele estar en memorias no volátiles y no suele incluir un sistema operativo.
- Son sistemas que funcionan en tiempo real, continuamente reaccionan variando su funcionamiento en el entorno del sistema.
- Por último, comentar que son sistemas críticos, o a los que no se les debe permitir fallar, como por ejemplo el sistema que controla el ABS de un coche.

Los sistemas empotrados utilizan todo tipo de procesadores, desde muy simples, como microcontroladores o procesadores de 8 bits, a procesadores muy complejos, como PowerPC o Pentium. Si la aplicación no requiere un software muy complejo, un microcontrolador o un procesador de 8 bits puede ser suficiente, pero si el sistema requiere mayor complejidad suele ser necesario procesadores más potentes. Sin embargo, es posible que el sistema no permita utilizar procesadores muy potentes y complejos, como un PowerPC, debido por ejemplo a requerimientos de consumo o espacio en la placa. En estos sistemas suele ser adecuada la utilización de procesadores simples de 32 bits.

Estos procesadores permiten ciertas ventajas sobre los de 8 bits, como mayor espacio de direccionamiento, menor coste de desarrollo debido a la existencia de mayor número de utilidades de desarrollo y de mayor y mejor soporte de bibliotecas, pero con un tamaño menor al de procesadores más complejos y potentes.

Los procesadores *soft-core* son procesadores escritos en lenguajes de descripción hardware preparados para ser sintetizados sobre FPGAs, en contraposición a los procesadores tradicionales o *hard-core*, que se encuentran en la oblea. Los hay de distinto tipo y potencia, desde pequeños microcontroladores a procesadores completos de 32 bits con todas las características que se esperan de un procesador moderno. Estos procesadores tienen además la ventaja de que son muy flexibles, ya que aquellas partes que no se necesiten pueden ser desactivadas, como cachés o MMU. Tienen la desventaja de que no alcanzan frecuencias tan altas como sus homólogos ASIC, si bien éstas suelen ser suficientes para la mayoría de aplicaciones.

Por tanto, para la mayoría de sistemas empotrados implementados en FPGAs pueden ser muy adecuados la utilización de este tipo de procesadores, debido a su flexibilidad y relativa velocidad de proceso.

Comentar también que los sistemas empotrados requieren desde su planteamiento un enfoque diferente al de otros sistemas, ya que en ellos las partes que lo componen, hardware y software, tienen la misma importancia y están profundamente ligadas entre sí. Esto obliga a utilizar una filosofía de diseño que integre ambas perspectivas de forma simultánea.

En esta parte del Proyecto se hará uso de la *suite* de Xilinx Inc. EDK — *Embedded Developer Kit* —. Es un conjunto de herramientas diseñadas para la creación de sistemas empotrados de forma integral, es decir, permite utilizar una metodología que persigue el desarrollo completo del sistema, tanto en su nivel hardware como software.

Las herramientas de las que se compone EDK son las siguientes:

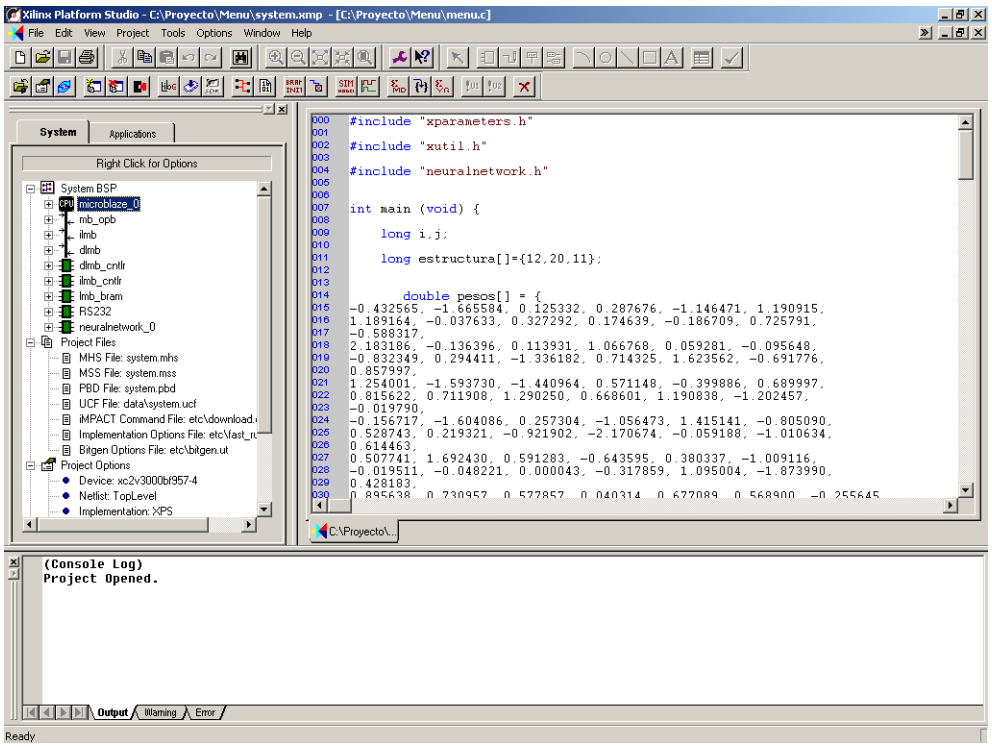


Figura 4.2. Apariencia de la herramienta Xilinx Platform Studio

- *Xilinx Platform Studio*, un IDE para la creación de diseños de sistemas empotrados, tanto desde el punto de vista software como hardware. Incluye una utilidad llamada *Platform Studio Software Development Kit* enfocada en el desarrollo y depuración de la parte software. En la figura 4.2 se observa su aspecto.

- Herramientas de desarrollo de software GNU, que incluye el compilador C/C++ *gcc* y el depurador *gdb*, entre otras.

- Otras utilidades para el desarrollo de software o hardware independientes a Xilinx Platform Studio, como un asistente para el desarrollo de sistemas integrados — *Base System Builder Wizard* —, el motor de depuración para los microprocesadores Microblaze y PowerPC — *XMD* —, o el gestor gráfico del mapa de memoria, entre otras. La gran mayoría de estas herramientas son accesibles desde Xilinx Platform Studio.

Sus características de diseño integral de sistemas empotrados, junto con su perfecta integración con las FPGAs de Xilinx, sobre las que se iban a realizar las implementaciones, hizo que nos decantásemos por esta aplicación.

4.1. Microprocesador elegido para la implementación

La elección del procesador merece un apartado aparte, ya que requiere ciertos comentarios más específicos que la simple descripción de características hecha en la introducción.

La integración tan alta del hardware con el software propia de los sistemas empotrados implica que sea necesario estudiar a fondo el procesador para comprobar que posee la funcionalidad necesaria y la ocupación de recursos mínima para la tarea que se aborda.

En principio se optó por la utilización del microcontrolador *soft-core* PicoBlaze, suministrado por Xilinx, debido a su pequeño tamaño y su código abierto, lo que permite modificarlo para que se ajuste a las necesidades del sistema, aunque esté limitado por licencia de uso a implementaciones únicamente sobre FPGAs de Xilinx.

Como características principales del PicoBlaze se pueden destacar:

- Pequeño tamaño, únicamente necesita 96 *slices* de una Spartan III.
- Alcanza una velocidad máxima de 43 MIPS en una Spartan III y hasta 76 MIPS en una Virtex-2.
- Memoria de programa de hasta 1.024 instrucciones, según la implementación.
- Contiene de 8 a 32 registros de 8 bits, dependiendo de la implementación.
- Permite realizar operaciones aritméticas (sumas y restas), lógicas y de desplazamiento en cualquiera de los registros.
- 256 puertos de entrada/salida de 8 bits.
- Una única entrada de interrupciones.
- Todas las instrucciones se ejecutan en 2 ciclos de reloj.
- No requiere ningún componente externo.
- Respuesta rápida y predecible a las interrupciones.

Para más información sobre sus características y diferentes versiones, véase [12], [13] y [14]. Además, en [5] se puede ver una tabla comparativa entre las distintas implementaciones.

En principio se pensó emplear la implementación del PicoBlaze para CPLD, el cual está definido utilizando código VHDL de alto nivel (codificación de estilo comportamental), lo que facilita su modificación ya que las restantes versiones de este microcontrolador están definidas a bajo nivel, utilizando primitivas específicas de la tecnología para conseguir mayores prestaciones, lo que sin embargo dificulta la portabilidad del procesador a otros dispositivos FPGA. Sus características son las comentadas, con 8 registros y memoria de programa de 256 instrucciones.

La falta de herramientas para el desarrollo de software sobre dicho microcontrolador es evidente, incluyendo únicamente un compilador de lenguaje ensamblador, llamado ams.exe, del cual se suministra el código, implementado en C.

Su arquitectura se muestra en la figura 4.3. Sus especificaciones se pueden ver en [13].

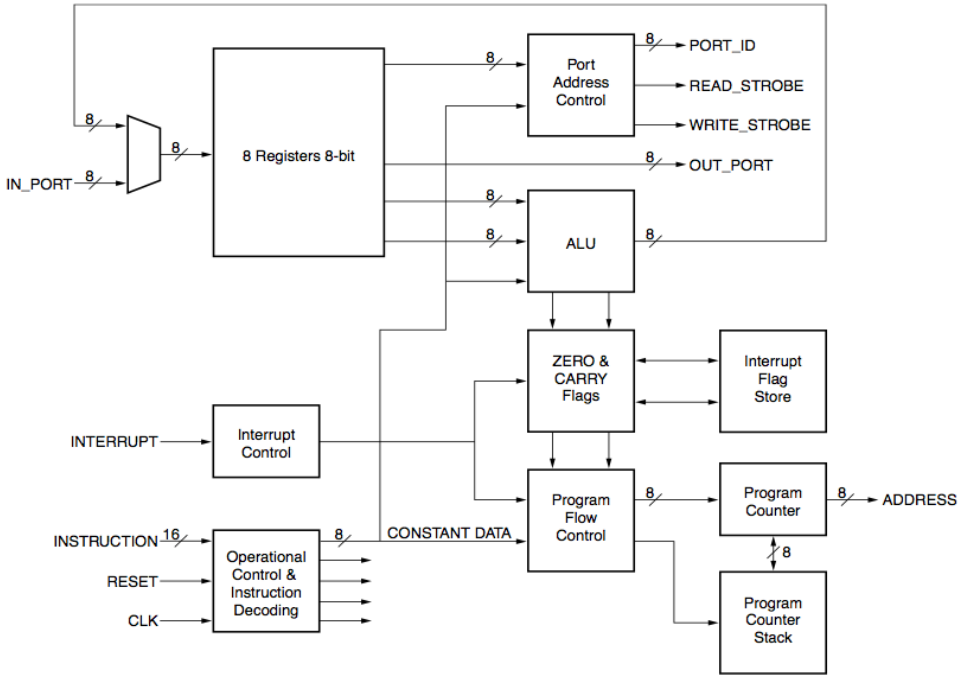


Figura 4.3. Arquitectura del PicoBlaze para CPLD

Por tanto, para el desarrollo de un software de cierto tamaño, la falta de un depurador es un problema muy serio para su realización. Por ello, se optó por el desarrollo de un depurador, y de un editor al que se integró una interfaz al depurador y la llamada al programa ensamblador, al que se le denominó PicoIDE, el cual se observa en la figura 4.4. Esto permite el desarrollo de software de manera integral para el PicoBlaze, permitiendo además comprobar el correcto funcionamiento del software desarrollado sin necesidad de programarlo sobre una FPGA.

Todo el software para PicoIDE fue desarrollado en Java, lo que permite su funcionamiento en cualquier ordenador con una máquina virtual de Java. Se comprobó su correcto funcionamiento tanto en sistemas operativos Windows, como en Linux y en Mac OS X.

El ensamblador tiene la misma licencia que el código del PicoBlaze, por lo que se permite modificar el código fuente. Se comprobó que el código proporcionado por Xilinx no funcionaba correctamente en otro entorno distinto a Windows, por lo que se modificó para conseguir que fuera independiente de la plataforma. El ensamblador

también se comprobó en los mismos sistemas que PicoIDE, utilizando en todos ellos el compilador de GNU, *gcc*. Por tanto, las únicas limitaciones para la ejecución del entorno de desarrollo es la existencia de una máquina virtual Java y la existencia de un compilador de C.

El depurador se desarrolló como una librería, por lo que es posible implementar una interfaz que opere con él, dando mayor flexibilidad a su uso. Además está diseñado pensando en la posibilidad de modificar el PicoBlaze, por lo que adaptar el depurador para que refleje esas modificaciones no es una tarea compleja. La adición de nuevas instrucciones es una tarea simple, la cual se documentó, y modificar el número de registros o su tamaño, así como la longitud máxima de la memoria de programa es algo inmediato. Además se permite modificar el nombre de los registros en tiempo de ejecución, característica que puede facilitar la lectura del código en el depurador.

Todo el software desarrollado está siendo empleado para la docencia de prácticas de la asignatura Arquitectura de Computadores de 4º de Ingeniero de Telecomunicación, con una experiencia satisfactoria.

PicoIDE utiliza la licencia GPL, para asegurar que continúe siendo de código abierto tanto el programa original como las posibles modificaciones.

Para más información sobre la aplicación, como arquitectura y funcionamiento, consultar [5].

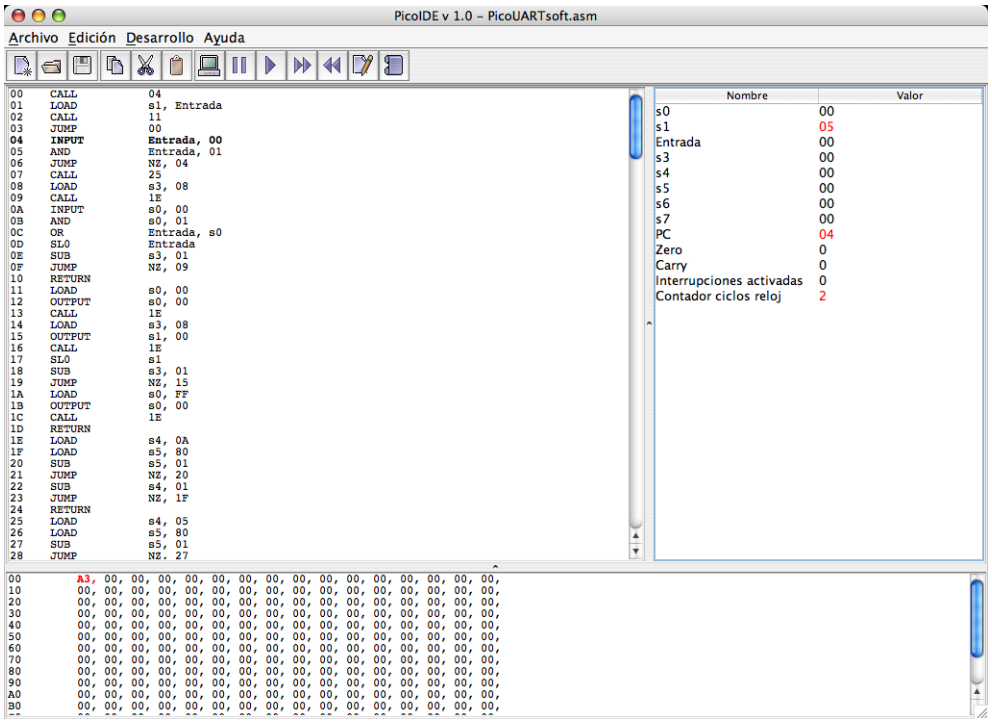


Figura 4.4. El entorno de de desarrollo PicoIDE

Tras realizar el software y trabajar con él, se comprobó que las características del PicoBlaze eran excesivamente modestas para nuestras necesidades. Su memoria de pro-

grama es demasiado pequeña, sólo permite números enteros y no realiza operaciones como multiplicación o división, únicamente suma y resta, sin contar su limitación para el almacenamiento de datos. Si bien es completamente modificable, añadir toda la funcionalidad básica necesaria para tratar con la red puede ser una tarea mayor aún que la de realizar la propia red.

Por tanto, se optó por abandonar esta opción en favor de microprocesadores de mayores prestaciones. Soportados directamente por Xilinx en su herramienta EDK nos encontramos el MicroBlaze, procesador *soft-core* completo de 32 bits, codificado de forma óptima para las FPGAs de Xilinx. Como características principales, cabe destacar:

- Alcanza velocidades considerables, 125 MHz en Virtex-II (con *speed grade* -5), lo que inplica aproximadamente 115 DMIPS.
- Al estar diseñado en VHDL es totalmente parametrizable, lo que permite las partes que no sean necesarias, como la FPU, no sean sintetizadas.
- Maximiza la utilización de LUTs en FPGAs Virtex.
- Puede ser sintetizado en cualquier FPGA que contenga Block RAM.

En la figura 4.5 se muestra la arquitectura del MicroBlaze. Para más información sobre el MicroBlaze, véase [11].

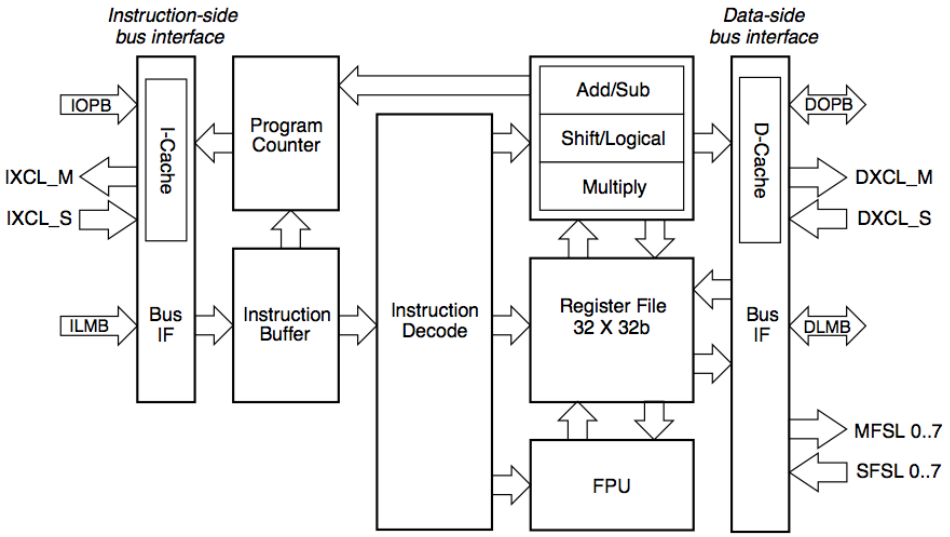


Figura 4.5. Arquitectura del MicroBlaze

Por otro lado, EDK soporta también el microprocesador PowerPC 405, también de 32 bits, incluido físicamente (*hard-core*) en la oblea de las FPGAs de la familia Virtex II Pro y Virtex 4 de Xilinx, pudiendo encontrar hasta 4 PowerPC por dispositivo. Como características de este microprocesador, se pueden destacar las siguientes:

- Cachés de 16 KB de datos e instrucciones.
- Pipeline de 5 etapas.
- Consta de una MMU empotrada.
- Al estar implementados físicamente, pueden funcionar a gran velocidad, mayor que en el caso del MicroBlaze, alcanzando hasta los 450 MHz de velocidad de reloj, lo que permite obtener 680 DMIPS.
- Tiene la desventaja de que no se encuentra en todas las FPGAs, únicamente en las Virtex II Pro y en las Virtex 4.

La arquitectura del procesador PowerPC se observa en la figura 4.6. Para conocer más en profundidad sus características, consultar [15].

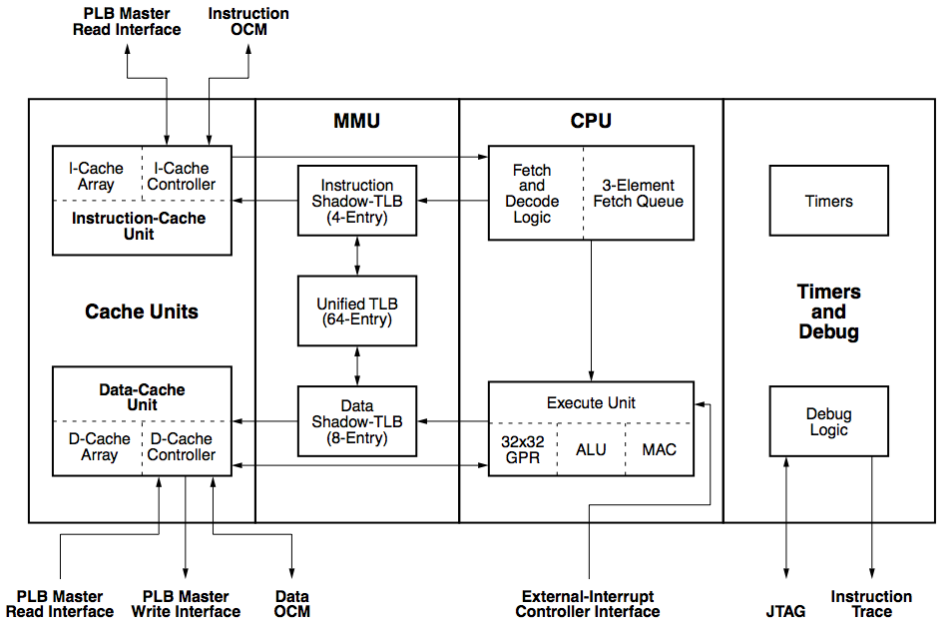


Figura 4.6. Arquitectura del PowerPC 405

Ambos procesadores — Microblaze y PowerPC — incluyen versiones de las mismas herramientas de desarrollo, como el compilador de C (*gcc*) o el depurador (*gdb*), entre otras. Además ambos procesadores poseen en sus librerías las mismas funciones no estándar suministradas por Xilinx, por lo que si se desarrolla código en C mediante el uso de las librerías estándar y empleando las funciones no estándar de Xilinx, el código puede compilarse sin ninguna modificación para cualquiera de los dos procesadores.

En este Proyecto, se empleará el MicroBlaze, debido a que la FPGA empleada para las pruebas no contiene el PowerPC. Sin embargo, con una simple recompilación debe funcionar todo lo desarrollado con el procesador PowerPC.

4.2. Arquitectura

El sistema de la red neuronal es implementado en este caso como periférico, no como parte integrante de un microprocesador. Esto es así debido a que de esta forma el sistema se hace independiente del procesador.

La nueva arquitectura de la red neuronal, que denominaremos HANN (Hardware ANN), tiene ciertas diferencias estructurales con respecto a la desarrollada para HANNA, con la intención de dotarla de mayor flexibilidad, principalmente debido a que esta arquitectura va a incluir la posibilidad de modificar los pesos. Dicha característica tiene su justificación en el aprovechamiento de las capacidades del sistema empotrado para la ejecución de software, lo que permite incluir un algoritmo para el cálculo de los pesos, algo impensable de desarrollar como hardware debido a los altos consumos de área que requeriría. También es interesante la opción de permitir la sustitución de los pesos de la red desde el exterior del sistema empotrado, de forma que sea posible implementar un algoritmo software que permita al usuario introducir los nuevos pesos a la red.

A pesar de los cambios, el funcionamiento de la red es el mismo que para el caso de HANNA, con los mismos retardos y los mismos requerimientos de tiempos.

En esta estructura, la cual se muestra en la figura 4.7, los pesos se encuentran definidos en el interior de la neurona, de forma que desaparece dicho bus de entrada. Sin embargo, para poder modificar e introducir los pesos, se añade una nueva entrada con la anchura en bits de un peso, por donde se irán introduciendo los nuevos pesos uno a uno, de forma serializada. Por tanto, se considera lógico almacenar los pesos en el interior de la neurona en un registro de desplazamiento. Salvo que se diga lo contrario, se supondrá que el primer peso corresponde al del bias, el segundo al de la primera entrada, y así sucesivamente, empleando la misma convención impuesta al especificar la arquitectura de la neurona.

Además, la neurona tiene una salida del mismo tamaño que la mencionada entrada, de forma que los pesos, al salir del registro de desplazamiento vayan saliendo por dicha salida. De esta forma, si se conecta la salida de pesos de una neurona con la entrada de la neurona siguiente, se pueden introducir los pesos de toda la red. Así, en el sistema completo, al ir introduciendo pesos por la última neurona de la última capa, éstos se irán propagando por la red, de forma que los pesos enviados en primer lugar se almacenen en la primera neurona de la capa de salida, los pesos enviados a continuación finalmente terminen en la segunda neurona de la capa de salida, y así sucesivamente hasta concluir con la transferencia de los pesos de la última neurona de la última capa.

Los pesos son enviados a un periférico HANN utilizando el bus OPB. El bus OPB (*On-Chip Peripherical Bus*) es un bus diseñado por IBM, que forma parte de la arquitectura *CoreConnect*. Es un bus totalmente síncrono que no está pensado para que se conecte directamente al procesador, sino que se utilice un *bridge* del bus PLB (*Processor Local Bus*) al OPB, de forma que se permita la conexión de los periféricos esclavos al microprocesador, y viceversa, es decir, que maestros del bus OPB accedan a periféricos que se encuentren conectados en el bus PLB. El bus OPB tiene como características más destacadas:

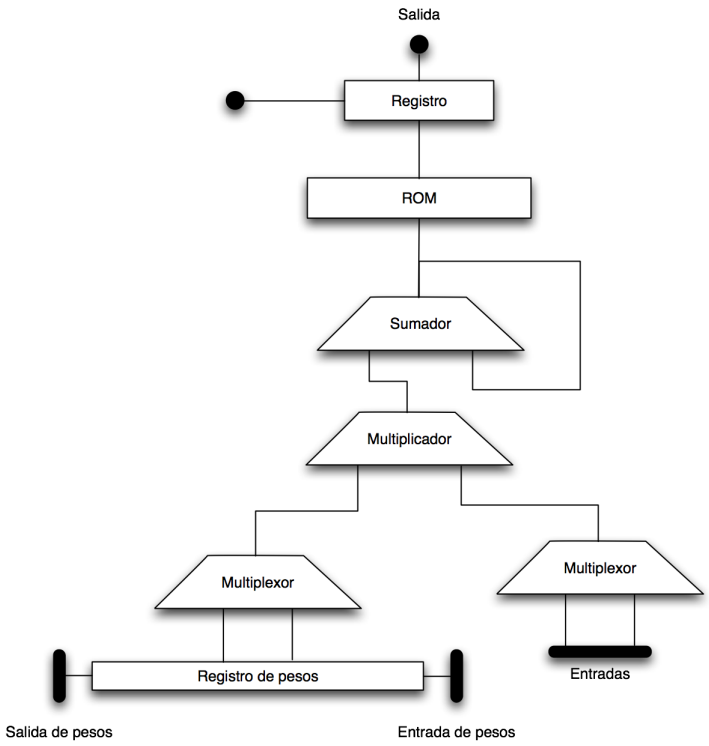


Figura 4.7. Arquitectura de una neurona para HANN

- Bus de direcciones de hasta 64 bits, permitiendo implementaciones del bus de datos de 32 y 64 bits.
- Permite la interconexión de esclavos de 8, 16, 32 y 64 bits.
- Soporte de varios maestros.
- Tamaño del bus dinámico: byte, media palabra, palabra y doble palabra.
- Es posible realizar transferencias en un solo ciclo entre maestro y esclavo del bus OPB.
- Soporte para el empleo de direcciones consecutivas.
- Los periféricos pueden ser de memoria mapeada, o funcionar por DMA, o ambos.
- Posibilidad de apropiación del bus por parte de los maestros.
- Posibilidad de los esclavos de deshabilitar el timeout y de solicitar retransmisiones.

En la figura 4.8 se puede observar una estructura típica de un sistema empotrado que utilice el bus OPB. En general, todos los periféricos que requieren de alto ancho

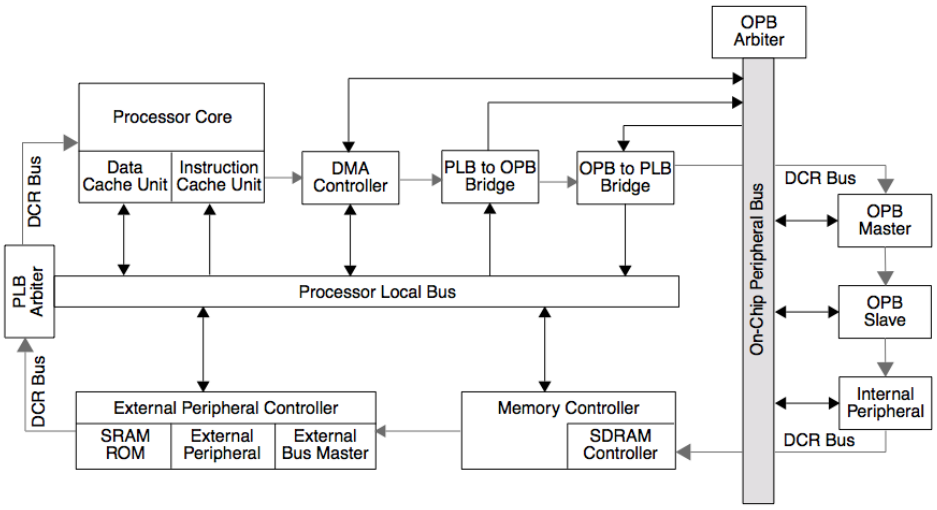


Figura 4.8. Interconexión del bus OPB

de banda se conectan al bus PLB, al cual está conectado directamente el procesador, conectándose los periféricos que no requieren velocidades tan altas al bus OPB. En nuestra arquitectura, las necesidades de ancho de banda entre el procesador y el periférico son muy pequeñas, limitándose al envío de los pesos, por lo que la conexión al bus OPB es lo más indicado.

Para una información más detallada del bus OPB y de la arquitectura *CoreConnect*, véase [9] y [10].

Nuestro periférico HANN se comporta como un esclavo del bus, donde la dirección base se emplea para la escritura de los pesos. Leyendo de la misma dirección se puede leer el último peso introducido.

En este Proyecto se utilizará la implementación del bus OPB realizada por Xilinx, la cual tiene una serie de limitaciones respecto a la especificación de IBM, por ser ineficiente en el consumo de recursos de la FPGA o porque reduzca la máxima velocidad del reloj al ser sintetizado el sistema, entre las cuales se puede destacar:

- Los buses de datos y direcciones han de ser de 32 bits. Los periféricos que utilicen otra anchura de bus tienen que conectarse a ese bus de 32 bits con la limitación de direccionamiento.

- Todos los periféricos tienen que hacer que su salida sea cero cuando estén inactivos.

En todo caso, estas limitaciones no modifican las especificaciones de IBM, por lo que nuestro periférico funcionará correctamente en cualquier especificación que la cumpla, siempre que sea de 32 bits. El resto de limitaciones y detalles del uso del bus OPB en las FPGAs Xilinx se pueden ver en [16].

El periférico tendrá entradas y salidas independientes, no conectadas al bus OPB, debido a que esto proporciona mayor flexibilidad del sistema, ya que permite un funcionamiento independiente del procesador principal. En caso de que fuese necesario un control completo de la E/S por parte del procesador, éste se podría realizar con dos sencillos bloques que controlen la conexión de la entrada y salida de la red neuronal con cualquiera de los buses del procesador (PLB u OPB).

Comentar además que en la arquitectura de HANN, la unidad de control de la red es diferente a la utilizada por HANNA. En este caso está creada como un nuevo bloque en VHDL, consistente básicamente en un acumulador y un comparador, de forma que dicho subsistema genere las señales de una capa determinada. El acumulador se va incrementando hasta alcanzar el número de entradas de la capa que controla, momento en el que el acumulador se detiene. Este cambio se hizo debido a que es sencillo y no consume excesivos recursos de la FPGA, y su implementación en VHDL era más simple. Al estar diseñada como un subsistema, es posible sustituir esta implementación por otra (como la ROM empleada en HANNA) con la misma funcionalidad y el mismo conexionado sin que se produzcan variaciones en el funcionamiento de la red.

4.2.1. Implementación del periférico

La arquitectura de red neuronal hardware (HANN) completamente genérica y parametrizable, que se ha desarrollado para funcionar como periférico del procesador MicroBlaze, cuya estructura se muestra en la figura 4.9, ha sido completamente desarrollada en VHDL, estando implementada en los siguientes ficheros:

- neuralnetwork.vhd: Sistema que se encuentra en el nivel superior de la jerarquía del periférico, el cual se conecta directamente al bus OPB. Simplemente infiere el subsistema *user_logic* e infiere la lógica de control necesaria para detectar cuando en el bus OPB se está direccionando al periférico.

- user_logic.vhd: Corazón del periférico, es el encargado de la generación de la red, de la inferencia de las unidades de control y de todas las interconexiones, así como de realizar la gestión de bajo nivel del bus OPB.

- Control.vhd: Encargado de modelar las unidades de control. Cada unidad de control gobierna a una capa.

- neuron_wcte.vhd: Código de la nueva neurona, con la arquitectura y modificaciones respecto a HANNA presentadas en apartados anteriores de este capítulo.

- types.vhd: La estructura de la red neuronal (entradas, y número de neuronas por capa) se pasa como genérico a *neuralnetwork* utilizando un tipo de dato definido en este fichero. Esto es necesario debido a que el nuevo tipo de dato no es posible definirlo en el mismo sistema que lo va a utilizar como genérico.

La lógica para detectar cuándo se está direccionando al periférico se implementa mediante el bloque de la librería estándar suministrada por EDK `common_v1_00_a`

llamado `pselect`, el cual activa su puerto de salida cuando hay una dirección válida del periférico. Esta señal de control se envía a la red neuronal como puerto del subsistema `user_logic`.

La unidad de control de cada capa se implementa mediante un acumulador, el cual se incrementa hasta llegar a un límite (el número de entradas de las neuronas de esa capa), para a continuación, en el siguiente ciclo de reloj, activar una señal que controle la salida de las neuronas, y finalmente detenerse. Esa señal sirve también de disparador para iniciar el acumulador de la unidad de control de la siguiente capa. En la última capa, esta señal se encarga de activar el reset de todas las unidades de control para poder reiniciar los cálculos.

La red se construye mediante dos bucles `for .... generate` anidados, uno que genera las capas de la red, y otro que implementa las neuronas de cada capa. En el interior del primer bucle, pero antes de entrar en el segundo bucle, se instancia la unidad de control.

Cabe destacar el hecho de que la primera y la última capa son específicas. La primera capa tiene como característica diferenciadora el hecho de que hay que conectar el registro que almacena el nuevo peso introducido al sistema con la entrada correspondiente de la última neurona. Además el disparador de la unidad de control siempre estará activo en esta capa, ya que no ha de esperar a que ninguna capa anterior termine sus cálculos. Por último, las entradas de las neuronas son las entradas de la red.

En el caso de la última capa, ésta tiene como peculiaridad el hecho de que la señal de control que activa la salida de las neuronas además debe reiniciar todas las unidades de control, para reiniciar así el proceso de los cálculos de la red. Además hay conectar la salida de las neuronas a la salida de la red.

Conjuntamente, las primera neurona de todas las capas restantes también son un caso especial, debido a que la entrada de los pesos debe de conectarse con la salida de los pesos de la última neurona de la capa anterior.

En cuanto a la lógica necesaria para tratar el envío y recepción de los pesos a través del bus OPB, ésta se ha implementado de forma que cada vez que se escribiera en una dirección del mapa de memoria del periférico, sea cual sea, el dato se toma como un nuevo peso. En caso de lectura a cualquier dirección del periférico, se devuelve el valor del último peso enviado.

En el caso de la neurona, ya se ha comentado que las principales diferencias con la ya implementada para HANNA es la nueva entrada de pesos, y su salida asociada. Además hay que indicar la existencia de una nueva señal de control para indicar cuándo se introduce un nuevo peso. Esta señal ha sido implementada de forma que utiliza lógica negada. Los pesos son almacenados en un único registro de desplazamiento, de forma que al introducir nuevos pesos, éstos se vayan desplazando a lo largo de dicho registro. Los pesos son extraídos con un multiplexor, de la forma comentada en el apartado 2.3.1.

En esta implementación sólo se puede utilizar como función de activación de la neurona la tangente hiperbólica. Dicha función de activación es generada automáticamente mediante la utilización de los parámetros de la tangente hiperbólica, definidos como genéricos de la red. El algoritmo empleado es similar al utilizado en la arquitectura

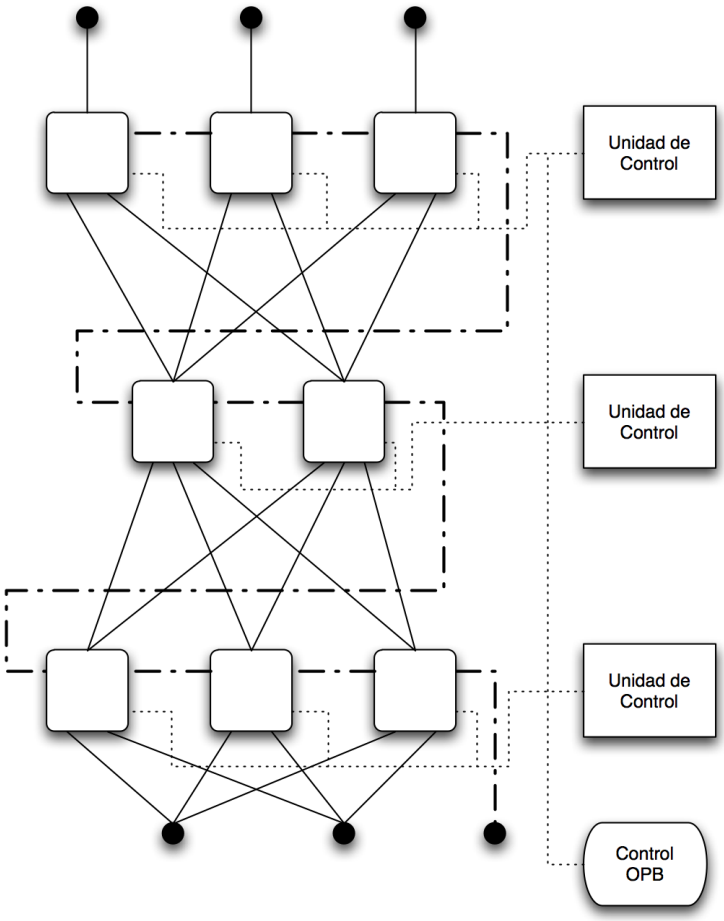


Figura 4.9. Arquitectura de HANN para ser utilizada como periférico

anterior. En este caso, también la función de activación es sintetizada como una ROM.

4.3. Funcionamiento del periférico

El periférico se comporta como cualquier otro dentro de la herramienta *Platform Studio*, si bien tiene ciertas características que conviene especificar.

```
1  entity neuralnetwork is
2    generic
3    (
4      -- Genéricos de la red ------------------
5     C_NLAYERS          : positive      := 2;
6     C_LAYERSTRUC       : layer_array   := (1, 5, 1);
7
8     C_vinputbits       : integer       := 8;
9     C_vweightbits      : integer       := 10;
10    C_a                : integer       := 2;
```

```
11      C_c                 : integer       := 0;
12      C_min               : integer       := -30;
13      C_max               : integer       := 30;
14      C_sigmoidbits       : integer       := 10;
15      inputs              : integer       := 12;
16      outputs             : integer       := 11;
17
18
19      -- Genéricos del bus OPB
20      C_BASEADDR                          : std_logic_vector      := X"00000000";
21      C_HIGHADDR                          : std_logic_vector      := X"0000FFFF";
22      C_OPB_AWIDTH                        : integer               := 32;
23      C_OPB_DWIDTH                        : integer               := 32;
24      C_FAMILY                            : string                := "virtex2p"
25    );
26    port
27    (
28      -- Puertos de la red
29      dinput      : in std_logic_vector(C_LAYERSTRUC(1)*C_vinputbits - 1
            downto 0);
30      doutput     : out std_logic_vector(C_LAYERSTRUC(C_NLAYERS+1)*
            C_vinputbits -1 downto 0);
31
32      -- Puertos del bus OPB
33      OPB_Clk                             : in  std_logic;
34      OPB_Rst                             : in  std_logic;
35      Sl_DBus                             : out std_logic_vector(0 to
            C_OPB_DWIDTH-1);
36      Sl_errAck                           : out std_logic;
37      Sl_retry                            : out std_logic;
38      Sl_toutSup                          : out std_logic;
39      Sl_xferAck                          : out std_logic;
40      OPB_ABus                            : in  std_logic_vector(0 to
            C_OPB_AWIDTH-1);
41      OPB_BE                              : in  std_logic_vector(0 to
            C_OPB_DWIDTH/8-1);
42      OPB_DBus                            : in  std_logic_vector(0 to
            C_OPB_DWIDTH-1);
43      OPB_RNW                             : in  std_logic;
44      OPB_select                          : in  std_logic;
45      OPB_seqAddr                         : in  std_logic
46    );
```

Código 4.1. Interfaz de HANN cuando actúa como periférico conectado al bus OPB

En el código 4.1, se observan todos los genéricos necesarios para definir la red, así como todos los puertos del periférico, junto con sus valores por defecto. A continuación se va explicar el significado de dichos genéricos, obviando aquellos que son utilizados por el bus OPB:

- `C_NLAYERS` muestra el número de capas de la red neuronal.
- `C_LAYERSTRUC` es un array que contiene la estructura de la red neuronal.

- `C_vinputbits` especifica la anchura en bits de la entrada y de la salida de cada neurona.
- `C_vweightbits` indica la anchura en bits de cada peso.
- `C_a` es la pendiente de la función de activación.
- `C_c` controla el centro de la función de activación.
- `C_min` muestra el valor de ordenadas mínimo en el que se definirá la función de activación.
- `C_max` ídem pero para el máximo.
- `C_sigmoidbits` especifica la resolución vertical en bits que tiene la función de activación.
- `inputs` indica el número de entradas de la red neuronal.
- `outputs` indica el número de salidas de la red.

En el caso de los puertos, se observan:

- `dinput` es el bus de datos de entrada de la red neuronal.
- `doutput` es el bus de datos de salida de la red neuronal.

Todos los genéricos se pueden modificar dentro de Xilinx Platform Studio, excepto `C_LAYERSTRUC`, debido a que no se pueden asignar genéricos que no sean de los tipos de datos básicos predefinidos, por lo que este parámetro es necesario modificarlo directamente en el código VHDL. El uso de los genéricos `inputs` y `outputs` es necesario debido a que Xilinx Platform Studio no puede leer genéricos que no son asignados dentro de la herramienta, por lo que la herramienta desconocería el tamaño de los buses `dinputs` y `doutputs` si fuesen definidos únicamente en el genérico `C_LAYERSTRUC`. El uso de dichos genéricos es únicamente la definición de la anchura ambos buses, de forma que la herramienta pueda tener constancia. Por tanto, hasta que se añada la posibilidad de emplear genéricos con tipos definidos por el usuario en la aplicación, será necesaria esta redundancia de genéricos.

4.4. Software del periférico

En EDK, cada periférico tiene un controlador software, o *driver*, de forma que el procesador pueda fácilmente comunicarse con el periférico, además de aportar una serie de facilidades añadidas para trabajar con el periférico. En nuestro caso, el driver tiene funciones para el envío de un peso, o de un conjunto de pesos, y para la lectura del último peso enviado (en caso de enviar un conjunto de pesos, esta función sólo puede leer el último peso del conjunto). Además como complemento, el driver incluye una

implementación simple del algoritmo de entrenamiento supervisado Backpropagation en coma flotante, una función para transformar los pesos calculados en enteros, y un simulador de la red neuronal.

Dichos complementos están escritos en ANSI C, por lo que dichas funciones se pueden ejecutar en cualquier computador con un compilador de C. Esto puede ser útil para ver cómo se comporta el sistema de un modo más cómodo que la implementación en una FPGA para realizar pruebas.

```
1  //
       ////////////////////////////////////////////////////////////////////////////
2  // Filename:          C:\Proyecto\Perifericos\MyProcessorIPLib\drivers\
       neuralnetwork_v1_00_a\src\neuralnetwork.h
3  // Version:           1.00.a
4  // Description:       neuralnetwork Driver Header File
5  // Date:              Thu Aug 25 16:19:03 2005 (by Create and Import
       Peripheral Wizard)
6  // Author:        José Carlos Fernández Conesa
7  //
       ////////////////////////////////////////////////////////////////////////////

8
9  #ifndef NEURALNETWORK_H
10 #define NEURALNETWORK_H
11
12 /****************** Include Files ********************/
13
14 #include "xbasic_types.h"
15 #include "xstatus.h"
16 #include "xio.h"
17
18 /****************** Constant Definitions *****************/
19
20
21 #define NEURALNETWORK_WriteWeight(BaseAddress, Data) \
22   Xio_Out32((BaseAddress), (Data))
23
24 #define NEURALNETWORK_mReadReg(BaseAddress, RegOffset) \
25   Xio_In32((BaseAddress) + (RegOffset))
26
27
28 #define NEURALNETWORK_readLastWeight(BaseAddress) \
29   Xio_In32((BaseAddress))
30
31 /****************** Function Prototypes ******************/
32
33
34 /****** Funciones para operar con el periférico ********/
35
36 // Manda un peso a la red neuronal
37 void writeWeight(Xuint32 BaseAddress, Xint32 Data);
38 //PARÁMETROS:
```

```
39 // BaseAddress: dirección base del periférico.
40 // Data: peso que se enviará.
41
42
43 // Manda un conjunto de pesos a la red neuronal
44 void writeWeights(Xuint32 BaseAddress, Xint32 Data[], int weights);
45 //PARÁMETROS:
46 // BaseAddress: dirección base del periférico.
47 // Data: puntero a los pesos que se enviarán.
48 // weights: número de pesos que se enviarán
49
50
51 // Lee el último peso que se envió
52 Xint32 readLastWeight(Xuint32 BaseAddress);
53 // Devuelve el último peso enviado a la red.
54 //PARÁMETROS:
55 // BaseAddress: dirección base del periférico.
56
57
58 /*************** Funciones adicionales ****************/
59 // Algoritmo de red neuronal empleando coma flotante
60 void red(double* pesos, double* entradas, long* estructura, int capas,
       double* salida, int a, int c);
61 // PARÁMETROS:
62 // pesos: puntero con los pesos de la red. El primer valor apunta al bias
       de la primera neurona de la primera capa
63 // entradas: puntero con las entradas de la red.
64 // estructura: puntero que contiene la estructura de la red, de la forma
       (entradas, capa1, capa2, ..., salida).
65 // capas: entero con el número de capas de la red.
66 // salida: puntero donde se guardarán las salidas de todas las neuronas.
67 // a: pendiente de la función de activación.
68 // c: centro de la función de activación.
69
70
71 // Inicializa todas las variables que necesitan gestión de memoria
72 void initNN(long* estructura, int capas);
73 // PARÁMETROS:
74 // estructura: puntero que contiene la estructura de la red, con la forma
       comentada en void red(...).
75 // capas: entero con el número de capas de la red.
76
77
78 // Algoritmo backpropagation
79 double BackProp(double* mu, double* test, double* deseada, int ntest,
       double* pesos, long* estructura, int capas,
80       int a, int c, double (*C)(int, int*, double, double), int argn, int
           * args, double erroresO);
81 // Devuelve el error cometido en todo la época.
82 // PARÁMETROS:
83 // mu: puntero que contiene el escalón del gradiente descendiente.
84 // test: puntero que contiene todos los patrones de test de la época, de
```

```
    la forma (entrada1delpatron1, entrada2delpatron2, ...,
    entrada1delpatron2, ..., entradaNdelpatronM).
85  // deseada: puntero que contiene las salidas deseadas de la red para los
    patrones.
86  // ntest: número de patrones de la época.
87  // pesos: puntero que contiene los pesos de la red. Al salir contiene los
     pesos calculados.
88  // estructura: puntero que contiene la estructura de la red, con la forma
     comentada en void red(...).
89  // capas: número de capas de la red.
90  // a: pendiente de la función de activación.
91  // c: centro de la función de activación.
92  // *C: puntero a la derivada de la función de coste.
93  // argn: número de argumentos que se introducirán en la función de coste.
94  // args: puntero que contiene los argumentos que se introducirán en la
    función de coste.
95  // erroresO: valor del error que se produjo en la época anterior.
96
97
98  // Derivada de la función de coste del error cuadrático medio
99  double coste(int narg, int* argumentos, double salida, double deseada);
100 // Devuelve la derivada de la función de coste.
101 // PARÁMETROS:
102 // narg: parámetro necesario para cumplir la cabecera y poder enviar el
    puntero a función como parámetro de BackProp.
103 // argumentos: idem.
104 // salida: salida de la red neuronal.
105 // deseada: salida deseada de la red neuronal.
106
107
108 // Función para pasar los pesos de coma flotante a enteros
109 void weightToInt(double* pesos, long* pesosConv, int npesos, int wbits);
110 // PARÁMETROS:
111 // pesos: puntero a pesos originales.
112 // pesosConv: puntero donde se guardarán los pesos convertidos.
113 // npesos: número de pesos que se convertirán.
114 // wbits: número bits de los pesos convertidos.
115
116
117 // Simulador del periférico, algoritmo de la red neuronal en enteros
118 void redInt(long *pesos, long *entradas, long* estructura, int capas,
    long *salida, int a, int c, int min, int max, int inputbits, int
    weightbits, int sigmoidbits);
119 // PARÁMETROS:
120 // pesos: puntero a los pesos de la red.
121 // entradas: puntero a las entradas de la red.
122 // estructura: puntero que contiene la estructura de la red, con la forma
     comentada en void red(...).
123 // capas: número de capas de la red neuronal.
124 // salida: puntero donde se guardarán las salidas de todas las neuronas.
125 // a: pendiente de la función de activación.
126 // c: centro de la función de activación.
```

```
127 // min: mínimo de la función de activación que se muestreará.
128 // max: máximo de la función de activación que se muestreará.
129 // inputbits: número de bits de las entradas y salidas.
130 // weightbits: número de bits de los pesos.
131 // sigmoidbits: número de bits de resolución de la función de activación.
132
133 #endif // NEURALNETWORK_H
```

Código 4.2. Cabecera del driver

En el código 4.2 se muestra la cabecera del driver desarrollado para HANN. En los comentarios se explica cada función y sus parámetros. Como se puede observar, en la implementación del Backpropagation la derivada de la función de coste es variable, introduciéndose como un puntero a función. En el driver se incluye la función de coste del error cuadrático, si bien se puede utilizar cualquier otra definida por el usuario que tenga los mismos argumentos. Para que el número de argumentos no sea un problema, el segundo de los parámetros es un puntero a entero, que puede contener tantos argumentos como se desee. El primer parámetro se usa para indicar el número de argumentos incluidos en el puntero, si fuese necesario.

Capítulo 5

Aplicación al Reconocimiento de Hablantes

En este capítulo se va a llevar a cabo la implementación de una red neuronal cuya función es el reconocimiento de cada miembro de un conjunto de once hablantes de idioma castellano. Dicha identificación se realiza mediante el análisis de las vocales *a*, *i* y *o*.

Para caracterizar cada vocal se van a analizar los cuatro primeros formantes de la frecuencia de resonancia de tracto vocal.

Por tanto, la red necesitará 12 entradas — una por formante — y 11 neuronas en la capa de salida — una por hablante —. Además se utilizará una capa oculta de 20 neuronas.

Se realizó una implementación sobre Matlab empleando un computador convencional — Pentium 4 a 3,2GHz con 2GB de RAM, sobre Windows XP Professional SP2 —, que servirá como referencia para comprobar si la implementación hardware proporciona un incremento en la velocidad. Esta implementación de referencia invirtió 5,62 milisegundos para el cálculo de 11 patrones de entrada, de forma que necesita 0,511 milisegundos para cada uno. Esto proporciona una velocidad máxima de procesamiento de aproximadamente 1.960 muestras por segundo.

5.1. Implementación *standalone* usando HANNA

La metodología a seguir para la implementación de la red neuronal se divide en dos pasos claramente diferenciados. En primer lugar, tras decidir la estructura de la red, se ha de realizar el cálculo de los pesos con algoritmos convencionales de la forma habitual. Este paso se puede obviar si dichos pesos son conocidos.

El segundo paso, específico de la implementación, consiste en la conversión de los pesos a enteros, y a continuación la implementación de la red mediante la interfaz gráfica de HANNA.

Para esta aplicación no se va a hacer referencia al cálculo de los pesos, ya que no es relevante para los resultados de las prestaciones de la red en cuanto a área y velocidad. El algoritmo de aprendizaje utilizado en esta aplicación ha sido el Backpropagation con las ecuaciones indicadas en el capítulo 3. Para obtener los pesos como enteros, simplemente se multiplican todos los pesos por la constante que haga que el peso con el

mayor valor absoluto obtenga el máximo valor posible en valor absoluto para el número de bits deseados.

Una vez obtenidos los pesos cuantizados, se procede a la generación de la red neuronal mediante el uso de la interfaz de HANNA, rellenando los datos solicitados. La mayor parte de dichos datos tienen que ver con la estructura de la red, por lo que son los mismos que si se empleara una red convencional. Los únicos parámetros no definidos en la red neuronal convencional son el número de bits que emplea la red. Se pueden definir tanto los bits de entrada, como los bits de los pesos, utilizados en el paso anterior al convertir los pesos a enteros, y finalmente la resolución en bits de la tabla de la función de activación. La anchura de cada salida de la función de activación tiene la misma anchura que la entrada, ya que su salida es la entrada de la siguiente capa.

Además de los bits de resolución, el diseñador también puede definir las propiedades de la función de activación, lo que puede llevar a que una misma red cambie su funcionamiento de forma importante sólo modificando la función de activación. En las pruebas que se han llevado a cabo, el aumento de la pendiente de la función de activación, hasta un cierto límite, para una misma red y con los mismos pesos condujo, en casos de clasificación, a una mejora en su funcionamiento, debido a que los umbrales son más marcados.

Después de realizar varias pruebas, los parámetros finalmente utilizados para esta aplicación de reconocimiento de hablantes son los siguientes:

- Estructura de la red `[12 20 11]`.
- Bits de la entrada, función de activación y pesos. Estos valores dependen de la implementación particular del ejemplo que se va a sintetizar. Para observar la variabilidad del sistema con respecto a estos parámetros, se han utilizado diferentes conjuntos de valores:

  8 bits para la entrada, pesos y función de activación.

  10 bits para la entrada, pesos y función de activación.

  12 bits para la entrada, pesos y función de activación.

  12 bits para la entrada y pesos y 10 bits para la función de activación.

  12 bits para la entrada y función de activación y 10 bits para los pesos.
- Matriz con los pesos de la red calculados anteriormente, adaptados al número de bits de los pesos. Así para pesos de 10 bits, los valores de los pesos oscilan entre -511 y 511.
- Nombre de la red del ejemplo que se está ejecutando.
- Función de activación de tipo sigmoidal, con los parámetros siguientes:

  Pendiente 2.

  Centro 0.

  Mínimo -20.

Máximo 20.

Tras cumplimentar todos los parámetros y pulsar el botón de generación, aparece en pantalla la red generada, preparada para ser conectada. En este caso, la red será conectada a un banco de pruebas, cuyo patrón de test está formado por los formantes de cada uno de los hablantes, en orden ascendente. El patrón de test lo componen doce bloques para la lectura de vectores, de forma que cada vector almacene los datos de un determinado formante. Cada elemento de un vector concreto corresponde con un hablante. A cada uno de estos bloques se conecta un bloque del tipo *Gateway In* que convierte el dato de la entrada a punto fijo, para a continuación, mediante un bloque *Reinterpret* conectarlo a cada entrada de la red neuronal.

En el caso de la salida de la red, ésta se conecta a otro bloque *Reinterpret*, para a continuación utilizar un bloque *Gateway Out* que convierta el dato a double. Finalmente se almacena el dato en una variable del *workspace* de Matlab mediante el uso del bloque *To Workspace*. En la figura 5.1 se observa el resultado tras la generación de la red y la colocación del patrón de test.

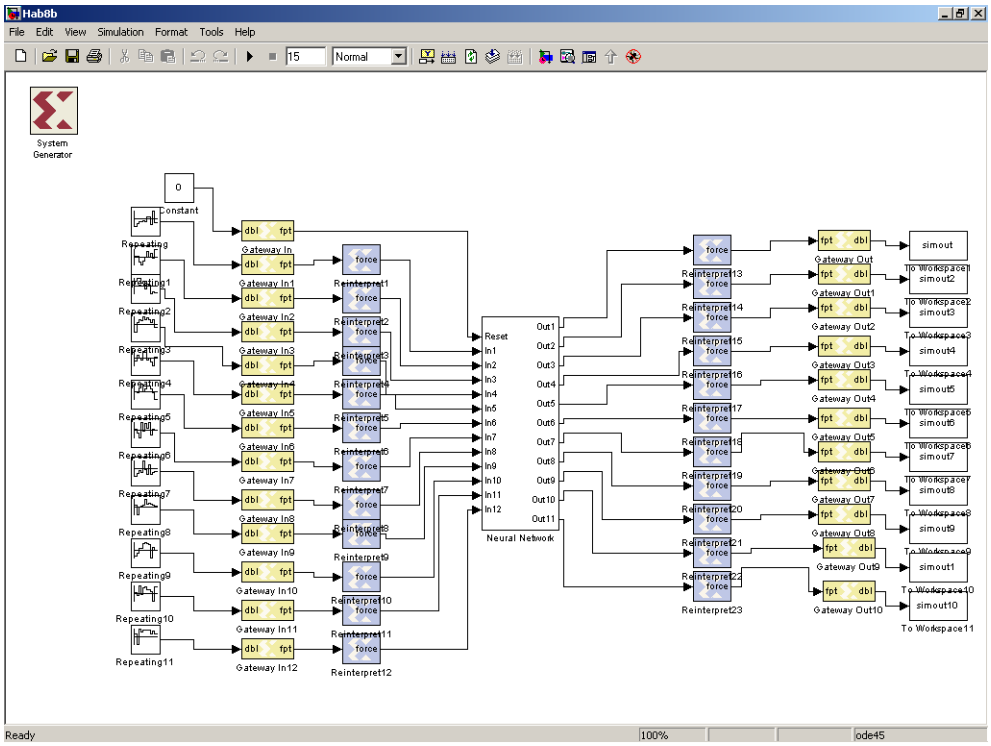


Figura 5.1. Red del ejemplo ya generada y preparada para sintetizar

Tras la realización de las simulaciones, se comprueba el correcto funcionamiento de la red con el patrón de test empleado. Al ejecutar diferentes simulaciones, se observa el hecho de que con sigmoides abruptas, ya sea porque la pendiente sea muy grande o porque sus límites sean muy grandes, se obtiene una clasificación perfecta a partir

de 8 bits tanto para entradas como para pesos, como para resolución de la función de activación.

Posteriormente, una vez verificado el correcto funcionamiento del sistema, puede procederse a realizar la síntesis del circuito correspondiente al Perceptrón configurado. En el caso de este ejemplo, ha sido implementado sobre una Virtex-II 2v3000bf957-4, utilizando las opciones por defecto definidas por *System Generator* al crear el proyecto de ISE.

Todas las implementaciones se hicieron con ISE versión 7.1, para utilizar el mismo sintetizador que en el ejemplo de la próxima sección 5.2, y poder así hacer una comparación más precisa de sus prestaciones.

Las primitivas reconocidas por la herramienta de síntesis son las siguientes, para el caso de la red de 8 bits tanto en pesos, como en el bus de entrada, como en la función de activación:

```
Macro Statistics :
# RAM                               : 31
#      256x8-bit single-port block RAM: 31
# ROMs                              : 1
#      36x11-bit ROM                : 1
# Registers                         : 63
#      1-bit register               : 31
#      19-bit register              : 20
#      20-bit register              : 11
#      6-bit register               : 1
# Multiplexers                      : 93
#      1-bit 4-to-1 multiplexer     : 31
#      8-bit 13-to-1 multiplexer    : 40
#      8-bit 21-to-1 multiplexer    : 22
# Adders/Subtractors                : 32
#      19-bit adder                 : 20
#      20-bit adder                 : 11
#      6-bit adder                  : 1
# Multipliers                       : 31
#      8x8-bit multiplier           : 31
# Comparators                       : 1
#      6-bit comparator less        : 1
```

Para el caso de 10 bits:

```
Macro Statistics :
# RAM                               : 31
#      1024x10-bit single-port block RAM: 31
# ROMs                              : 1
#      36x11-bit ROM                : 1
# Registers                         : 63
#      1-bit register               : 31
```

```
#      23-bit register             : 20
#      24-bit register             : 11
#      6-bit register              : 1
# Multiplexers                     : 93
#      1-bit 4-to-1 multiplexer    : 31
#      10-bit 13-to-1 multiplexer  : 40
#      10-bit 21-to-1 multiplexer  : 22
# Adders/Subtractors               : 32
#      23-bit adder                : 20
#      24-bit adder                : 11
#      6-bit adder                 : 1
# Multipliers                      : 31
#      10x10-bit multiplier        : 31
# Comparators                      : 1
#      6-bit comparator less       : 1
```

En el caso de 12 bits:

```
Macro Statistics :
# RAM                              : 31
#      4096x12-bit single-port block RAM: 31
# ROMs                             : 1
#      36x11-bit ROM               : 1
# Registers                        : 63
#      1-bit register              : 31
#      27-bit register             : 20
#      28-bit register             : 11
#      6-bit register              : 1
# Multiplexers                     : 93
#      1-bit 4-to-1 multiplexer    : 31
#      12-bit 13-to-1 multiplexer  : 40
#      12-bit 21-to-1 multiplexer  : 22
# Adders/Subtractors               : 32
#      27-bit adder                : 20
#      28-bit adder                : 11
#      6-bit adder                 : 1
# Multipliers                      : 31
#      12x12-bit multiplier        : 31
# Comparators                      : 1
#      6-bit comparator less       : 1
```

En todos los casos anteriores se observa que se genera un *block RAM* por neurona, del tamaño que se le indicó, que se emplea para la función de activación, y una ROM, que es utilizada por la unidad de control.

Dentro de los registros, aparecen 3 tamaños diferentes por cada implementación: de 1 bit, empleados como flip-flops; y dos tamaños que dependen de la anchura de los buses, si bien la cantidad de registros de ambos tipos coinciden: 20 y 11. Estos registros sirven para almacenar el resultado del acumulador de las neuronas de las capas 1 y 2, respectivamente.

Algo similar ocurre con los sumadores, si bien en este caso además hay un sumador de 6 bits, utilizado por la unidad de control.

Tal y como se esperaba, se comprueba que hay un multiplicador por neurona. Además hay un comparador de 6 bits, utilizado también por la unidad de control, para reiniciarse al llegar a la última palabra de la ROM.

Aprovechando el hecho de que la función de activación empleada es simétrica, se va a realizar la misma implementación utilizando la opción de simetría, que permite reducir el tamaño de la ROM que almacena los valores de la función de activación a la mitad.

En ese caso, para 10 bits, las primitivas reconocidas son las siguientes:

```
Macro Statistics
# ROMs                          : 32
 36x11-bit ROM                  : 1
 512x10-bit ROM                 : 31
# Multipliers                   : 31
 10x10-bit multiplier           : 31
# Adders/Subtractors            : 94
 10-bit adder                   : 31
 23-bit adder                   : 20
 24-bit adder                   : 11
 6-bit adder                    : 1
 9-bit adder                    : 31
# Registers                     : 94
 1-bit register                 : 31
 10-bit register                : 31
 23-bit register                : 20
 24-bit register                : 11
 6-bit register                 : 1
# Comparators                   : 1
 6-bit comparator less          : 1
# Multiplexers                  : 124
 1-bit 4-to-1 multiplexer       : 31
 10-bit 13-to-1 multiplexer     : 40
 10-bit 21-to-1 multiplexer     : 22
 9-bit 4-to-1 multiplexer       : 31
```

La mayor parte de primitivas reconocidas coinciden con las descritas anteriormente, apareciendo varias primitivas nuevas. En primer lugar se observa el uso de 31 registros de 10 bits — uno por neurona —, que se emplean para almacenar la salida de las

neuronas. Este registro antes no era necesario, ya que se conectaba la ROM a la salida directamente, utilizando la señal de control que indicaba cuando la neurona había terminado sus cálculos como señal de habilitación de lectura de la ROM. En esta implementación, al haber lógica entre la salida y la ROM obliga a usar este registro para almacenar el último valor válido calculado por la neurona.

Por otro lado se observa la aparición de dos sumadores/restadores de 9 y 10 bits, los cuales se van a utilizar únicamente para realizar la operación de complemento a dos en el interior de la neurona. El restador de 9 bits se emplea para cambiar el signo de los bits más significativos del acumulador. Este restador se usa en conjunción con el multiplexor de 9 bits con 4 entradas, que también aparece en esta implementación. Ambas primitivas se emplean con el objetivo de conseguir que los bits más significativos del acumulador, los cuales se utilizan para leer la ROM, sean positivos. Así, si el acumulador contiene un número negativo, la salida del multiplexor será el complemento a dos de sus bits más significativos, mediante el uso del restador; en el caso de que el acumulador sea positivo, a la salida del multiplexor estarán los bits más significativos del acumulador sin ninguna sufrir modificación.

En el caso del restador de 10 bits, éste se emplea a la salida de la ROM, de forma que haga el complemento a dos de su salida cuando el acumulador almacene un valor negativo. Esto se realiza conectando el bit del signo del acumulador a la entrada que indica el tipo de operación que va a realizar el sumador, 0 para indicar una suma y 1 para indicar una resta, y conectando el segundo operando a cero.

bits (pesos, sigmoide)	8, 8	10, 10	12, 12	12, 10	10, 12	2v3000bf957-4
Slices	1.208	1.477	1.745	1.745	1.622	14.336
LUTs	2.170	2.661	3.148	3.148	2.941	28.672
BRAMs	31	31	93	31	93	96
Multiplicadores	31	31	31	31	31	96
Frecuencia máx (Mhz)	44,8	43,34	39,058	41,513	39,939	—
Vel. Muestras (MSa)	1,24	1,20	1,085	1,153	1,109	—

Tabla 5.1. Ocupación y velocidad para varios tamaños

En la tabla 5.1 se muestran los resultados de ocupación de área y velocidad de procesamiento para todos los casos analizados. Las entradas son de 8, 10, 12, 12 y 12 bits de anchura respectivamente. En la tabla 5.2 se observa la misma tabla para el caso de utilizar la opción de simetría en la función de activación. En este caso se observa que no se han utilizado BRAM, de forma que las ROMs que contienen las funciones de activación de las neuronas son implementadas mediante *LUTs*. Esta decisión se debe a que por defecto se da al sintetizador libertad para utilizar *LUTs* o BRAMs en la implementación de las ROMs. Esta política incrementa el uso de recursos de la red en un número importante, llegando a casi triplicarlos en el peor de los casos. No obstante, esta decisión produjo un incremento significativo en la velocidad de la red, debido a la mayor velocidad de las *LUTs* respecto a los BRAM.

De estos resultados obtenidos se pueden sacar ciertas conclusiones:

bits (pesos, sigmoide)	8, 8	10, 10	12, 12	12, 10	10, 12	2v3000bf957-4
Slices	2.009	2.998	4.812	3.314	4.646	14.336
LUTs	3.608	5.373	8.479	5.906	8.325	28.672
BRAMs	—	—	—	—	—	96
Multiplicadores	31	31	31	31	31	96
Frecuencia máx (Mhz)	48,706	47,845	42,256	42,528	43,262	—
Vel. Muestras (MSa)	1,353	1,329	1,174	1,181	1,202	—

Tabla 5.2. Ocupación y velocidad para varios tamaños con la opción de simetría

- Uno de los elementos a tener en cuenta es el número de BRAM de los que dispone la FPGA, ya que es el principal limitador del tamaño de la red, en el caso de que sean utilizadas. Como se puede observar en la tabla 5.1, el número de BRAM necesarios depende del tamaño de la función de activación, no siendo tan importante la unidad de control, ya que, aunque en este caso es implementada como *LUTs*, su tamaño es pequeño y no varía al modificar los parámetros de la red, para una misma estructura.
- Al incrementar el tamaño de la red, se reduce la velocidad de ésta, debido a su mayor gasto en lógica, y por tanto, se necesitan rutas más largas para alcanzar toda la lógica.

Se observa el hecho curioso de que una reducción de los bits de la función de activación no produce una reducción en el tamaño de la red, únicamente una reducción del número de BRAMs, debido a que la lógica restante es idéntica, únicamente se reducen las líneas del bus de direcciones de la RAM. Sin embargo, sí aumenta la velocidad, debido a que las rutas son más cortas, al tener que conectar menos componentes

5.2. Implementación como coprocesador

En el supuesto de utilizar nuestra arquitectura de red neuronal como periférico (versión HANN), se van a realizar dos pruebas diferentes. Por un lado se van a obtener los resultados de implementación de la red neuronal en solitario, para a continuación obtener los resultados de la red integrada en una arquitectura de cómputo estándar basada en microprocesador.

5.2.1. Implementación del periférico en solitario

En este primer caso, la metodología de diseño consiste simplemente en generar un proyecto en ISE e incluir los ficheros VHDL de HANN. Tras ello, se modifican los genéricos con los valores de la red deseados.

La FPGA utilizada para la proyección del circuito será la misma que en el caso anterior, con las opciones por defecto del sintetizador, utilizando igualmente la versión 7.1i de ISE, para poder establecer una comparativa con las máximas garantías.

En este caso, las construcciones reconocidas por el compilador son las siguientes, para el modelo de 8 bits:

```
Macro Statistics :
# RAM                              : 31
#      256x8-bit single-port block RAM: 31
# Registers                        : 569
#      1-bit register              : 44
#      19-bit register             : 20
#      20-bit register             : 11
#      4-bit register              : 2
#      8-bit register              : 492
# Multiplexers                     : 94
#      1-bit 4-to-1 multiplexer    : 32
#      8-bit 13-to-1 multiplexer   : 40
#      8-bit 21-to-1 multiplexer   : 22
# Adders/Subtractors               : 31
#      19-bit adder                : 20
#      20-bit adder                : 11
# Multipliers                      : 31
#      8x8-bit multiplier          : 31
# Comparators                      : 3
#      5-bit comparator less       : 1
#      6-bit comparator greatequal : 1
#      6-bit comparator less       : 1
```

Para el caso de 10 bits:

```
Macro Statistics :
# RAM                              : 31
#      1024x10-bit single-port block RAM: 31
# Registers                        : 571
#      1-bit register              : 46
#      10-bit register             : 492
#      23-bit register             : 20
#      24-bit register             : 11
#      4-bit register              : 2
# Multiplexers                     : 94
#      1-bit 4-to-1 multiplexer    : 32
#      10-bit 13-to-1 multiplexer  : 40
#      10-bit 21-to-1 multiplexer  : 22
# Adders/Subtractors               : 31
#      23-bit adder                : 20
```

```
#      24-bit adder                : 11
# Multipliers                      : 31
#      10x10-bit multiplier        : 31
# Comparators                      : 3
#      5-bit comparator less       : 1
#      6-bit comparator greatequal : 1
#      6-bit comparator less       : 1
```

Para 12 bits:

```
Macro Statistics :
# RAM                              : 31
#      4096x12-bit single-port block RAM: 31
# Registers                        : 573
#      1-bit register              : 48
#      12-bit register             : 492
#      27-bit register             : 20
#      28-bit register             : 11
#      4-bit register              : 2
# Multiplexers                     : 94
#      1-bit 4-to-1 multiplexer    : 32
#      12-bit 13-to-1 multiplexer  : 40
#      12-bit 21-to-1 multiplexer  : 22
# Adders/Subtractors               : 31
#      27-bit adder                : 20
#      28-bit adder                : 11
# Multipliers                      : 31
#      12x12-bit multiplier        : 31
# Comparators                      : 3
#      5-bit comparator less       : 1
#      6-bit comparator greatequal : 1
#      6-bit comparator less       : 1
```

Se observan 31 *BRAM*, uno para cada función de activación, tal y como se esperaba. Con la arquitectura HANN, el número de registros es mayor que en el caso anterior, debido a que los pesos no son constantes, y por tanto no son cableadas, almacenándose en registros. Se observan 20 registros que serán los acumuladores de la primera capa, y 11 que serán los de la segunda. Además se observan 492 registros, 491 empleados para contener los pesos y uno para contener el peso enviado a través del bus OPB, y 2 registros de 4 bits, que contendrán el contador de la unidad de control. El resto de registros tiene que ver, además de con las neuronas, con la lógica del bus OPB y con lógica de control.

Además se observan por un lado 20 sumadores y 11 por otro, para las neuronas de la primera capa y de la segunda, respectivamente, al igual que en el caso anterior.

Igualmente, se observan 31 multiplicadores, uno por neurona y 3 comparadores, para las dos unidades de control (los dos comparadores de menor) y otro para comprobar en la última capa cuándo se ha llegado al final para reiniciar las unidades de control.

Las primitivas reconocidas al sintetizar activando la opción de reducir la ROM de la función de activación simétrica, para el caso de 10 bits son las siguientes:

```
Macro Statistics :
# ROMs                          : 31
 512x10-bit ROM                 : 31
# Multipliers                   : 31
 10x10-bit multiplier           : 31
# Adders/Subtractors            : 93
 10-bit adder                   : 31
 23-bit adder                   : 20
 24-bit adder                   : 11
 9-bit adder                    : 31
# Counters                      : 2
 4-bit up counter               : 1
 5-bit up counter               : 1
# Registers                     : 600
 1-bit register                 : 46
 10-bit register                : 523
 23-bit register                : 20
 24-bit register                : 11
# Comparators                   : 3
 5-bit comparator less          : 1
 6-bit comparator greatequal    : 1
 6-bit comparator less          : 1
# Multiplexers                  : 125
 1-bit 4-to-1 multiplexer       : 32
 10-bit 13-to-1 multiplexer     : 40
 10-bit 21-to-1 multiplexer     : 22
 9-bit 4-to-1 multiplexer       : 31
```

bits (pesos, sigmoide)	8, 8	10, 10	12, 12	12, 10	10, 12	2v3000bf957-4
Slices	4.665	5.805	6.984	6.984	6.036	14.336
LUTs	4.363	5.390	6.418	6.418	5.711	28.672
BRAMs	31	31	93	31	93	96
Multiplicadores	31	31	31	31	31	96
Frecuencia máx (Mhz)	46,281	43,894	39,981	41,15	40,005	—
Vel. Muestras (MSa)	1,285	1,219	1,106	1,143	1,112	—

Tabla 5.3. Ocupación y velocidad del periférico para varios tamaños

En este caso, al igual que en la implementación con la opción de simetría de HANNA, aparecen nuevas primitivas, que coinciden con las aparecidas en aquélla. Su explicación es la misma, por lo que no se van comentar.

bits (pesos, sigmoide)	8, 8	10, 10	12, 12	12, 10	10, 12	2v3000bf957-4
Slices	5.193	6.957	9.551	8.289	8.599	14.336
LUTs	5.364	7.558	11.125	8.953	10.446	28.672
BRAMs	—	—	—	—	–	96
Multiplicadores	31	31	31	31	31	96
Frecuencia máx (Mhz)	49,692	48,454	44,825	44,849	46,145	—
Vel. Muestras (MSa)	1,380	1,346	1,245	1,245	1,282	—

Tabla 5.4. Ocupación y velocidad del periférico para varios tamaños con la opción de simetría

En la tabla 5.3 se observan los resultados de ocupación y velocidad del periférico en solitario. Las entradas son iguales que en el caso de la red generada por HANNA — 8, 10, 12, 12 y 12 respectivamente —. En ella se observa que hay un consumo apreciable de *slices* y *LUTs*, debido a la necesidad de registros para contener los pesos y el cambio en la unidad de control. El consumo de multiplicadores y *BRAM* es el esperado.

En la tabla 5.4 se resumen los resultados de ocupación y velocidad para la función de activación simétrica. Se puede observar el incremento del número de *slices* y *LUTs* respecto a la tabla anterior, debido a que el sintetizador ha decidido implementar las ROM en *LUTs* en lugar de utilizar los *BRAM*, al igual que en el ejemplo con HANNA. Esta política ha provocado un incremento significativo en la velocidad máxima de la red.

5.2.2. Implementación del periférico en un sistema empotrado

A continuación se va a implementar el periférico en un sistema empotrado, el cual, constará de los siguientes subsistemas, todos ellos con la configuración por defecto, salvo que se diga lo contrario:

- El microprocesador MicroBlaze, con la unidad de coma flotante activada y la multiplicación realizada por hardware.
- Dos controladoras del bus LMB, una para las instrucciones y otra para los datos.
- Una RAM de 64 KB en el bus LMB, que contendrá el código y los datos.
- Una UART, implementada mediante el periférico *UARTLite*.
- La red neuronal con arquitectura tipo HANN.

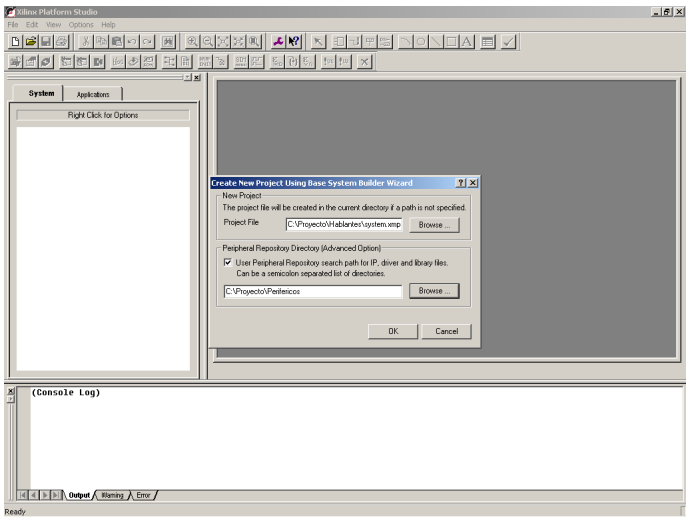


Figura 5.2. Creación de un proyecto con Platform Studio

El procedimiento para realizar el ejemplo fue generar un proyecto de EDK con los subsistemas comentados, añadiendo el repositorio donde se encuentra la red neuronal, y conectarlos, definiendo todos los genéricos, teniendo en cuenta la peculiaridad de la red neuronal. En este caso, la metodología es un tanto diferente, debido a limitaciones de la herramienta Platform Studio, que no puede asignar valores a genéricos definidos por el usuario, como ya se comentó. A continuación se describe este procedimiento en detalle.

En primer lugar se genera un proyecto nuevo con la herramienta Platform Studio. Tras seleccionar dicha opción, en la primera ventana que aparece, la herramienta solicita el nombre del proyecto y la ruta a los repositorios con periféricos del usuario, tal y como se observa en la figura 5.2. En este segundo parámetro se indicó la ruta al repositorio que incluía el periférico de la red neuronal. El resto de opciones que son necesarias al crear el proyecto tienen relación con los parámetros de la placa y la FPGA que se empleará para la síntesis del proyecto, y los periféricos básicos que se añadirán al sistema, como, en este caso, la UART, que se empleará como entrada y salida estándar.

Una vez creado el proyecto, se procede a añadir los periféricos restantes. Para ello se utiliza la opción *Add/Edit Cores* del menú contextual de *System BSP*, tal y como se muestra en la figura 5.3. *System BSP* es un árbol, que de manera gráfica, muestra todos los periféricos y buses del sistema empotrado, permitiendo, al expandir sus ramas, un fácil acceso y configuración de sus parámetros.

Tras seleccionar la mencionada opción, aparece la ventana de configuración de las especificaciones del sistema empotrado. Esta ventana se encarga de la mayor parte de la configuración de los periféricos del sistema, obviando los parámetros del dispositivo sobre el que se sintetizará. En la primera pestaña de la ventana, que se muestra en la figura 5.4 se nos ofrece la opción de añadir o eliminar periféricos del sistema. En nuestro caso, únicamente se incluirá la red neuronal, que aparece en la figura seleccionada de

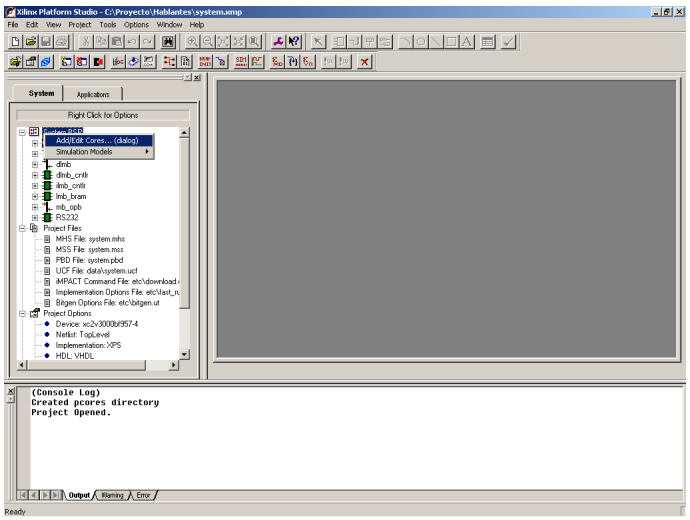


Figura 5.3. Opción utilizada para añadir nuevos periféricos

la lista de periféricos localizados por la herramienta.

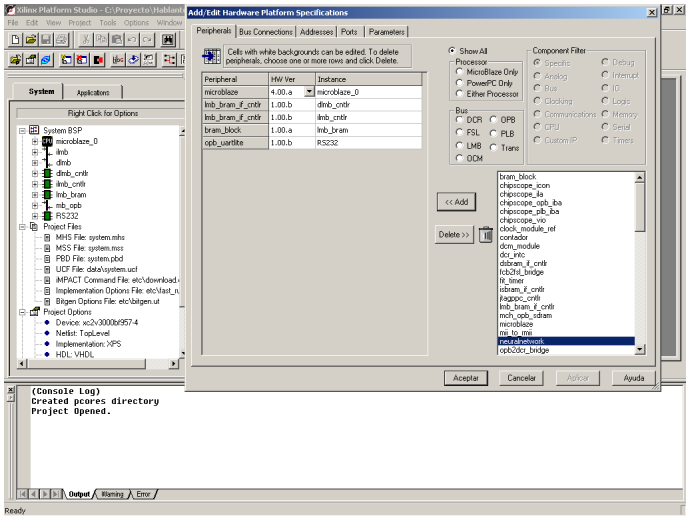


Figura 5.4. Ventana encargada de la gestión de los periféricos

Después de añadir el periférico, se accede a la siguiente pestaña de la ventana, encargada de las conexiones de los elementos del sistema a los distintos buses. Para conectar un subsistema a un bus, hay que pulsar con el botón izquierdo sobre el cuadrado correspondiente. Pulsando varias veces se permite cambiar su comportamiento en el bus, pudiendo ser maestro, esclavo o maestro-esclavo, si el subsistema tiene implementadas dichas capacidades. En nuestro caso, conectaremos la red neuronal como esclavo del bus OPB, tal y como se observa en la figura 5.5.

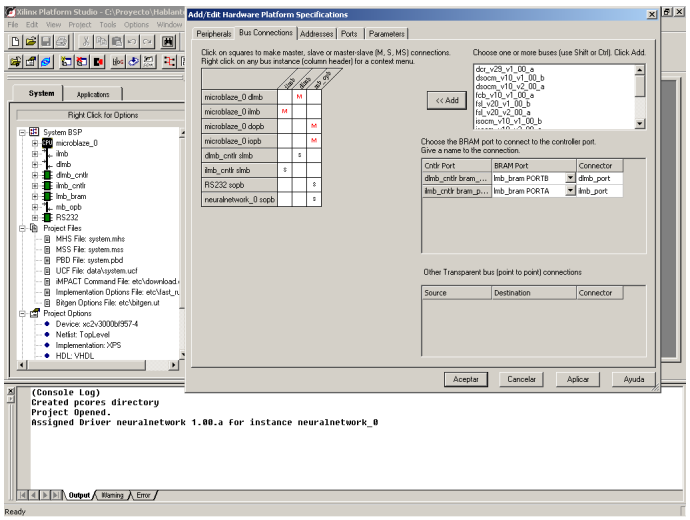


Figura 5.5. Ventana encargada de la gestión de la conexión de los elementos del sistema a los buses

La tercera pestaña se encarga de la gestión del mapa de memoria del sistema empotrado, mediante la asignación de rangos de direcciones a los distintos periféricos del sistema, tal y como se muestra en la figura 5.6. En este ejemplo, dicha gestión es algo sencillo, ya que el sistema lo componen muy pocos elementos.

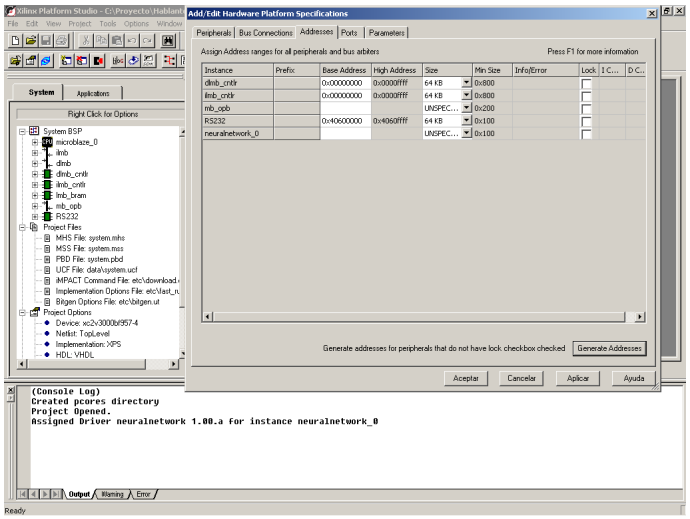


Figura 5.6. Ventana encargada del mapa de memorias del sistema

La última pestaña de la ventana se encarga de la configuración de los genéricos de los diferentes periféricos, mediante la edición de varios parámetros. En la figura 5.7 se observa su aspecto. Como se puede ver, a la derecha aparece una lista con todos los parámetros del periférico y sus valores por defecto. Para modificar estos parámetros

es necesario seleccionarlos y pulsar el botón *Add*, de forma que se añadan a la lista de parámetros de la izquierda, y sustituir en dicha lista los valores por defecto por los deseados.

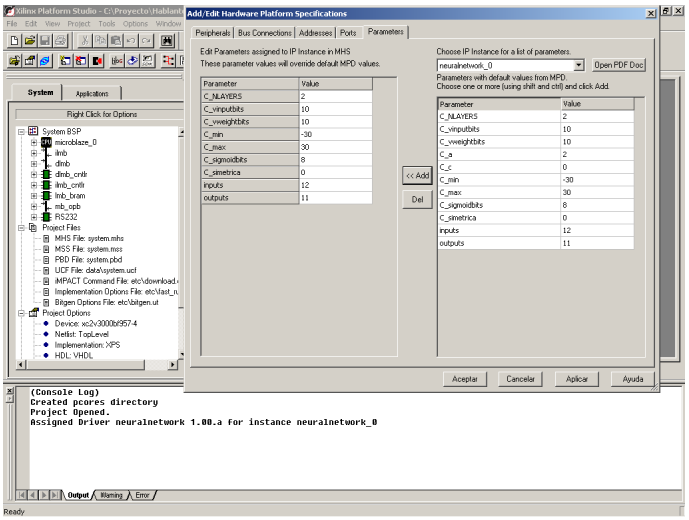


Figura 5.7. Ventana encargada de la configuración de los parámetros de los periféricos

Como se puede observar, en los parámetros de la red neuronal no aparece el que hace referencia a la estructura de la red neuronal. Esto se debe, como ya se explicó anteriormente, a que está definido con un tipo de datos definido por el usuario. Por tanto, dicho genérico — `C_LAYERSTRUC` — se debe modificar directamente en el código VHDL del periférico, que se encuentra en *Repositorio*\MyProcessorIPLib\pcores\neural_network_v2_00_a\hdl\vhdl\neuralnetwork.vhd.

Finalmente, la cuarta pestaña se encarga del conexionado de los subsistemas del sistema empotrado, como se observa en la figura 5.8. En la parte inferior se encuentran las conexiones internas, aquéllas que no son accesibles desde el exterior de la red, como las conexiones a los buses del sistema. En la parte superior se encuentran las conexiones externas, que son aquellas que son directamente accesibles a través de los pines de la FPGA.

En este paso, en primer lugar se conectarán aquellos puertos de la red que faltan por conectar, debido a que no tienen una conexión por defecto. Las conexiones por defecto se producen al conectar el periférico a un determinado bus. En el caso de la red neuronal, el reloj — `OPB_Clk` — y la entrada y salida de la red — `dinput` y `doutput` respectivamente — son los únicos puertos sin conexión por defecto. Para nuestro ejemplo, el reloj se conectará a la linea del reloj del sistema, por defecto `sys_clk_s`, y las conexiones a la entrada y salida de la red se harán externas, de forma que puedan comunicarse con el exterior. Para hacer las conexiones externas, tras haber realizado las conexiones como internas, se seleccionan de dicha lista y se le pulsa al botón *Make External*. Tras ello, aparecen en la lista superior, tal y como se ve en la figura 5.8. Por último es necesario definir el rango de los puertos, ya que al hacerlos externos, la herramienta ignora el

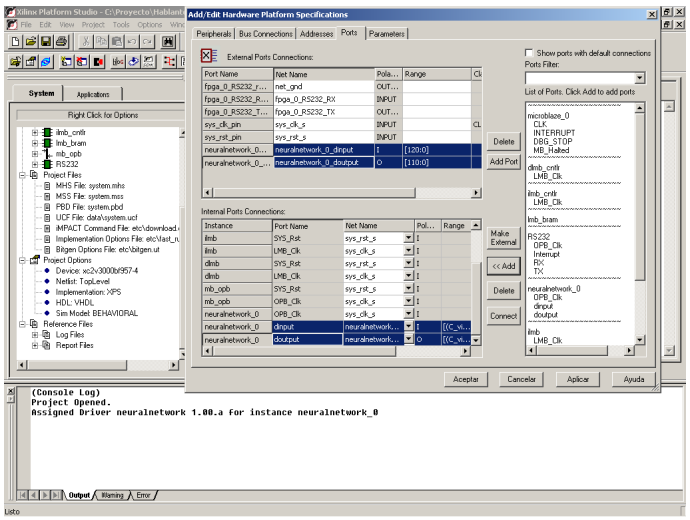


Figura 5.8. Aspecto tras realizar las conexiones externas

rango indicado en la implementación del periférico.

Tras finalizar estos pasos, se aceptan los cambios. En este momento, se ha completado la configuración de los subsistemas que componen el sistema empotrado, por lo que solo resta el desarrollo del software para tener completo el sistema empotrado.

El software que ejecutará el microprocesador se encargará de realizar el entrenamiento de la red con la implementación del BackProgapation suministrada, para a continuación convertir los pesos calculados a enteros y enviarlos al periférico para que los utilice. Además, el software simulará la ejecución de la red con los pesos calculados y mostrará los valores obtenidos para la salida de la red utilizando la UART como entrada/salida estándar para comunicarse con un PC externo mediante un terminal. El código desarrollado se encuentra en el Anexo A.

Las opciones de compilación del software son

```
1 mb-gcc -O2 menu.c -o Menu/executable.elf \
2 -Wl,-defsym -Wl,_STACK_SIZE=0x800 -mno-xl-soft-mul -mhard-float -g
      -I./microblaze_0/include/ -L./microblaze_0/lib/ \
```

Se observa que se ha utilizado un tamaño de pila de 0x800 bytes, debido a que algunas funciones como la del Backpropagation requieren un tamaño en pila considerable, por el gran número de argumentos que necesitan.

El tamaño del ejecutable es de 31.206 bytes, quedando aproximadamente la mitad disponible del bloque de 64 KB definido para la memoria de programa.

No se hacen referencias a las macros generadas por el compilador, ya que serán, para el caso de la red neuronal, los mismos que los vistos en el subapartado anterior, ya que el código VHDL es el mismo.

El tamaño del sistema completo se muestra en las tablas 5.5 y 5.6 para los diferentes valores de cuantización considerados. En la primera de ellas se observan los resultados de la implementación del sistema empotrado con función de activación completa, en

bits (pesos, sigmoide)	8, 8	10, 10	12, 12	12, 10	10, 12	2v3000bf957-4
Slices	6.051	7.188	—	8.344	—	14.336
LUTs	6.518	7.547	—	8.577	—	28.672
BRAMs	63	63	—	63	—	96
Multiplicadores	38	38	—	38	—	96
Frecuencia máx (Mhz)	40,079	40,006	—	40,099	—	—
Vel. Muestras (MSa)	1,113	1,111	—	1,113	—	—

Tabla 5.5. Ocupación y velocidad del sistema empotrado para varios tamaños

donde aparecen en blanco las implementaciones que requieren 12 bits en la función de activación. Eso se debe a que no se puede implementar por la falta de BRAM en la FPGA. Sin embargo, este problema no aparece en la implementación de la tabla 5.6, que aprovecha la simetría de la función de activación, ya que también en esta ocasión el sintetizador decidió implementar las ROMs en *LUTs.*

bits (pesos, sigmoide)	8, 8	10, 10	12, 12	12, 10	10, 12	2v3000bf957-4
Slices	6.550	8.287	11.300	10.359	9.672	14.336
LUTs	7.525	9.690	14.098	13.418	11.177	28.672
BRAMs	32	32	32	32	32	96
Multiplicadores	38	38	38	38	38	96
Frecuencia máx (Mhz)	40,626	40,422	40,174	40,561	40,271	—
Vel. Muestras (MSa)	1,128	1,123	1,115	1,127	1,118	—

Tabla 5.6. Ocupación y velocidad del sistema empotrado para varios tamaños con la opción de simetría

Como detalles se observa que la velocidad máxima del sistema es de aproximadamente 40MHz, que coincide aproximadamente con la de la red neuronal en solitario, por lo que se concluye que la red limita la velocidad del sistema.

5.3. Conclusiones del ejemplo

La principal conclusión que se puede extraer de la comparativa entre la implementación software y las arquitecturas hardware de las redes neuronales es el aumento de velocidad de las últimas con respecto a las primeras. Este aumento, de entre 2 y 3 órdenes de magnitud — de aproximadamente 2 kSa a un valor superior a 1MSa —, justificaría el uso de cualquiera de las arquitecturas propuestas en aplicaciones que requieran de una mayor velocidad de proceso que la que puede suministrar un computador convencional.

Se debe destacar que, de entre las dos aproximaciones hardware para la implementación de redes neuronales, la elección de una u otra depende de la aplicación que se

vaya a desarrollar: el empleo de HANNA es apropiado para la creación hardware de redes ya diseñadas y probadas, con el objetivo de conseguir un incremento de velocidad, sin necesidad de tener grandes conocimientos de arquitectura de computadores, y de forma sencilla, mediante el uso de una simple interfaz gráfica; por otro lado, la utilización del periférico está pensada para situaciones en las que deba ser conectado en un sistema empotrado junto con otros periféricos y subsistemas, lo que permite utilizarlo en aplicaciones donde un ordenador convencional no es apropiado o no alcanza las prestaciones de cálculo suficientes, pero que además soporta restricciones de coste, tamaño o consumo.

La arquitectura HANN implica además que para su uso se posea un mayor conocimiento de FPGAs y sistemas empotrados, proporcionando a cambio una mayor flexibilidad.

En cuanto a prestaciones de las diferentes implementaciones, se observa una ligera diferencia de velocidad a favor del periférico debido a que el uso de los bloques de *System Generator* introduce una pequeña pérdida de rendimiento respecto a la implementación directa en VHDL. Por otro lado, la ocupación de área es menor para el caso de las redes generadas por HANNA, debido a la necesidad del periférico de alojar los pesos en registros.

Capítulo 6

Conclusiones y ampliaciones

6.1. Conclusiones

En este Proyecto se ha desarrollado una arquitectura optimizada sobre hardware para el diseño de algoritmos de redes neuronales, en particular del tipo Perceptrón Multicapa y similares. Esta arquitectura permite una fácil automatización del diseño de estas redes, debido a que básicamente consiste en la interconexión de un conjunto de neuronas elementales, junto a las señales de control generadas por una unidad de control. El uso de redes neuronales implementadas en hardware resulta de gran interés, ya que permite un mayor grado de paralelismo en la ejecución del algoritmo, frente al uso de computadores convencionales, lo que hace presuponer una mayor velocidad de cálculo.

Nuestra arquitectura presenta como ventaja que está diseñada a nivel de neurona, en lugar de a nivel de a nivel de transferencia de registros. El nivel de transferencia de registros es la descripción de un circuito electrónico en función del flujo de datos entre registros, especificando dónde y qué información es almacenada, y como progresa por el circuito durante su funcionamiento. Utilizar como unidad básica la neurona facilita la comprensión de los diseños generados sin necesidad de grandes conocimientos de sistemas electrónicos digitales.

Además se ha desarrollado una herramienta — HANNA — para la automatización de la generación de este tipo de redes neuronales sobre circuitos programables, que permite su uso sin necesidad de grandes conocimientos de arquitectura de computadores y FPGAs. Esta herramienta desarrollada sobre Matlab permite la integración de las redes generadas en el flujo convencional de diseño de sistemas de Simulink, permitiendo aprovechar bancos de pruebas ya diseñados previamente sobre dicha herramienta, facilitando la comparación de prestaciones de la implementación de la misma red sobre filosofías de diseño diferentes, o incluso su uso en sistemas mixtos.

De forma conjunta a HANNA, se ha desarrollado un periférico que implementa la arquitectura diseñada, para su uso en sistemas empotrados — HANN —. El periférico posee una interfaz compatible con el estándar del bus OPB, por lo que puede conectarse a cualquier sistema de cómputo que soporte dicha especificación, lo que aumenta su versatilidad.

La utilización como periférico de un acelerador por hardware de redes neuronales

redunda en un incremento de la velocidad del sistema empotrado, ya que libera al procesador de dichos cálculos, y en una mayor velocidad en la ejecución del algoritmo de la red neuronal. Además, al estar implementado como periférico puede ser empleado en sistemas basados en cualquier tipo de procesador, o llegar al extremo de no requerir un procesador para su funcionamiento, dándole una mayor flexibilidad a su uso.

Finalmente se ha verificado el correcto funcionamiento de la metodología de diseño y las arquitecturas propuestas en ambas aproximaciones, aplicándolas a la solución de un problema específico de reconocimiento de hablantes, confirmando la hipótesis inicial que sugería que la implementación sobre hardware es más rápida que la realizada sobre un ordenador personal ejecutando el algoritmo convencional. Los excelentes resultados obtenidos muestran, para la aplicación seleccionada, un incremento en la velocidad de procesamiento con respecto a un computador tipo PC de entre 2 y 3 órdenes de magnitud.

6.2. Futuros trabajos

Una propuesta interesante consistiría en incluir la posibilidad de poder asignar distintas funciones de activación en cada neurona de la red, lo que llevaría a un replanteamiento de la actual arquitectura, ya que la arquitectura implementada no contempla esta opción, utilizando los mismos parámetros en todas las neuronas. Sin embargo, esto plantea la dificultad del elevado número de parámetros que serían necesarios para configurar la red.

También sería conveniente modificar la arquitectura para conseguir implementar otro tipo de redes neuronales diferentes al Perceptrón Multicapa, como funciones de base radial, lo que le daría una mayor flexibilidad a la arquitectura.

Apéndice A

Software utilizado en la aplicación al Reconocimiento de Hablantes

```
#include "xparameters.h"

#include "xutil.h"

#include "neuralnetwork.h"


int main (void) {

  long i,j;

    long estructura[]={12,20,11};


        double pesos[] = {
-0.432565, -1.665584, 0.125332, 0.287676, -1.146471, 1.190915,
1.189164, -0.037633, 0.327292, 0.174639, -0.186709, 0.725791,
-0.588317,
2.183186, -0.136396, 0.113931, 1.066768, 0.059281, -0.095648,
-0.832349, 0.294411, -1.336182, 0.714325, 1.623562, -0.691776,
0.857997,
1.254001, -1.593730, -1.440964, 0.571148, -0.399886, 0.689997,
0.815622, 0.711908, 1.290250, 0.668601, 1.190838, -1.202457,
-0.019790,
-0.156717, -1.604086, 0.257304, -1.056473, 1.415141, -0.805090,
0.528743, 0.219321, -0.921902, -2.170674, -0.059188, -1.010634,
0.614463,
0.507741, 1.692430, 0.591283, -0.643595, 0.380337, -1.009116,
-0.019511, -0.048221, 0.000043, -0.317859, 1.095004, -1.873990,
0.428183,
0.895638, 0.730957, 0.577857, 0.040314, 0.677089, 0.568900, -0.255645,
-0.377469, -0.295887, -1.475135, -0.234004, 0.118445, 0.314809,
1.443508, -0.350975, 0.623234, 0.799049, 0.940890, -0.992092,
0.212035, 0.237882, -1.007763, -0.742045, 1.082295, -0.131500,
0.389880,
0.087987, -0.635465, -0.559573, 0.443653, -0.949904, 0.781182,
0.568961, -0.821714, -0.265607, -1.187777, -2.202321, 0.986337,
```

```
38  -0.518635,
39  0.327368, 0.234057, 0.021466, -1.003944, -0.947146, -0.374429,
40  -1.185886, -1.055903, 1.472480, 0.055744, -1.217317, -0.041227,
41  -1.128344,
42  -1.349278, -0.261102, 0.953465, 0.128644, 0.656468, -1.167819,
43  -0.460605, -0.262440, -1.213152, -1.319437, 0.931218, 0.011245,
44  -0.645146,
45  0.805729, 0.231626, -0.989760, 1.339586, 0.289502, 1.478917, 1.138028,
46  -0.684139, -1.291936, -0.072926, -0.330599, -0.843628, 0.497770,
47  1.488490, -0.546476, -0.846758, -0.246337, 0.663024, -0.854197,
48  -1.201315, -0.119869, -0.065294, 0.485296, -0.595491, -0.149668,
49  -0.434752,
50  -0.079330, 1.535152, -0.606483, -1.347363, 0.469383, -0.903567,
51  0.035880, -0.627531, 0.535398, 0.552884, -0.203690, -2.054325,
52  0.132561,
53  1.592941, 1.018412, -1.580402, -0.078662, -0.681657, -1.024553,
54  -1.234353, 0.288807, -0.429303, 0.055801, -0.367874, -0.464973,
55  0.370961,
56  0.728283, 2.112160, -1.357298, -1.022610, 1.037834, -0.389800,
57  -1.381266, 0.315543, 1.553243, 0.707894, 1.957385, 0.504542, 1.864529,
58  -0.339812, -1.139779, -0.211123, 1.190245, -1.116209, 0.635274,
59  -0.601412, 0.551185, -1.099840, 0.085991, -2.004563, -0.493088,
60  0.462048,
61  -0.321005, 1.236556, -0.631280, -2.325211, -1.231637, 1.055648,
62  -0.113224, 0.379224, 0.944200, -2.120427, -0.644679, -0.704302,
63  -1.018137,
64  -0.182082, 1.521013, -0.038439, 1.227448, -0.696205, 0.007524,
65  -0.782893, 0.586939, -0.251207, 0.480136, 0.668155, -0.078321,
66  0.889173,
67  2.309287, 0.524639, -0.011787, 0.913141, 0.055941, -1.107070,
68  0.485498, -0.005005, -0.276218, 1.276452, 1.863401, -0.522559,
69  0.103424,
70  -0.807649, 0.680439, -2.364590, 0.990115, 0.218899, 0.261662,
71  1.213444, -0.274667, -0.133134, -1.270500, -1.663606, -0.703554,
72  0.280880,
73
74  -0.541209, -1.333531, 1.072686, -0.712085, -0.011286, -0.000817,
75  -0.249436, 0.396575, -0.264013, -1.664011, -1.028975, 0.243095,
76  -1.256590, -0.347183, -0.941372, -1.174560, -1.021142, -0.401667,
77  0.173666, -0.116118, 1.064119,
78  -0.245386, -1.517539, 0.009734, 0.071373, 0.316536, 0.499826,
79  1.278084, -0.547816, 0.260808, -0.013177, -0.580264, 2.136308,
80  -0.257617, -1.409528, 1.770101, 0.325546, -1.119040, 0.620350,
81  1.269782, -0.896043, 0.135175,
82  -0.139040, -1.163395, 1.183720, -0.015430, 0.536219, -0.716429,
83  -0.655559, 0.314363, 0.106814, 1.848216, -0.275106, 2.212554,
84  1.508526, -1.945079, -1.680543, -0.573534, -0.185817, 0.008934,
85  0.836950, -0.722271, -0.721490,
86  -0.201181, -0.020464, 0.278890, 1.058295, 0.621673, -1.750615,
87  0.697348, 0.811486, 0.636345, 1.310080, 0.327098, -0.672993,
88  -0.149327, -2.449018, 0.473286, 0.116946, -0.591104, -0.654708,
89  -1.080662, -0.047731, 0.379345,
```

```
90  -0.330361, -0.499898, -0.035979, -0.174760, -0.957265, 1.292548,
91  0.440910, 1.280941, -0.497730, -1.118717, 0.807650, 0.041200,
92  -0.756209, -0.089129, -2.008850, 1.083918, -0.981191, -0.688489,
93  1.339479, -0.909243, -0.412858,
94  -0.506163, 1.619748, 0.080901, -1.081056, -1.124518, 1.735676,
95  1.937459, 1.635068, -1.255940, -0.213538, -0.198932, 0.307499,
96  -0.572325, -0.977648, -0.446809, 1.082092, 2.372648, 0.229288,
97  -0.266623, 0.701672, -0.487590,
98  1.862480, 1.106851, -1.227566, -0.669885, 1.340929, 0.388083,
99  0.393059, -1.707334, 0.227859, 0.685633, -0.636790, -1.002606,
100 -0.185621, -1.054033, -0.071539, 0.279198, 1.373275, 0.179841,
101 -0.542017, 1.634191, 0.825215,
102 0.230761, 0.671634, -0.508078, 0.856352, 0.268503, 0.624975,
103 -1.047338, 1.535670, 0.434426, -1.917136, 0.469940, 1.274351,
104 0.638542, 1.380782, 1.319843, -0.909429, -2.305605, 1.788730,
105 0.390798, 0.020324, -0.405977,
106 -1.534895, 0.221373, -1.374479, -0.839286, -0.208643, 0.755913,
107 0.375734, -1.345413, 1.481876, 0.032736, 1.870453, -1.208991,
108 -0.782632, -0.767299, -0.107200, -0.977057, -0.963988, -2.379172,
109 -0.838188, 0.257346, -0.183834,
110 -0.167615, -0.116989, 0.168488, -0.501206, -0.705076, 0.508165,
111 -0.420922, 0.229133, -0.959497, -0.146043, 0.744538, -0.890496,
112 0.139062, -0.236144, -0.075459, -0.358572, -2.077635, -0.143546,
113 1.393341, 0.651804, -0.377134,
114 -0.661443, 0.248958, -0.383516, -0.528480, 0.055388, 1.253769,
115 -2.520004, 0.584856, -1.008064, 0.944285, -2.423957, -0.223831,
116 0.058070, -0.424614, -0.202918, -1.513077, -1.126352, -0.815002,
117 0.366614, -0.586107, 1.537409
118
119                                         };
120
121       double mu = 0.05;
122
123        long pesosl[491];
124        long templ[31];
125
126        long salidasl[512];
127        double temp[31];
128        double entradas;
129        double erroresO = 1e5;
130
131        double test[] =    {
132        -1.0000, 0.1164, 0.5521, -1.0000, -0.4336, -0.0418, 0.0658,
             --1.0000,
133 0.2612, -0.8008, 0.0677, 0.2828,
134        -0.7290, 0.0822, 1.0000, 0.2665, 0.2981, -0.1432, -0.4807,
             --0.6557,
135 -1.0000, -0.3870, -0.3442, -1.0000,
136        -0.4255, -0.6353, 0.9445, 0.5512, -0.0163, 0.4621, -1.0000,
             --0.3244,
137 -0.1039, 0.2120, 1.0000, 1.0000,
```

```
138         -0.4859, -1.0000, 0.3342, 0.4460, -0.3356, 1.0000, 0.9713,
               --0.4978,
139 0.0841, 0.2482, -0.1283, 0.5452,
140         -0.2518, -0.5045, 0.5976, 1.0000, 0.8699, -0.0527, -0.1626,
               --1.0000,
141 0.1980, 0.8450, 0.7389, 0.8171,
142         -0.4175, -0.0262, -0.4467, 0.6251, 1.0000, 0.3602, 0.8174,
               --0.1712,
143 1.0000, 1.0000, 0.8150, 0.8161,
144         0.1859, 0.7429, 0.2192, 0.8467, -0.3062, 0.7341, 0.2139, -0.8031,
145 0.3032, 0.8682, 0.4789, 0.8704,
146         0.2462, -0.0033, -0.0659, 0.5374, -0.0474, 0.6867, 1.0000, 0.4861,
147 0.5173, 0.7420, 0.4021, 0.8117,
148         -0.8326, 0.4011, -1.0000, 0.2048, -0.4878, 0.3938, -0.7328,
               --0.7737,
149 0.3011, -1.0000, 0.1855, 0.3762,
150         1.0000, -0.1399, -0.8179, 0.5383, -1.0000, -0.1863, 0.0231,
               --0.5912,
151 0.2634, 0.3805, -1.0000, 0.6954,
152         -0.4903, 1.0000, -0.8816, -0.3893, 0.0569, -1.0000, -0.0995,
               --0.3972,
153 0.1145, -0.0462, 0.7882, 0.3013
154         };
155
156         double deseada[] = {
157                                                   1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
                                                      --
158                                                 -1,
                                                      --1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
                                                      --
159                                                 -1,-1,
                                                      --1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
                                                      --
160                                                 -1,-1,-1,
                                                      --1,-1,-1,-1,-1,-1,-1,-1,-1,
                                                      --
161                                                 -1,-1,-1,-1,
                                                      --1,-1,-1,-1,-1,-1,-1,-1,
162                                                 -1,-1,-1,-1,-1,
                                                      --1,-1,-1,-1,-1,-1,
163                                                 -1,-1,-1,-1,-1,-1,
                                                      --1,-1,-1,-1,-1,
164                                                 -1,-1,-1,-1,-1,-1,-1,
                                                      --1,-1,-1,-1,
165                                                 -1,-1,-1,-1,-1,-1,-1,-1,
                                                      --1,-1,-1,
166                                                 -1,-1,-1,-1,-1,-1,-1,-1,-1,
                                                      -- 1,-1,
167                                                 -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
                                                      -- 1
168         };
169
```

```
170         initNN(estructura, 2);
171
172         for (i=0;i<300;i++)
173                 erroresO = BackProp(&mu, test, deseada, 11, pesos,
                        —estructura, 2,
174                                     2, 0, &coste, 0, 0, erroresO);
175
176
177          weightToInt(pesos, pesosI, 491, 8);
178
179      writeWeights(XPAR_NEURALNETWORK_0_BASEADDR, pesosI, 491);
180
181
182
183
184         long testI[] =   {
185         -127, 15, 70, -127, -55, -5, 8, -127, 33, -102, 9, 36,
186         -93, 10, 127, 34, 38, -18, -61, -83, -127, -49, -44, -127,
187         -54, -81, 120, 70, -2, 59, -127, -41, -13, 27, 127, 127,
188         -62, -127, 42, 57, -43, 127, 123, -63, 11, 32, -16, 69,
189         -32, -64, 76, 127, 110, -7, -21, 127, 25, 107, 94, 104,
190         -53, -3, -57, 79, 127, 46, 104, -22, 127, 127, 104, 104,
191          24, 94, 28, 108, -39, 93, 27, -102, 39, 110, 61, 111,
192          31 , 0, -8, 68, -6, 87, 127, 62, 66, 94, 51, 103,
193         -106, 51, -127, 26, -62, 50, -93, -98, 38, -127, 24, 48,
194         127, -18, -104, 68, -127, -24 , 3, -75, 33, 48, -127, 88,
195         -62, 127, -112, -49, 7, -127, -13, -50, 15, -6, 100, 38
196
197         };
198
199
200         for (i=0,j=0; i<11; i+=1,j+=12)
201         {
202                 redInt(pesosI, &testI[j], estructura, 2, tempI, 2, 0, -20,
                        — 20, 8, 8, 8);
203
204                 xil_printf("%d, %d, %d, %d, %d %d, %d, %d, %d, %d, %d %d\r\n
                        —", tempI[20],
205 tempI[21], tempI[22], tempI[23], tempI[24], tempI[25], tempI[26],
206 tempI[27], tempI[28], tempI[29], tempI[30]);
207
208         }
209
210         return 0;
211 }
```

Bibliografía

[1] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann, 2ª edición, 2002.

[2] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1996.

[3] Richard O. Duda, Peter E. Hart, y David G. Stork. *Pattern Classification*. John Wiley & Sons, 2001.

[4] F. Javier Garrigós Guerrero, F. Javier Toledo Moreo, y J. Javier Martínez Álvarez. *Síntesis de Sistemas Digitales con VHDL*. Universidad Politécnica de Cartagena, 2003.

[5] J. Garrigós Guerrero, J. Fernández Conesa, J. Martínez Álvarez, y J. Toledo Moreo. Entorno de desarrollo y depuración de aplicaciones basadas en el microcontrolador soft-core PicoBlaze. *IV Jornadas de Computación Reconfigurable y Aplicaciones*, páginas 227-234, 2004.

[6] J. Garrigós Guerrero, J. Fernández Conesa, J. Toledo Moreo, y J. Martínez Álvarez. Prototipado de ANN con System Generator. *V Jornadas de Computación Reconfigurable y Aplicaciones*, páginas 237-242, 2005.

[7] Martin T. Hagan, Howard B. Demuth, y Mark Beale. *Neural Network Design*. PWS, 1996.

[8] Simon Haykin. *Neural Networks, A Comprehensive Foundation*. Prentice Hall, 2nd edition edición, 1999.

[9] IBM. *CoreConnect Bus Architecture*.

[10] IBM. *On-Chip Peripheral Bus Architecture Specifications*.

[11] Xilinx Inc. *MicroBlaze Processor Reference Guide*.

[12] Xilinx Inc. *PicoBlaze 8-bit Embedded Microcontroller User Guide*.

[13] Xilinx Inc. *PicoBlaze 8-Bit Microcontroller for CPLD Devices*.

[14] Xilinx Inc. *PicoBlaze 8-Bit Microcontroller for Virtex-E and Spartan-II/IIE Device.*

[15] Xilinx Inc. *PowerPC Processor Reference Guide.*

[16] Xilinx Inc. *Processor IP Reference Guide.*