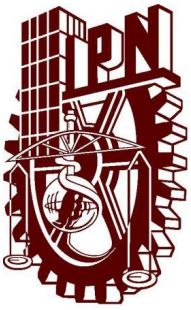

INSTITUTO POLITÉCNICO NACIONAL

CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN

Implementación de una red neuronal morfológica con procesamiento dendral en un FPGA

TESIS

QUE PARA OBTENER EL GRADO DE:

Maestría en Ciencias en Ingeniería de Cómputo con opción en Sistemas Digitales

P R E S E N T A:

Tonantzin Marcayda Guerrero Velázquez

Directores de tesis:
Dr. Juan Humberto Sossa Azuela
Dr. Herón Molina Lozano

México, D.F. Julio 2015

SIP-14 bis

INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

ACTA DE REVISIÓN DE TESIS

En la Ciudad de México, D.F. siendo las 12:00 horas del día 5 del mes de junio de 2015 se reunieron los miembros de la Comisión Revisora de la Tesis, designada por el Colegio de Profesores de Estudios de Posgrado e Investigación del:

Centro de Investigación en Computación

para examinar la tesis titulada:

"Implementación de una red neuronal morfológica con procesamiento dendral en un FPGA"

Presentada por el alumno(a):

Guerrero	**Velázquez**	**Tonantzin Marcayda**
Apellido paterno	Apellido materno	Nombre(s)

Con registro:

A	1	3	0	2	3	0

aspirante de: **MAESTRÍA EN CIENCIAS EN INGENIERÍA DE CÓMPUTO CON OPCIÓN EN SISTEMAS DIGITALES**

Después de intercambiar opiniones los miembros de la Comisión manifestaron ***APROBAR LA TESIS***, en virtud de que satisface los requisitos señalados por las disposiciones reglamentarias vigentes.

LA COMISIÓN REVISORA

Directores de Tesis

Dr. Juan Humberto Sossa Azuela

Dr. Herón Molina Lozano

Dr. Sergio Suárez Guerra

Dr. Victor Hugo Ponce Ponce

Dra. Elsa Rubio Espino

Dr. Ricardo Barrón Fernández

PRESIDENTE DEL COLEGIO DE PROFESORES

INSTITUTO POLITECNICO NACIONAL
CENTRO DE INVESTIGACIÓN
EN COMPUTACIÓN

Dr. Luis Alfonso Villa Vargas

INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

CARTA CESIÓN DE DERECHOS

En la Ciudad de México Distrito Federal, el día 17 del mes junio del año 2015, la que suscribe **Tonantzin Marcayda Guerrero Velázquez**, alumna del **Programa de maestría en Ciencias en Ingeniería de Cómputo** con número de registro **A130230**, adscrito al **Centro de Investigación en Computación**, manifiesta que es autora intelectual del presente trabajo de Tesis bajo la dirección del **Dr. Juan Humberto Sossa Azuela y del Dr. Herón Molina Lozano** y cede los derechos del trabajo intitulado **Implementación de una red neuronal morfológica con procesamiento dendral en un FPGA**, al Instituto Politécnico Nacional para su difusión, con fines académicos y de investigación.

Los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expreso del autor y/o director del trabajo. Este puede ser obtenido escribiendo a la siguiente dirección **mysonjeyton2@gmail.com**. Si el permiso se otorga, el usuario deberá dar el agradecimiento correspondiente y citar la fuente del mismo.

Tonantzin Marcayda Guerrero Velázquez

Resumen

Las redes neuronales artificiales destacan como un campo de investigación de gran interés, teniendo innumerables aplicaciones en diversas áreas de entre las cuales destaca el reconocimiento de patrones, el control de sistemas, el procesamiento de señales y la toma de decisiciones. La teoría de estas redes ha sido satisfactoriamente aplicada en una amplia variedad de problemas referentes al reconocimiento de patrones y segmentación de imágenes.

Existe una nueva clase de redes neuronales artificiales llamadas redes neuronales morfológicas. Dichas redes utilizan máximos o mínimos de sumas para llevar a cabo sus operaciones, a diferencia de la mayoría de las redes neuronales, las cuales basan su funcionamiento en sumas de productos. Con el planteamiento de este tipo de esquema y con la incorporación de nuevas ideas sobre redes neuronales con procesamiento en sus dendritas, las redes neuronales morfológicas han podido igualar y en ocasiones superar el rendimiento de otro tipo de redes neuronales, entrenadas con la regla de aprendizaje con base en retropropagación (*backpropagation*).

Por otro lado, el diseño e implementación de arquitecturas hardware permiten optimizar y obtener un mayor rendimiento y velocidad en las aplicaciones que requieren un procesamiento rápido de la información. Durante el desarrollo de esta tesis se hará uso de dispositivos FPGAs (*Field Programmable Gate Array*).

Un FPGA es un circuito integrado que consta de arreglos lógicos programables que se interconectan por medio de una matriz de cables e interruptores programables, de esta manera se puede realizar procesamiento de información compleja de manera eficiente. Los sistemas basados en FPGA pueden ser programados y reprogramados muchas veces, gracias a esto se puede obtener una reducción de costos en el desarrollo de una sistema hardware.

Partiendo de lo descrito anteriormente, la meta que se propone para este tema de tesis consiste en realizar el diseño, implementación y validación en hardware de una red neuronal morfológica con procesamiento en sus dendritas y su entrenamiento, para datos de una y dos dimensiones con p clases.

Abstract

Artificial neural networks have been shown to be an outstanding field of research, it is wide and interesting. With an infinite number of applications in a great number of areas among which stands out image processing. The theory of neural networks has been successfully applied in an great variety of problems dealing with the pattern recognition and the image segmentation.

There is a new kind of artificial neuronal networks named *morphological neural networks*. Such neural networks use maximun and minimum of algebraic adders to carry out their operations unlike the majority of the neural networks, which rely their functioning in product additions. With this new scheme and with the incorporation of the new ideas on morphological neural networks, morphological neural networks have matched and sometimes overcome the performance of the backpropagation algorithm that is aplied to multilayer neural networks.

On the other hand, the design and implementation of hardware architectures allow the optimization and acquisition of a better performance and speed in the applications that require data processing in real time. FPGAs devices will be used during the development of this thesis.

An FPGA is an integrated circuit that is made of programable logical arrays that are interconnected via a matrix of programable switches and wires. In this way, it turns out to be a highly programable device and it is capable of doing complex information procesing in an efficient way. Systems based on FPGA can be programed and reprogramed several times, and due to it, a reduction in costs in the development of a hardware system can be obtained. Another advantege in the use of FPGA technologies, it allows to realize architectures designs according to the needs of the application to be implemented.

From what has been described the goal to be met by the thesis proposed consists of realizing the design, implementation and validation in hardware of the dendritic morphological neural network and its training.

.

Dedicado

A mis padres Cristina y Juan...

Quienes siempre estuvieron a mi lado apoyándome en cada paso dado,
quienes con esfuerzo y sacrificios me brindaron todo el apoyo económico a lo largo de mis estudios,
ayudándome a ser una persona de provecho,
por que gracias a ellos he logrado mis metas,
pero sobre todo, por el inmenso amor que me han brindado siempre.

Gracias papás los amo.

A mi esposo Jesús...

Por ayudarme y apoyarme incondicionalemente,
brindándome todo su amor, cariño, respeto y confianza,
por ser mi complemento y el amor de mi vida.

Te amo.

A mi hermano Carlos...

Por ser el mejor hermano del mundo,
alentándome a seguir adelante con sus palabras de aliento y su gran cariño,
compartiendo nuestros triunfos en cada paso desde niños.

Te amo hermano.

Tonantzin Marcayda

Agradecimientos

A Dios por darme la fuerza para seguir adelante y no darme por vencida.

A mis asesores, el Dr. Juan Humberto Sossa Azuela y el Dr. Herón Molina Lozano por brindarme la oportunidad de poder trabajar con ellos, y guiarme en este arduo camino.

A mi comité de evaluación por sus valiosas observaciones.

A mis amigos por estar conmigo en los buenos y malos momentos, gracias.

Al Centro de Investigación en Computación del Instituto Politécnico Nacional por permitirme realizar mis estudios de posgrado y brindarme los cimientos de mi formación académica dentro de uno de los mejores centros de investigación del País.

Al COMECyT por el apoyo económico que me fue brindado en el transcurso de mis estudios de maestría.

Para concluir, este trabajo de tesis se realizó en el marco de los proyectos con número de registro: SIP 20131505, SIP 20131182, SIP 20144538, SIP 20141310, SIP 20151769, SIP 20151187, SIP 20151625 y CONACyT 155014.

Índice general

Índice de figuras

Glosario

- FPGA: una FPGA (del inglés Field Programmable Gate Array) es un dispositivo semiconductor que contiene bloques de lógica cuya interconexión y funcionalidad puede ser configurada mediante un lenguaje de descripción especializado.
- RNMD: red neuronal morfológica con procesamiento en sus dendritas.
- MATLAB: abreviatura de *MATrix LABoratory*–laboratorio de matrices, es una herramienta de software matemático que ofrece un entorno de desarrollo integrado (IDE) con un lenguaje de programación propio (lenguaje M).
- QUARTUS: es una herramienta de software producida por Altera para el análisis y la síntesis de diseños realizados en HDL.
- VLSI: es la sigla en inglés de *Very Large Scale Integration*, integración a gran escala. La integración a gran escala de sistemas de circuitos basados en transistores en circuitos integrados comenzó en los años 1980, como parte de las tecnologías de semiconductores y comunicación que se estaban desarrollando.
- VDHL: es un lenguaje definido por el IEEE (Institute of Electrical and Electronics Engineers) (ANSI/IEEE 1076-1993) usado por ingenieros para describir circuitos digitales. VHDL es el acrónimo que representa la combinación de VHSIC y HDL, donde VHSIC es el acrónimo de *Very High Speed Integrated Circuit* y HDL es a su vez el acrónimo de *Hardware Description Language*.
- *PIPELINE*: consiste en múltiples procesos ordenados de tal forma que el flujo de salida de un proceso alimenta la entrada del siguiente proceso.
- RAM: Es una memoria de acceso aleatorio (*Random-Access Memory*, RAM). Se denominan *de acceso aleatorio* porque se puede leer o escribir en una posición de memoria con un tiempo de espera igual para cualquier posición, no siendo necesario seguir un orden para acceder (acceso secuencial) a la información de la manera más rápida posible.
- LUTs: En este contexto una LUT(*look–up table*) es una colección de compuertas lógicas cableadas dentro de un FPGA. Una LUT almacena una lista predefinida de salidas para cada combinación de entradas y proveé una forma rápida de recuperar la salida de una operación lógica.

Capítulo 1

Introducción

Desde el comienzo de la revolución computacional ha habido gran interés por el desarrollo de máquinas inteligentes. El enfoque: *inteligencia computacional*, busca el aplicar inteligencia artificial en problemas reales, basándose en estructuras copiadas de la naturaleza. Las redes neuronales artificiales emulan de manera simplificada el funcionamiento de las redes biológicas [30].

El uso de redes neuronales ocupa hoy en día un lugar muy importante en la solución de problemas complejos, tales como: reconocimiento de patrones, predicción, codificación, clasificación, control, optimización, entre otros. Por lo que su implementación en hardware es una parte esencial para el desarrollo de estas aplicaciones [1].

Las investigaciones han estado dirigidas principalmente a la simulación en computadora de los algoritmos que se proponen, lo que se aleja del comportamiento real de las neuronas biológicas.

La gran mayoría de las aplicaciones de redes neuronales artificiales, se escriben en un programa secuencial que simula el entrenamiento y ejecución de la red neuronal, este procedimiento no toma en cuenta el paralelismo natural encontrado en la topología de la red neuronal artificial, sin embargo, la implementación de redes neuronales en hardware puede ayudar a solucionar este inconveiente [2] [3].

Tomando en cuenta que los niveles de integración y las prestaciones de los FPGAs aumentan constantemente cada año, es viable suponer que en un futuro cercano esta aproximación hará válido el desarrollo de sistemas bioinspirados soportados por hardware reconfigurable [4].

1.1. Planteamiento del problema

La redes neuronales con procesamiento dendral son una de las mejores opciones hoy en día para la clasificación de patrones, al especificar sus pesos sinápticos usando el algoritmo descrito en [29]. En [29] y [32], pueden encontrarse ejemplos ilustrativos de la aplicación de esta red neuronal junto con su algoritmo de entrenamiento en diversas tareas de clasificación de patrones y análisis de imágenes.

El algoritmo reportado en [29] presenta las siguientes características:

- Convergencia a una solución en un número finito de iteraciones.
- Perfecta clasificación del conjunto de entrenamiento dado.
- No existe un traslapado entre los hiper–cubos generados de distintas clases.
- No depende del orden en el que se presenten las clases.
- No hay áres de indesición en la fase de prueba.
- El algoritmo puede ser aplicado a la clasificación de problemas de p clases y n atributos.

Todas estas características hacen que dicho algoritmo sea muy eficiente para resolver satisfactoriamente diferentes problemas de clasificación. El principal incoveniente del algoritmo propuesto en [29] es que crece exponencialmente a medida que el número de dimensiones incrementa; en una máquina secuencial cuando el número de dimensiones llega a $n = 20$, el tiempo del cálculo de los hiper–cubos deja de ser práctico.

A continuación, se muestra un ejemplo de lo descrito anteriormente. Se tomó como referencia un conjunto de datos de entrenamiento de 1000 valores generados de manera aleatoria, se fueron agregando atributos al conjunto de datos para su entrenamiento, obteniendo así conjuntos de datos de 1000 valores con $n = 2$ hasta $n = 20$ donde n es el número de dimensiones o atributos. Este ejemplo fue otorgado por los autores de la referencia citada en [29]

En la figura 1.1, se puede observar que el tiempo de entrenamiento crece de acuerdo al número de dimensiones que tenga el conjunto de datos, además de que cuando $n = 20$ el tiempo de entrenamiento del conjunto de datos tiende a crecer exponencialmente, debido a que el algoritmo de entrenamiento es de complejidad exponencial.

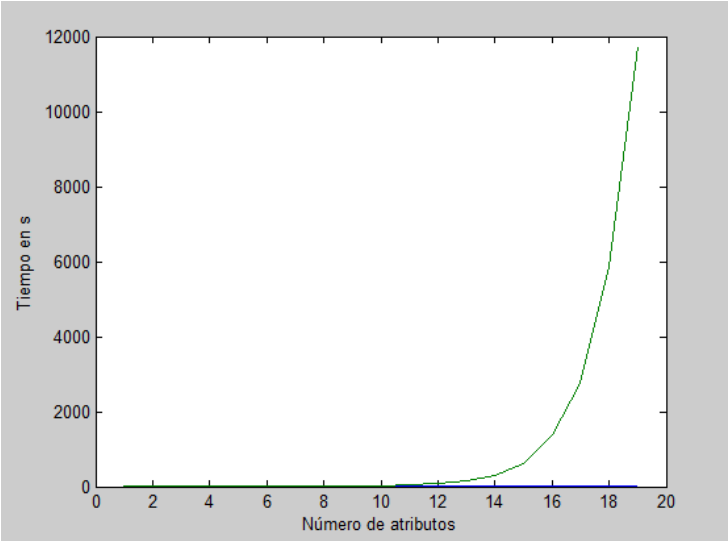

Figura 1.1: Tiempo de entrenamiento exponencial

Con el fin de reducir el tiempo de entrenamiento para un problema dado, en este tema de tesis se propone la implementación de una red neuronal morfológica con procesamiento en sus dendritas y su correspondiente algoritmo de entrenamiento en un FPGA, sin perdida de generalidad para el caso de datos de una y dos dimensiones.

1.2. Justificación

La redes neuronales morfológicas con procesamiento en sus dendritas son una alternativa a las redes neuronales clásicas. Hasta el momento este tipo de redes neuronales se han implementado (de acuerdo a la investigación realizada) en máquinas secuenciales utilizando diversos lenguajes de programación tales como: MATLAB, C, C#, entre otros.

Hoy en día una de las corrientes tecnológicas fuertemente impulsadas para la resolución de problemas con alta demanda de recursos, es la utilización de los dispositivos lógicos programables. Basado en lo anterior, se propone realizar por primera vez una implementación en hardware de este tipo de redes, haciendo uso de un dispositivo FPGA.

1.3. Objetivos

1.3.1. Objetivo general

Implementar una RNMD en un dispositivo FPGA, para resolver problemas de clasificación con p clases para una y dos dimensiones.

1.3.2. Objetivos específicos

- Implementar una RNMD para datos de una y dos dimensiones con p clases en VHDL.
- Implementar el *algoritmo de entrenamiento* para este tipo de redes en lenguaje VHDL, reportado en [29].
- Probar y verificar la eficiencia y eficacia del algoritmo de entrenamiento implementado, haciendo uso de cuatro conjuntos de datos y comparar resultados con los obtenidos en MATLAB.

1.4. Organización del documento

El presente trabajo de tesis esta conformado por 6 capítulos, cada capítulo expone lo siguiente:

- El Capítulo 1 consiste en la parte introductoria del documento y tiene la finalidad de establecer el planteamiento del problema, la justificación y los objetivos de este trabajo de tesis.
- En el Capítulo 2 se habla de los antecedentes acerca de implementaciones de redes neuronales sobre FPGA, así como los trabajos relacionados. Cabe señalar que sobre la implementación de una red neuronal morfológica con procesamiento en sus dendritas aún no existen trabajos relacionados en hardware. Además, se mencionan los resultados esperados al finalizar este trabajo de tesis.
- En el Capítulo 3 se profundiza en los diferentes temas que permiten comprender el trabajo de tesis. Se describe el comportamiento de una red neuronal morfológica con procesamiento en sus dendritas y cuáles son las operaciones que la rigen. Se describe el algortimo de entrenamiento y se da un ejemplo para comprender la ejecución del mismo. De manera adicional, se describen las características y ventajas de los dispositivos programables FPGA(*Field Programmable Gate Array*).
- En el Capítulo 4 se expone el diseño de la red neuronal morfológica con procesamiento en sus dendritas así como el entrenamiento de la misma, ambos en lenguaje VHDL.
- En el Capítulo 5, se llevan a cabo una serie de pruebas con varios conjuntos de datos de entrenamiento, con la finalidad de probar los resultados obtenidos del entrenamiento. Se realiza una comparación, entre los resultados dados por la implementación de MATLAB y la realizada en FPGA, mediante las simulaciones en

FPGA con la tarjeta de desarrollo DEA2 de Altera. Además, se muestra el reporte de compilación de los diseños implementados, el cual contiene información sobre los recursos lógicos utilizados. También, se muestran los diagramas con los tiempos de entrenamiento obtenidos.

- En el Capítulo 6 se exponen las conclusiones y contribuciones que se obtuvierón al realizar este trabajo de tesis. Finalmente, se mencionan el trabajo a futuro.

Capítulo 2

Estado del arte

En este capítulo se describen brevemente los trabajos relacionados con el tema de tesis a desarrollar. Se tratan trabajos sobre implementaciones de redes neuronales en FPGA, memorias asociativas e implementaciones basadas en RAM. Cabe mencionar que aún no existe alguna publicación que trate sobre la implementación de una RNMD en un dispositivo hardware o en FPGA.

2.1. Implementación de redes neuronales sobre FPGA

El diseño de dispositivos hardware para realizar arquitecturas de redes neuronales artificiales y sus algoritmos de aprendizaje asociados, toman especial ventaja del inherente paralelismo en el procesamiento neuronal así como del hardware mismo.

La ventajas de llevar a cabo este tipo de implementaciones se listan a continuación[1]:

- Velocidad: el hardware especializado puede ofrecer un muy alto poder computacional, especialmente en el dominio de las redes neuronales en donde el paralelismo y el cómputo distribuido se encuentran estrechamente relacionados.
- Costo: una implementación en hardware permite reducir el costo del sistema, al utilizar menos recursos y con ello disminuir los requisitos de la energía a utilizar.
- Tolerancia a fallos del sistema: una limitación propia de cualquier aplicación basada en un único procesador secuencial es su vulnerabilidad a detener la funcionalidad de la misma, debido a fallas en el sistema. La razón principal es la falta de suficiente redundancia en la arquitectura del procesador, en contraste con las arquitecturas paralelas y distribuidas, que permiten que las aplicaciones continúen funcionando,

aunque con un rendimiento ligeramente reducido aún en presencia de fallas en algunos componentes. Para aplicaciones de redes neuronales artificiales se requiere de una completa disponibilidad de la información, por lo tanto la tolerancia a fallos es de gran importancia, y en lo que respecta las implementaciones en hardware ofrecen una ventaja considerable [3].

Una red neuronal artificial generalmente se especifica en términos de la topología de la red, la función de activación, el algoritmo de entrenamiento, número y tipo de entradas y salidas, número de elementos a procesar (neuronas), interconexiones sinapticas, y número de capas. Para las implementaciones en hardware, adicionalmente, las especificaciones pueden incluir la tecnología usada (analógica, digital o *fpga*), la representación de los datos (entero, punto fijo o punto flotante), el almacenamiento de los pesos y los bits de presición.

Uno de los motivos más importantes del resurgir de las redes neuronales en la década de los ochenta fue el desarrollo de la tecnología microlectrónica de alta escala de integración o *VLSI (Very Large Scale Integration)*, debido a dos circunstancias. Por una parte, posibilitó el desarrollo de computadoras potentes y baratas, lo que facilitó la simulación de modelos de redes neuronales artificiales de un relativamente alto nivel de complejidad, permitiendo su aplicación a numerosos problemas prácticos en los que se demostró un excelente comportamiento. Por otra lado, la integración de sistemas VLSI permitió la realización directa en hardware de una red neuronal artificial, haciendo uso del cálculo paralelo para dar solución a problemas computacionalmente costosos, como la visión o el reconocimiento de patrones [31].

Los dispositivos reconfigurables FPGA son un recurso programable efectivo para implementar redes neuronales artificiales en hardware, ya que permiten implementar diversas opciones de diseño para ser evaluadas es un lapso de tiempo pequeño. Son de bajo costo, estan fácilmente disponibles, y ofrecen la flexibilidad de implementación que da el software. Además, las actuales características de reconfiguración parcial y dinámica en las últimas generaciones de FPGA ofrecen ventajas adicionales.

Los continuos avances tanto en las herramientas de diseño como en los dispositivos FPGA, han permitido que se puedan implementar sistemas digitales de mayores prestaciones basados en esta tecnología. La princial característica que permite esto, es que dichos dispositivos ya cuentan con la habilidad de poder reconfigurar una porción de ellos mismos aún en tiempo de ejecución. De esta manera, se permite una mayor reusabilidad del hardware así como implementar hardware adaptativo, el cual pueda adaptarse a las nuevas condiciones del sistema.

La mayoría de técnicas y herramientas actuales para el diseño en FPGA se encuentran enfocadas a diseños estáticos, esto limita enormemente que se puedan adaptar al uso de tecnologías de reconfiguración dinámica. El implementar diseños de sistemas con el uso de estas herramientas sigue siendo algo complejo debido a que existen diversas desventajas tales como: a) se requiere un alto nivel de conocimiento sobre la arquitectura del dispositivo; b) se deben especificar las regiones configurables en la última etapa del proceso de diseño; c) la reconfiguración dinámica se establece a través de la línea de comandos; d) el uso de recursos es a bajo nivel; e) sólo es soportada por determinadas arquitecturas y; f) además, aún no se cuenta con un procedimiento especifíco para llevar a cabo su diseño.

2.2. Trabajos relacionados

A continuación, se describen brevemente algunos de los trabajos realizados en el área de implementación de redes neuronales artificiales sobre dispositivos FPGA, así como las aplicaciones que se les han dado a cada uno de ellos, descritos en [34].

Krips presenta en [5] una implementación en FPGA de una red neuronal diseñada para la detección de una mano en tiempo real y un sistema de rastreo aplicado a imagénes de vídeo. Yang and Paindavoine [6] presentan un hardware basado en FPGA con una tasa de éxito del 92 %, para el rastreo de rostros y la verificación de identidad en secuencias de vídeo. Maeda y Tada [7] describen una implementación en FPGA de una red neuronal de densidad de impulsos usando el método de perturbaciones simultáneas como esquema de aprendizaje. El método de pertubaciones simultáneas es más sensible a una implementación en hardware que el de una regla de aprendizaje de tipo gradiente. Los sistemas de redes neuronales de densidad de impulsos son robustos frente a condiciones ruidosas.

Los FPGA son fácilmente adaptables a un costo razonable y tienen un ciclo reducido de desarrollo de hardware. Los sistemas basados en FPGA pueden ser adaptados a una configuración especifíca de una red neuronal artificial. Por ejemplo, Gadea en [8] presenta la implementación de un perceptrón multicapa para el análisis de un arreglo de datos sistólicos, utiliza el método *pipeline* para implementar el algoritmo de aprendizaje *back propagation*. La multiplicación es poco costosa usando FPGA ya que cada conexión sináptica en una red neuronal artificial requiere de un solo multiplicador, y este número tipicamente crece como el número de neuronas al cuadrado. Las FPGA modernas, tales como Virtex II Pro de Xilinx [9] y Stratis III de Altera, pueden llegar a tener cientos de multiplicadores dedicados.

En un trabajo relativamente reciente, Himavathi en [10] ha usado la técnica de multiplexación de capas para implementar redes multicapa de retroalimentación hacia adelante, dentro de un FPGA. La técnica de multiplexación de capas sugerida, implica implementar solamente la capa que tenga el mayor número de neuronas. Se diseña un bloque de control, con las neuronas seleccionadas apropiadamente de esta capa para simular el comportamiento de cualquier otra capa y asignar así las entradas, los pesos, el bias, y la función de excitación para cada neurona de la capa que esta siendo emulada actualmente en paralelo.

En otro estudio reciente, Rice en [11] reporta que una implementación con base en FPGA de un neocórtex inspirado en un modelo cognitivo, puede proporcionar una ganancia promedio de rendimiento 75 veces mayor que la de una implementación en software hecha sobre una super computadora. Se hace del modelo de red *Bayesiano* de tipo jerárquico basado en el neocórtex, desarrollado por George and Hawkins [12].

Un problema importante que enfrentan los diseñadores de redes neuronales artificiales en hardware con base en FPGAs, es el de seleccionar adecuadamente el modelo de red neuronal para resolver un problema en especifíco, e implementarlo haciendo un uso óptimo de los recursos del hardware. Simon Jothson y algunos otros investigadores han propuesto ideas interesantes a cerca del uso óptimo de los recursos [13]. Llevarón a cabo un análisis comparativo de los requerimientos de hardware para implementar cuatro modelos diferentes de redes neuronales artificiales dentro de un dispositivo FPGA. Los resultados del estudio sugieren que el modelo de red neuronal de tipo con fuga en la integración y disparo (*leaky and integrate and fire*) podría ser la elección más apropiada para la implementación de tareas de clasificación no lineales.

2.2.1. Implementaciones sobre FPGAs de modelos estocásticos de redes neuronales artificiales

Las implementaciones prácticas en hardware de redes neuronales artificiales de gran tamaño tienen un requerimiento crítico, el cual implica el reducir significativamente la circuitería dedicada a las operaciones de multiplicación. Una forma de llevar a cabo esta reducción, es el hacer uso de la computación estocástica de bits en serie [14]. Utiliza cadenas probabilísticas de bits relativamente largas, en donde su valor númerico es proporcional a la densidad de los unos que contenga. La multiplicación de dos cadenas probabilísticas de bits, puede ser realizada con una única compuerta lógica de dos entradas, esto hace que sea factible implementar redes completamente paralelas con tolerancia a fallos.

Muchos de los modelos estocásticos de redes neuronales artificiales han sido implementados en hardware haciendo uso de los FPGAs. Daalen en [15] describe una arquitectura digital expandible con base en FPGAs con computación estocástica de bits en serie, para llevar a cabo el cálculo sináptico en paralelo. Los autores describen que las redes multicapa completamente conectadas pueden ser implementadas con tiempo de multiplexación haciendo uso de esta arquitectura.

Nedjah y Mourelle en [16] describen y comparan las características de dos FPGA de la familia Xinlinx VIRTEX-E, basándose en el prototipado de diferentes arquitecturas para el modelado de redes neuronales artificiales totalmente conectadas con propragación hacia adelante, con un tamaño de hasta 256 neuronas. El primer prototipo usa sumadores y multiplicadores tradicionales de entradas binarias, mientras que el segundo tiene representación estocástica de las entradas con sus correspondientes cálculos estocásticos. Comparan ambos prototipos en términos de los requerimientos de espacio, los retardos de la red, y finalmente el factor *tiempo* $\times$ *área*. Como era de esperarse, la representación estocástica reduce los requerimientos del espacio en buena medida aunque, las redes resultantes son ligeramente más lentas comparadas con las de los modelos binarios.

Szabo en [17] sugiere un método de implementación de red neuronal de bits en serie/paralelo, para redes neuronales artificiales pre-entrenadas, usando aritmética de bits distribuida en serie para la implementación de filtros digitales. La implementación de un multiplicador de matrices por vectores, se basa en un algortimo de optimización, el cual hace uso de la representación canónica de digítos con signo y un patrón de coincidencias a nivel de bit. La arquitectura resultante se puede implementar usando un FPGA, y se puede integrar en automático a los ambientes de diseño de redes neuronales.

2.2.2. Memorias asociativas

La operación básica de una memoria asociativa es el mapeo entre dos conjuntos de patrones finitos haciendo uso de la operación de umbralado. Palm en [18] estudió un modelo muy simple de red neuronal que realizaba está tarea eficientemente, en donde la entrada, la salida y la conexión entre pesos eran binarios. Posteriormente, Ruckert en [19] [20] diseñó arquitecturas VLSI para este modelo, usando técnicas de diseño de circuitos tanto digitales, como analógicas o una mezcla de ambas.

Willshaw en [21] define un modelo de memoria asociativa llamado *memoria de matriz de correlación*, en donde el patrón de salida es una etiqueta asociada con el patrón más similar que fue almacenado a la entrada. Justin en [22] presenta una implementación basada

en FPGA de una memoria asociativa con arquitectura segmentada (*pipeline*), en la cual se lleva a cabo la fase de entrenamiento y prueba, y se aplica a tareas de reconocimiento de patrones.

2.2.3. Implementaciones con base en memoria RAM

Bledsoe y Browning en [23] introdujerón la implementación de redes neuronales con base en memorias RAM (*Random Access Memory*), las cuales se componen de neuronas que tienen solamente entradas y salidas binarias, y además, no existen los pesos entre la conexión de los nodos. Las funciones neuronales son almacenadas dentro de tablas de busqueda (LUT – *Look up table*), las cuáles pueden ser implementadas en RAMs comerciales. A diferencia de otros modelos de redes neuronales, estas redes pueden ser entrenadas rápidamente haciendo uso de hardware simple. En lugar de ajustar los pesos de la red neuronal de manera convencional, las redes neuronales con base en RAM son entrenadas cambiando el contenido de las LUTs. Este tipo de redes, tienen aplicaciones en métodos para los sistemas de reconocimiento de reconstrucción de patrones.

Alexander en [24] muestra la primera implementación en hardware de un sistema de propósito general para el reconocimiento de imágenes llamado WISARD, el cual se basa en circuitos de memoria RAM. En [25], se presenta una implementación en hardware de una red neuronal artificial probabilística con base en RAM así como su algoritmo de entrenamiento.

El área de investigación sobre la implementación de redes neuronales artificiales en dispositivos hardware y sus aplicaciones, ha ido incrementando progresivamente. Han ido apareciendo constantemente nuevas áreas de aplicación novedosas, tales como, microcontroladores embebidos para el control de robots autónomos, control autónomo de vuelo, así como varias aplicaciones que intentan emular el comportamiento de las funciones cerebrales del ser humano.

En este trabajo de tesis se llevará a cabo la implementación en hardware de una red neuronal morfológica con procesamiento en sus dendritas así como su entrenamiento. Se pretende implementar dicha red, para una y dos dimensiones de p clases cada una.

Capítulo 3

Marco teórico

En este capítulo se describen a detalle los conceptos que engloban el desarrollo e implementación de este trabajo de tesis. En la primera parte se da una pequeña introducción sobre el concepto general de una red neuronal morfológica y sus características, mencionando en que difiere con los modelos clásicos de redes neuronales artificiales. Posteriormente, se trata sobre el concepto, funcionamiento y ecuaciones características de una red neuronal morfológica con procesamiento en sus dendritas, que es la red neuronal artificial que se implementó en el dispositivo FPGA. Finalmente, se introducen los conceptos básicos de los circuitos reconfigurables y del dispositivo FPGA, así como de los lenguaje de descripción de hardware con los cuales se puede programar, en especial del lenguaje VHDL con el cual se llevó a cabo el desarrollo de este tema de tesis.

3.1. Red neuronal morfológica

El estudio de las redes neuronales artificiales ha tenido recientemente una explosión en resultados prácticos y teóricos. Una red neuronal morfológica, es algebraicamente distinta a las clásicas redes neuronales artificiales así como también sus aplicaciones. La principal diferencia entre las redes clásicas y las morfológicas, se da en la manera en la que cada nodo combina algebraicamente la información númerica. Cada nodo en una red neuronal clásica combina la información mediante la multiplicación de los valores de la salida con su correspondiente peso y posteriomente la suma de los mismos, mientras que en una red neuronal morfológica, la combinación de las operaciones consiste en la suma de los valores con sus correspondientes pesos, para así, posteriormente tomar el valor máximo [33].

Las redes neuronales morfológicas [26], difieren de los modelos clásicos en la manera en que modelan la interacción entre las señales de entrada y los diferentes parámetros de la neurona. En particular, las señales interactúa con los pesos sinápticos de manera aditiva

y no multiplicativa; así pues, el procesamiento de la información tanto en las dendritas como en el cuerpo celular se realiza haciendo uso de operadores morfológicos (máximo y mínimo) y no sumas como en el caso clásico.

3.2. Red neuronal morfológica con procesamiento en sus dendritas

Las redes neuronales morfológicas utilizan operaciones de tipo *lattice*, $\vee$ (máximo) o $\wedge$ (mínimo) , y + para los semi-anillos $(R_{-\infty}, \vee, +)$ o $(R_{\infty}, \wedge, +)$ donde $R_{-\infty} = R \bigcup \{-\infty\}$ y $R_{\infty} = R \bigcup \{\infty\}$. El cálculo de la neurona en una red neuronal morfológica para una entrada $x = (x_1, x_2, \ldots, x_n)$ esta dado por las siguientes ecuaciones:

$$\tau_j = a_j \bigvee_{i=1}^{n} b_{ij}(x_i + w_{ij}) \tag{3.1}$$

o

$$\tau_j = a_j \bigwedge_{i=1}^{n} b_{ij}(x_i + w_{ij}) \tag{3.2}$$

en donde $b_{ij} = \pm 1$ denota si la i-ésima neurona ocasiona exitación o inhibición en la j-ésima neurona, $a_j = \pm 1$ denota la respuesta a la salida (exitación o inhibición) de la j-ésima neurona a las neuronas con las cuales sus áxones tienen contacto y w_{ij} denota la fuerza sináptica entre la i-ésima neurona y la j-ésima neurona. Los parámetros b_{ij} y a_j toman el valor de $+1$ ó -1 dependiendo de si la i-ésima neurona de entrada causa exitación o inhibición a la j-ésima neurona.

El cálculo realizado por la k-ésima dendrita puede ser expresado mediante la siguiente fórmula:

$$D_k(x) = a_k \bigwedge_{i \epsilon I} \bigwedge_{l \epsilon L} (-1)^{1-l}(x_i + w_{ik}^l) \tag{3.3}$$

en donde $x = (x_1, x_2, \ldots, x_n) \epsilon R_n$ corresponde a la entrada de las neuronas, $I \subseteq 1, \ldots, n$ denota a el conjunto de todas las neuronas de entrada N_i con fibras terminales que hacen sinapsis en la k-ésima dendrita de la neurona morfológica N, $L \subseteq 0, 1$ corresponde a el conjunto de las fibras terminales de la i-ésima neurona que hace sinapsis con la k-ésima dendrita de N, y $a_k \epsilon -1, 1$ denota la respuesta de exitación o inhibición de la k-ésima dendrita.

Claramente, $I \neq 0$ y $L \neq 0$ ya que hay al menos una fibra axonal proveniente de al menos una de las neuronas de entrada con la sinapsis de la dendrita k. La función de

activación usada en las redes neuronales morfológicas es una función de límite duro, la cuál asigna 1 si la entrada es mayor o igual a 0 y asigna 0 si la entrada es menor que 0. Una explicación más detallada puede ser encontrada en [27] [28].

3.2.1. Algoritmo de entrenamiento

Una problemática clave en el diseño de una red neuronal artificial con procesamiento en sus dendritras (RNMD) es el entrenamiento; esto es, la selección del número de dendritas y los valores de los pesos sinápticos para cada una de ellas. A continuación, se dará un ejemplo simple con el cual se explicará el algoritmo. El ejemplo consta de tres clases con dos atributos, en la figura 3.1 se muestra los patrones de ejemplo, los patrones C^1 son puntos rojos, los patrones de C^2 son puntos verdes y los patrones de C^3 son puntos azules.

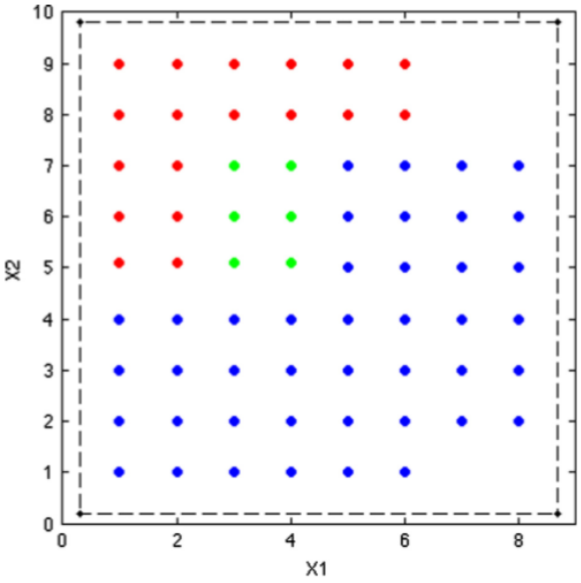

Figura 3.1: Patrones de Ejemplo

Dadas p clases de patrones C^k, $k = 1, 2, \ldots, p$, cada uno con n atributos, el algoritmo a aplicar es el siguiente:

1. Seleccionar los patrones de todas las clases y abrir un hiper-cubo HC^n (donde n es el número de atributos) con un tamaño tal que todos los elementos de las clases queden dentro del hipe-rcubo. El hiper-cubo debe tener un margen M de cada lado, para tener una mejor tolerancia al ruido al momento de clasificar.

2. Dividir el hiper-cubo en 2^n hiper-cubos más pequeños y verificar si se satisface el criterio de paro. El cual implica, que cada hiper-cubo encierre patrones que pertenezcan a una misma clase. Si es el caso, etiquetar el hiper-cubo con el nombre de la

clase correspondiente y parar el proceso de aprendizaje, seguir con el paso 4. En el ejemplo, la primera división de la caja se muestra en la figura 3.2.

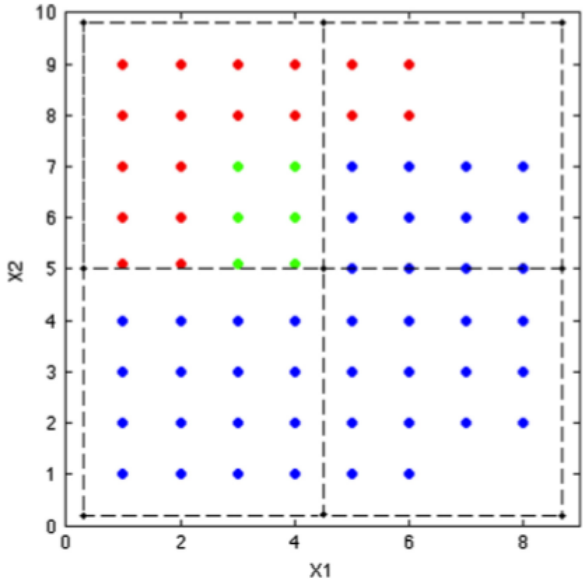

Figura 3.2: Primera división ejecutada por el algoritmo de entrenamiento

3. El paso 3 consta de dos etapas:

   3.1. Si al menos uno de los hiper-cubos generados (HC^n) tiene patrones de más de una clase, dividir el hiper-cubo en 2^n hiper-cubos más pequeños. Repetir de manera iterativa el proceso de verificación dentro de cada nuevo hiper-cubo generado, hasta que el criterio de paro se satisfaga. En la figura 3.3 se muestran los hiper-cubos generados por el algoritmo de entrenamiento.

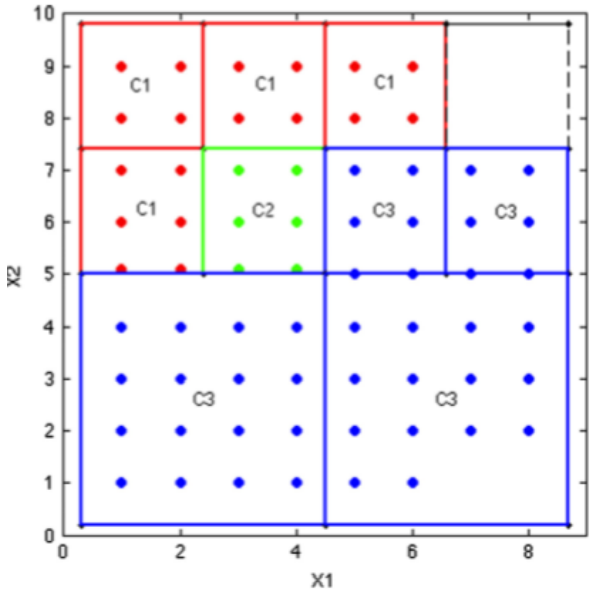

Figura 3.3: Cajas generadas después del proceso iterativo de división

3.2. Una vez que todos los hiper-cubos fueron generados, si dos o mas hiper-cubos de la misma clase comparten un lado en común serán agrupados dentro de una región. En la figura 3.4 se muestra como es que se aplica este proceso de simplificación, el cuál provoca que el número de hiper-cubos se reduzca.

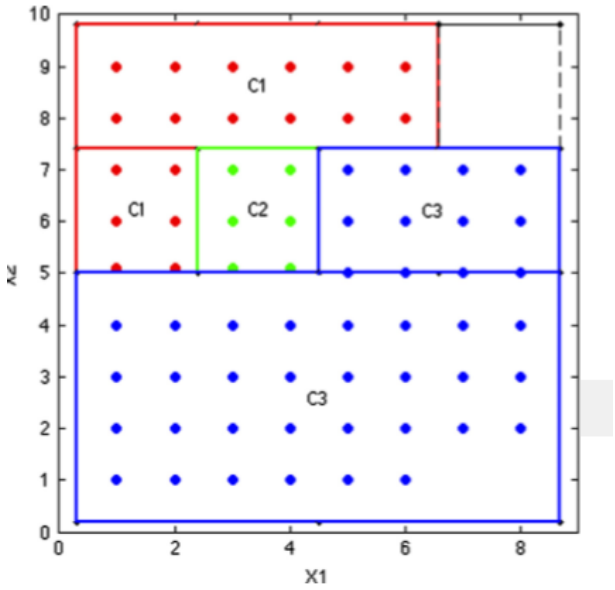

Figura 3.4: Cajas obtenidas después de aplicar el algoritmo de simplificación

4. Con base en las coordenadas de cada eje, calcular los pesos para cada hiper-cubo que encierra los patrones pertenecientes de la clase C^k, tomando en cuenta solo aquellos hiper-cubos que encierren elementos de la clase C^k.

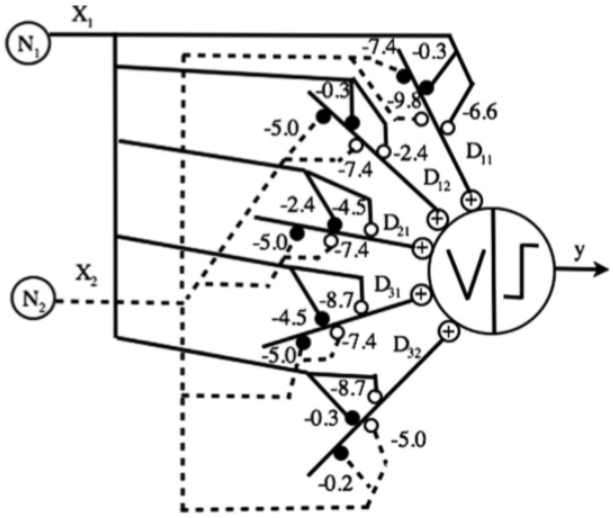

Figura 3.5: Diseño de una red neuronal morfológica con procesamiento en sus dendritas

Una vez hecho lo anterior, se procede a diseñar las red neuronal morfológica con procesamiento en sus dendritas (DMNN), tal y como se muestra en la figura 3.5, la cual tiene una capa de entrada que separa las tres clases: C^1, C^2 y C^3. Las neuronas de la capa de entrada están conectadas a la siguiente capa por medio de las dendritas. Los circulos blancos y negros, denotan las conexiones de exitación y de inhibición, respectivamente. La interpretación geométrica del cálculo realizado por una dendrita es que cada dendrita determina un hiper-cubo el cual puede ser definido por una sola dendrita por medio del valor de sus pesos w_{ij} como se muestra en el ejemplo.

Para probar el diseño de la red resultante, se hará uso de dos patrones de prueba $X_1 = \binom{4,5}{8,5}$ que pertenece a la clase C^1 y $X_2 = \binom{4}{3,5}$ que pertenece a la clase C^3. Cuando la ecuación 3.3 es aplicada a la primera dendrita, se obtienen los siguientes resultados:

$$D_{11}(X_1) = D_{11}\binom{4,5}{8,5} = [(4{,}5-0{,}3)\wedge-(4{,}5-6{,}6)]\wedge[(8{,}5-7{,}4)\wedge-(8{,}5-9{,}8)] = [2{,}1\wedge 1{,}1] = 1{,}1$$

$$D_{11}(X_2) = D_{11}\binom{4}{3,5} = [(4{,}0-0{,}3)\wedge-(4{,}0-6{,}6)]\wedge[(3{,}5-7{,}4)\wedge-(3{,}5-9{,}8)] = [2{,}6\wedge-3{,}9] = -3{,}9$$

Las demás dendritas se calculan de igual manera:

- Para X_1 : D_{12} = -2.1, D_{21} = -1.1, D_{31} =-1.1, D_{32} = -3.5
- Para X_2 : D_{12} = -1.6, D_{21} = -1.5, D_{31} =-1.5, D_{32} = 1.5

Con los valores obtenidos y usando la ecuación 3.1, se logra la clasificación de los datos de prueba:

$$\tau(X_1) = (D_{11} \vee D_{12} \vee D_{21} \vee D_{31} \vee D_{32}) = (1{,}1 \vee -2{,}1 \vee -1{,}1 \vee -1{,}1 \vee -3{,}5) = 1{,}1$$

$$\tau(X_2) = (D_{11} \vee D_{12} \vee D_{21} \vee D_{31} \vee D_{32}) = (-3{,}9 \vee -1{,}6 \vee -1{,}5 \vee -1{,}5 \vee 1{,}5) = 1{,}5$$

Por lo tanto, $\tau(X_1) = 1{,}1 \geq 0$ corresponde a la dendrita D_{11} (índice de C^1) así $y(X_1) = 1$, el patrón de entrada es clasificado dentro de la clase C^1 como era de esperarse. Por otra parte, $\tau(X_2) = 1{,}5 \geq 0$ corresponde a la dendrita D_{32} (índice de C^3) así $y(X_2) = 3$, el patrón de entrada es correctamente clasificado dentro de la clase C^3. Si se diera el caso de que el valor de la neurona (τ) no es mayor o igual a cero, entonces el patrón no es clasificado dentro de ninguna clase. Para mayor información consultar la referencia [29].

3.3. Circuitos de lógica reconfigurable

Los primeros diseño digitales, en donde se utilizarón circuitos discretos fue con el uso de circuitos integrados de funcionalidad fija. Al hacer uso de este tipo de componentes, se daba por sentado que la operación de los mismos se encontraba completamente definida por el fabricante además de que su manufactura estaba fuertemente restringida. Un ejemplo de esto, es la familia de circuitos integrados TTL 7400 con los cuáles fue posible construir los primeros equipos de cómputo.

En trabajos recientes sobre el diseño e implementacipon de sistemas digitales, hay una tendencia clara hacia el uso de dispositivos con base en lógica reconfigurable, dado que una ventaja de este tipo de dispositivos radica en que su funcionalidad no es fija, es decir, el diseñador es completamente libre de operar dicho dispositivo y hacer hardware a la medida.

3.4. *Field Programmable Gate Arrays* (FPGAs)

Los FPGAs (por sus siglas en inglés, Field Programable Gate Array) son dispositivos semiconductores que contienen bloques de lógica cuya interconexión puede ser programada en campo. La lógica programable puede reproducir desde funciones tan sencillas como una compuerta lógica o un sistema combinatorio, hasta sistemas complejos implementados en un chip. En la figura 3.6, se observa la imagén de la tarjeta que fue utilizada para esta tesis.

Figura 3.6: Modelos de FPGA

3.4.1. Características

Un dispositivo lógico programable es un circuito de propósito general el cual posee una estructura interna para poder ser modificada por el usuario, con el propósito de implementar una amplia cantidad de aplicaciones. Un FPGA, es un dispositivo semiconductor

el cual su estructura esta compuesta principalmente de celdas lógicas interconectadas por una matriz de interruptores programables (figura 3.7). Caracteristícamente, cada una de las celdas permite implementar tareas combinatorias o secuenciales, así pues un diseño puede ser implementado por medio de la descripción funcional de cada celda lógica y la correcta interconexión de los elementos involucrados.

Los FPGA comerciales emplean alguno de los siguientes elementos para la fabricación de las celdas lógicas:

- Pares de transistores.
- Compuertas lógicas NAND o XOR de dos entradas.
- Multiplexores.
- Tablas de búsqueda (LUTs).

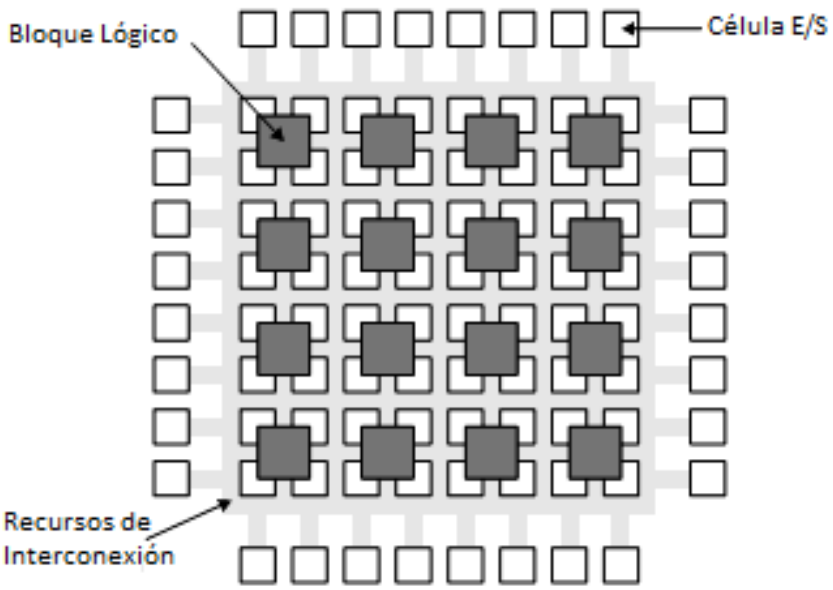

Figura 3.7: Arquitectura general de un FPGA

Para realizar la programación de los interruptores que se emplean para interconectar cada celda lógica, se hace uso principalmente de tres tecnologías:

- Memoria estática de acceso aleatorio (SRAM), el proceso de conmutación lo realiza un transistor de paso, controlado por un bit de estado, almacenado en la memoria.

- Memoria de sólo lectura, que se puede borrar y programar (EPROM), donde el proceso de conmutación puede anularse al aplicar cargas eléctricas.
- Antifusible, donde al programar el dispositivo, se forman rutas de baja resistencia.

El dispisitivo FPGA empleado para realizar el presente trabajo de tesis, pertenece a la familia DE-2 del fabricante Altera, cuenta con bloques M4K interconectado. En la figura 3.8 se muestra el diagrama de bloques del circuito reconfigurable de Altera.

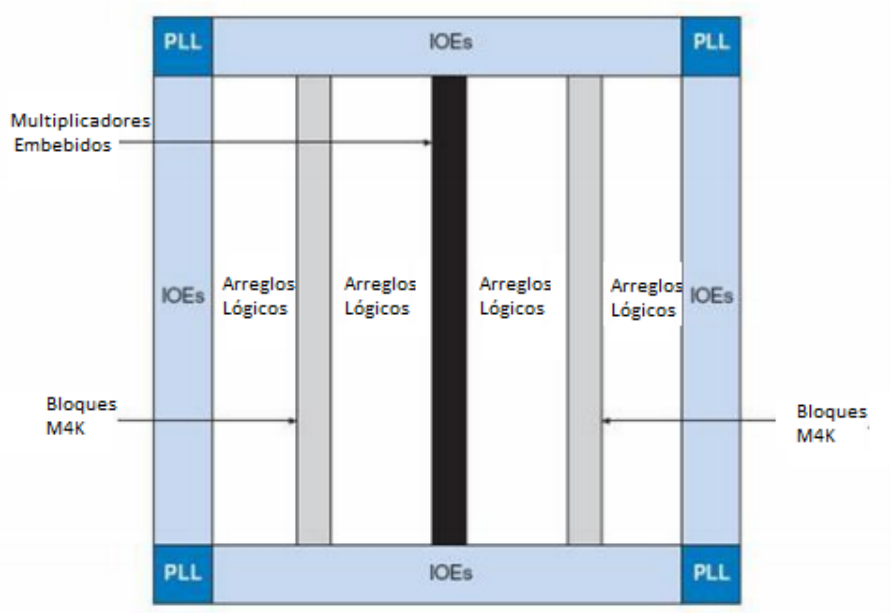

Figura 3.8: Diagrama de bloques del FPGA Altera

3.4.2. Ventajas de uso

Los dispositivos FPGA son realmente dispositivos revolucionarios, ya que mezclan beneficios que ofrecen tanto el software (flexibilidad) como el hardware (eficiencia y rendimiento) para realizar tareas computacionales. Las ventajas de uso que ofrecen los FPGA son las siguientes:

- Gran capacidad de procesamiento de información.
- Paralelismo, puede ejecutar varios procesos de manera concurrente.
- Bajo costo de prototipado.
- Reducción de tiempo de desarrollo.
- Se puede adaptar a un gran número de aplicaciones y estándares, dado que son reprogramables.

- Como son dispositivos reprogramables, ofrecen la posibilidad de reconfigurarse mientras se encuentran en funcionamiento.

3.5. Lenguajes de descripción de hardware

Inicialmente los lenguajes de descripción de hardware (HDL) estaban compuestos solamente de un conjunto de sentencias simples, con ello se podía definir la interconexión de los elementos que conformaban un diseño digital. A este tipo de lenguajes se les denominó con el nombre de *Net list*, debido a que definían un archivo que contenía un listado de conexiones.

Posteriormente, se hizo necesario un mayor nivel de abstracción para verificar la funcionalidad de los elementos que conformaban un diseño digital, debido a esto fue necesaria la creación de herramientas que permitierán simular el comportamiento de los mismos, llevando así a que los lenguajes de descripción de hardware evolucionarán hacia niveles de abstracción más altos, pudiendo obtener de esta menera modelos sintetizables, es decir, cuanto más poder tengan las sentencias de programación disponibles en el lenguaje, será posible crear diseños con mayor similitud entre la etapa de modelado y la de verificación funcional.

Existen dos lenguajes de descripción de hardware ampliamente usados: *VHDL y Verilog*. Para el desarrollo de este trabajo de tesis se hará uso del lenguaje *VHDL*.

3.5.1. Características del lenguaje de descripción VHDL

El lenguaje de descripción de hardware VHDL fue creado por el Departamento de Defensa de los Estados Unidos, con el fin de aplicar técnicas de diseño jerárquico en el desarrollo de un circuito digital y poder modelar así un sistema digital completo desde diferentes niveles de abstracción, iendo desde la descripción algorítmica de un diseño hasta el nivel de descripción por medio de la definición de compuertas lógicas.

Existen varias ventajas sobre el uso de este lenguaje VHDL, entre las cuáles se encuentran las siguientes:

- Existe una gran cantidad de tipos de datos ya predefinidos por el lenguaje.
- Cada unidad diseñada se puede compilar por separado.
- Pueden definirse procedimientos y funciones.

- Los paquetes que se definan, pueden ser utilizados por cualquier unidad.
- Constructores para el modelado de alto nivel.
- Pueden crearse librerías a partir de unidades compiladas.
- Se puede establecer el llamado concurrente a procedimientos.

Diseño en VHDL

Antes de implementar el diseño de un modelo sobre un dispositivo basado en lógica reconfigurable, es necesario llevar a cabo la etapa de diseño y simulación mediante la división de bloques abstractos del diseño completo, los cuáles son conocidos como componentes o unidades.

Una unidad de diseño entidad (*entity*) está integrada por la definición de los puertos de entrada y/o salida, mientras que la unidad de diseño arquitectura (*architecture*), permite describir la funcionalidad de la entidad. Pueden realizarse multiples descripciones funcionales de la misma unidad.

Sentencias en VHDL

En VHDL existen tres tipos de enunciados con los cuales es posible instanciar múltiples unidades de diseño, estos son mencionados a continuación:

- *Declaraciones.* Deben aparecer antes de que se realice la instancia de alguna unidad de diseño.
- *Postulados secuenciales.* Son ejecutados dependiendo del orden y de las condiciones del flujo de datos.
- *Postulados concurrentes.* Son ejecutados en paralelo, ya que se ejecutan en el mismo instante de tiempo y son independientes de otros postulados ya sean concurrentes o secuenciales.

Definición de objetos en VHDL

En VHDL es posible declarar cuatro tipos de objetos de datos:

- *Constantes.* Permite declarar un valor constante y su valor no puede variar durante la ejecución del programa.

- *Variables.* Es usada para almacenar valores de manera temporal y su valor puede cambiar en diferentes instantes de tiempo. Puede ser utilizado únicamente dentro de la definición de un proceso (*process*) y se utiliza de manera local, ya que otros procesos no pueden acceder a ella.

- *Señales.* Mantiene una lista de valores, desde el valor actual hasta valores futuros. Dentro del lenguaje de descripción de hardware, una señal puede ser vista como una conexión física, una unidad de almacenamiento simple (flip–flop) o como un esquema de almacenamiento en grupo, es decir como un registro. Se utiliza de manera global, ya que todos los procesos definidos en la arquitectura pueden acceder a las señales definidas.

- *Archivos.* Este tipo de objeto hace referencia a un sistema de archivos y no puede ser sintetizado, pero su importancia radica en que puede ser usando durante la etapa de simulación para proporcionar estímulos de entrada, y así almacenar la respuesta en un archivo de salida.

Capítulo 4

Diseño e implementación

La problemática principal de la implementación de una red neuronal morfológica con procesamiento en sus dendritas, dado el algoritmo de entrenamiento descrito en el capítulo anterior (4.2.1), radica en que al ir aumentando el número de dimensiones n, es decir, aumentar las características de las clases, su crecimiento se vuelve exponencial.

En este trabajo de tesis, se busca implementar en hardware el algoritmo de entrenamiento de una RNMD y comprobar su correcto funcionamiento. El algoritmo de entrenamiento fue implementado con lenguaje de descripción de hardware (VHDL), para su posterior síntesis en un dispositivo programable FPGA.

4.1. Diseño del algoritmo de entrenamiento de la RNMD de p clases 1 dimensión en FPGA

Al hablar de una dimensión, el algoritmo de entrenamiento de una RNMD se puede limitar a la idea de tener un conjunto de puntos de distintos tipos o clases esparcidos sobre un eje coordenado unidimensional (figura 4.1), los cuales deben delimitarse por divisiones o particiones de tal forma que cada una de esas particiones encierre a puntos de la misma clase (figura 4.2).

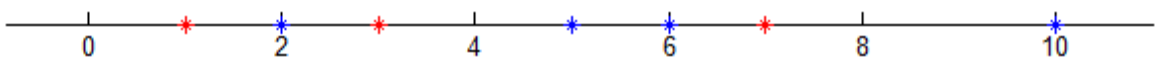

Figura 4.1: Escenario inicial para un conjunto de patrones de entrenamiento de 1 dimensión y p clases

Figura 4.2: Resultado del entrenamiento para 1 dimensión y p clases

Por lo tanto, el resultado es un conjunto de pares ordenados que representan las particiones del eje unidimensional y cada uno de ellos está asociado a una clase a la que pertenecen los datos que encierra esa partición. Este conjunto de pares ordenados y su respectiva clase asociada se representa como una matriz denominada PR, donde el número de filas es determinado por el número de particiones, y las columnas son x_{i1}, x_{i2}, c_i, siendo el par de valores de la partición i y su clase asociada respectivamente. Esta matriz PR es el resultado y objetivo del entrenamiento de la red y representará posteriormente el valor de los pesos en las dendritas de la RNMD.

El conjunto de datos dados inicialmente y su clase asociada es denominado conjunto inicial o de entrenamiento y se representa con una matriz I en la que sus filas representan cada uno de los puntos dados y sus clases asociadas. Definimos a I_j como la j–ésima fila de la matriz I.

4.1.1. Algoritmo general

Dada la matriz de entrada I

1. Se crea la Matriz PR con solo una partición que encierra el conjunto total de datos dejando un margen M alrededor de ellos, y se asocia con cualquiera de las clases del conjunto de datos de entrada.

2. Se inicializan las variables $i, j = 0$, que representarán la fila de PR y la fila de I respectivamente que se estarán usando en determinado punto del algoritmo.

3. Se inicializa la variable $clase = 0$, que representará la clase a la que pertenece la partición PR_i .

4. Se verifica si el elemento I_j pertenece o se encuentra dentro de la partición PR_i.

   4.1. Si pertenece y $clase = 0$, lo que indica que es el primer elemento de I que pertenece a la partición PR_i.

      4.1.1. Se asigna a $clase$ y a la partición PR_i el valor de la clase del elemento I_j.

   4.2. Si pertenece y $clase \neq 0$, lo que indica que ya hay otros elementos de I que pertenecen a PR.

4.2.1. Si *clase* es diferente a la clase del elemento I_j, quiere decir que hay más de una clase de elementos en la partición PR_i.

4.2.1.1. Dividir PR_i en 2^n particiones, donde n representa el número de dimensiones. PR_i será la primera partición creada y las otras se agregan como una nueva fila de PR.

4.2.1.2. Decrementar i, para revisar de nuevo la partición PR_i modificada.

4.2.1.3. Hacer $j = filas$ de I para comenzar de nuevo desde el elemento I_0.

4.3. Si $j < filas$ de I, entonces incrementar j y regresar a 4.

4.4. Si $i < filas$ de PR hacer $clase = 0, j = 0$ e incrementar i, regresar a 4.

5. Fin

El diagrama de flujo que representa el algoritmo anterior se muestra en la figura 4.3.

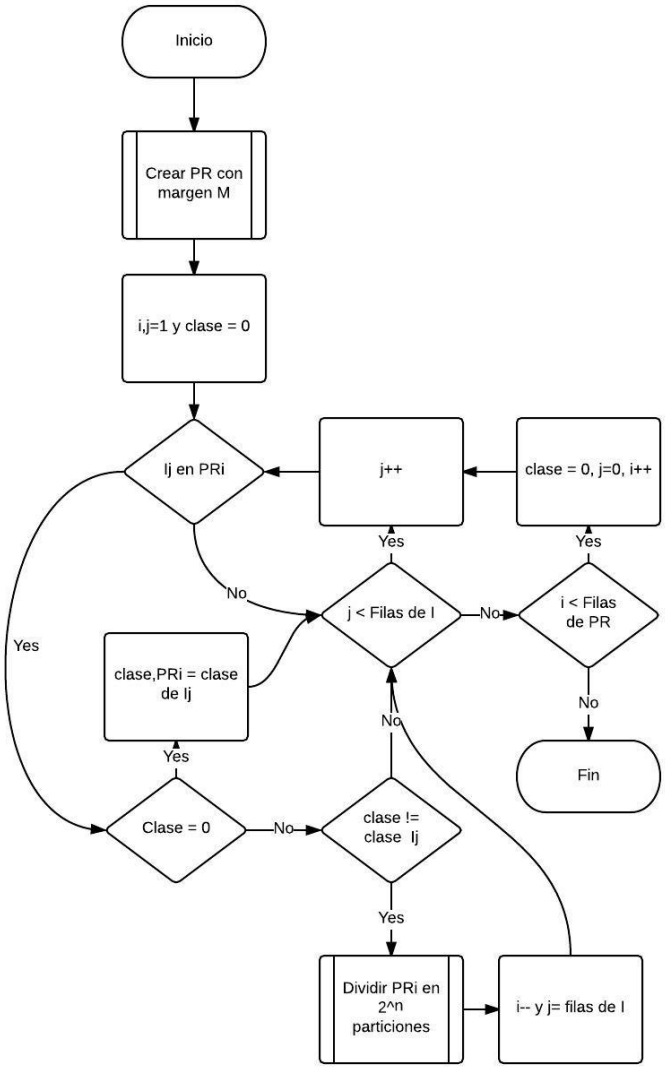

Figura 4.3: Diagrama de flujo para el algoritmo de entrenamiento en FPGA

A continuación, se presenta un ejemplo del uso del algoritmo presentado en 4.1.1. Dado el siguiente conjunto de datos de entrada (figura 4.4):

```
I =

    Dato   Clase
      1      2
      7      2
      4      1
     12      1
      9      2
     15      2
     16      3
     20      1
     25      1
     29      3
     30      1
```

Figura 4.4: Conjunto de datos de entrenamiento para 1 dimensión p clases

Se crea la matriz PR la cual contiene una sola partición y encierra a todo el conjunto de datos con un margen M:

```
PR =
     x1     x2     clase
     1      30       0
```

Figura 4.5: Matriz inicial PR

Se asigna a la partición de la matriz PR en turno la clase del primer elemento de la matriz I que pertenezca a la partición, en este caso es 2.

```
PR =
     x1     x2     clase
      1     30       0

clase =

     2
```

Figura 4.6: Asignación de clase a la matriz PR

Se continúa verificando cada elemento de la matriz I, si algún elemento pertenece a la partición y es de una clase distinta entonces se divide la matriz PR.

I =

dato	clase
1	2
7	2
4	1
12	1
9	2
15	2
16	3
20	1
25	1
29	3
30	1

PR =

x1	x2	clase
1.0000	15.5000	2.0000
15.5000	30.0000	0

Figura 4.7: Matriz *PR* particionada

Se sigue dividiendo dicha partición hasta que ya no haya más elementos de I que pertenezcan a la nueva partición y que su clase sea diferente (figura 4.8).

clase =

2

PR =

x1	x2	clase
1.0000	2.8125	2.0000
15.5000	30.0000	0
8.2500	15.5000	0
4.6250	8.2500	0
2.8125	4.6250	0

Figura 4.8: Matriz *PR* particionada 2

Siguiendo el algoritmo, se lleva a cabo el mismo procedimiento para todas las demás particiones generadas, hasta obtener la matriz resultante *PR* que es la que contiene los pesos de las dendritas de la RNMD (figura 4.9).

PR =

x1	x2	clase
1.0000	2.8125	2.0000
15.5000	19.1250	3.0000
8.2500	11.8750	2.0000
4.6250	8.2500	2.0000
2.8125	4.6250	1.0000
22.7500	26.3750	1.0000
19.1250	22.7500	1.0000
11.8750	13.6875	1.0000
26.3750	28.1875	3.0000
13.6875	15.5000	2.0000
28.1875	29.0938	3.0000
29.0938	30.0000	1.0000

Figura 4.9: Matriz *PR* resultante

Finalmente, en la figura 4.10 se puede observar como se generarón las particiones para el conjunto de datos de entrenamiento.

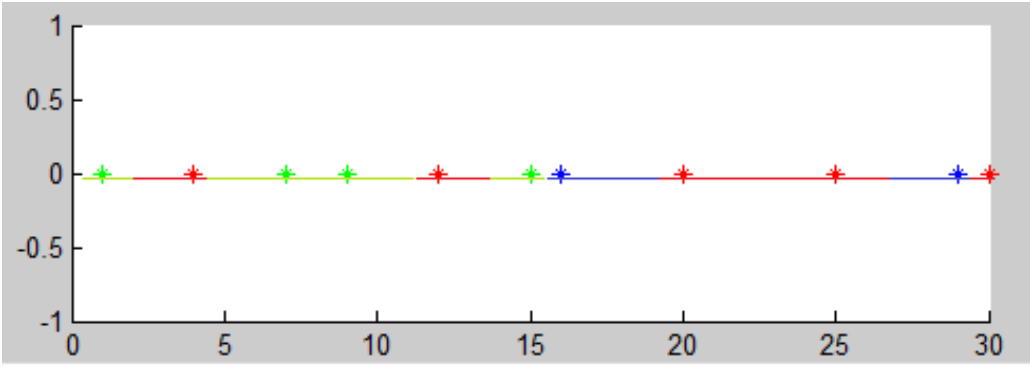

Figura 4.10: Segmentos generados a partir del algoritmo de entrenamiento

4.1.2. Implementación del algoritmo en FPGA

Si se sigue el algoritmo, lo primero que se requiere es almacenar los valores de la matriz I, es decir, el conjunto de datos de entrenamiento, para esto se hace uso de una memoria ROM la cual es un medio de almacenamiento que permite sólo la lectura de la información y no su escritura, se pretende que esta entrada sea independiente y pueda ser obtenida de una memoria o archivo externo. En este caso, fue implementada de manera interna ya que cabe la posibilidad que en trabajos futuros, se pueda mejorar esta implementación cargando los datos de entrenamiento desde una memoria externa para su rápida lectura sin necesidad de modificar el diseño. Para la matriz *PR* se usó una memoria RAM, ya

que se estará modificando constantemente durante toda la ejecución del programa hasta que se encuentren todas las particiones que definen los pesos sinápticos de las dendritas.

El ciclo completo del algoritmo consiste en resumen en recorrer PR, verificar si la partición contiene elementos de clases diferentes y si es el caso dividir en 2^n particiones agregando las particiones a la matriz PR, en esta adición de particiones a PR es donde se encuentra el primer problema de implementación en el FPGA.

Inicialmente se trató de implementar el algoritmo de entrenamiento haciendo uso de ciclos *for*, con la finalidad de ir generando dinámicamente el valor de la longitud de la matriz PR ya que no se definía un valor fijo, debido a que las particiones se generan hasta que se satisface la condición de paro. Sin embargo, este tipo de solución en una implementación descrita en lenguaje de descripción de hardware, resulta no factible ya que los índices no pueden ser variables, es decir, tienen que ser un valor constante (fig. 4.11).

```
bloque_i:process(clk,clr)
variable j:integer:=0;
begin
if clr='1' then
   indica <= (others=>'0');
elsif(clk'event and clk ='1')then
   for i in 0 to j loop
            i_en <=  MEMROM_I(i,0) + i_en;
            j := j+1;
   end loop;

   indica <= CONV_STD_LOGIC_VECTOR(i_en, 24);
end if;
```

Figura 4.11: Código de ciclo *for* con indices variables (no factible)

Al tratar de compilar el algoritmo implementado, el entorno de desarrollo arrojaba un error de sintaxis en el que se describe el problema mencionado anteriormente. Por lo tanto, para solventar esta problemática se definió a la matriz PR a un tamaño fijo.

Otra característica del algoritmo es la necesidad de tener control sobre la variable que se está iterando, ya que debe verificarse que en la partición PR_i ya no existan elementos de la matriz I (datos de entrada), que pertenezcan a la misma. Esto se logra fácilmente con un ciclo *while*, como se muestra en la figura 4.12.

```
process(clk, j)
begin
  while (i <= 10) loop
        if(i=5 and j <10)then
           i <= i - 1;
        end if;
        i <= i + 1;
        j <= j + 1;

  end loop;
end process;
```

Figura 4.12: Ciclo while en vhdl

Si se estudia detenidamente el algoritmo de entrenamiento de la RNMD, se puede observar que cada elemento a lo más puede estar en contacto con 2^n particiones, con excepción de los dos más cercanos al margen M, y es posible que en dichas particiones no exista ningún otro elemento. Considerando lo anterior se supone de manera informal que en el peor de los casos, es decir en el que todos los elementos de la matriz I son de diferente clase, y que todos ellos se encuentren en el límite de la partición de cada dimensión; entonces se generarán aproximadamente $\mid I \mid \times 2^n$ particiones (fig. 4.13). En base a este análisis se establece a la matriz PR con un tamaño inicial de $\mid I \mid \times 2^n$ filas, en donde $\mid I \mid$ es el número de fila de la matriz de datos I y n es el número de dimensiones con las que se este trabajando.

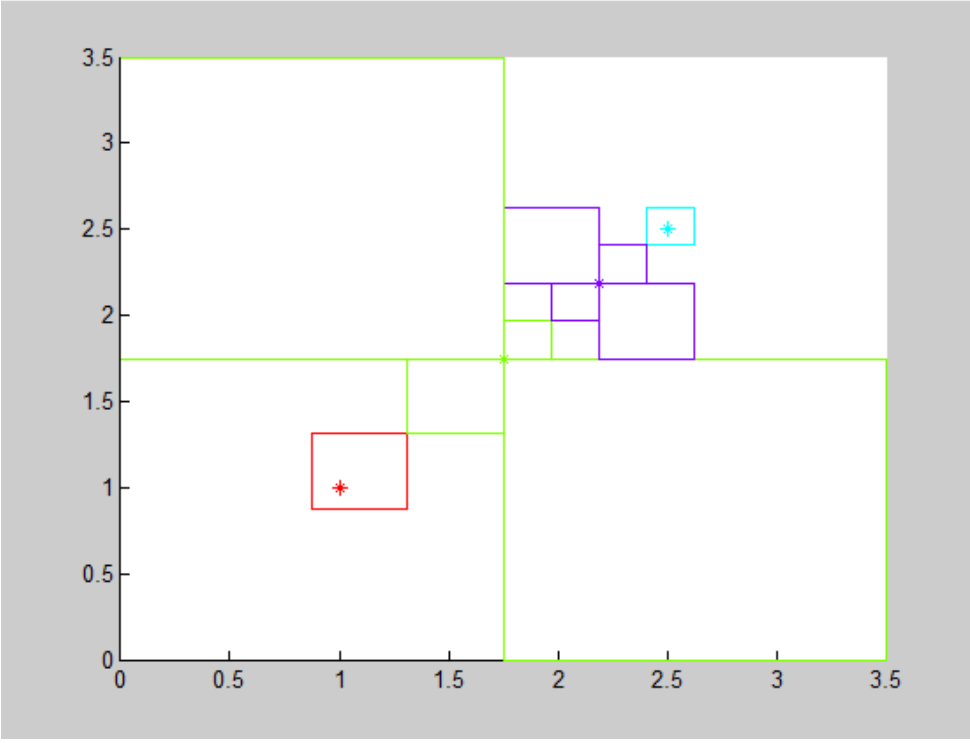

Figura 4.13: Número de particiones máximas generadas

Continuando con el algoritmo, se creó una función llamada *función de pertenencia* que es la que se encarga de verificar si la fila I_j se encuentra dentro de la partición PR_i, esta función recibe como entrada los límites de la partición para cada dimensión, en este caso serían los límites para el eje x y los límites para el eje y dado que se está trabajando con dos dimensiones, así como el valor del conjunto de datos de la fila I_j de igual manera para x y y, de esta forma se puede llamar a la *función de pertenencia* de manera iterativa en el caso de n dimensiones para así poder usarla más adelante en una algoritmo de una RNMD con p clases y n dimensiones.

Desafortunadamente en VHDL sólo se permiten ciclos *while* de hasta 10 000 iteraciones, lo que limitaría el poder de el algoritmo para un mayor número de dimensiones y de datos de entrenamiento que se pudierán usar al entrenar la RNMD, por ejemplo, si se tiene una matriz I de 200 datos y se trabaja con una RNMD de 10 dimensiones, de acuerdo a lo descrito anteriormente se necesitarán más de 200 000 iteraciones lo que no se puede lograr con la instrucción *while*.

Para resolver esta problemática fue necesario proponer un diseño en VHDL que permitiera controlar la variable sobre la cuál se está iterando al igual que lo permite el ciclo *while*, y que además no este limitado a un cierto número de iteraciones. Se diseñó un sistema basado en listas sensibles dentro de un bloque *process*, estas condiciones aprovechan la repetición infinita del bloque *process* ya que en cada repetición se evalúa la fase del ciclo anidado en la que se encuentra el contador. En la figura 4.14 se muestra un ejemplo de esta implementación:

```
bloque_i:process(clk)
variable i:integer:=0;
begin
   if(clk' event and clk ='1')then


   if(i=5 and j<10)then
      i := i - 1;
   end if;


   if(i<10)then
         i := i+1;
         j <= j+1;
   end if;


   end if;
   i_out <= i;
   j_out <= j;
end process;
```

Figura 4.14: Código que implementa la función *while* con un bloque *process*

Como se puede observar, cada repetición del ciclo interno depende de la señal de reloj(clk) y el ciclo externo depende del ciclo interno como sucede en los ciclos *for* anidados. De igual manera, es posible modificar el valor de índice que es la característica necesaria para la implementación de dicho algoritmo.

4.2. Implementación del algoritmo de entrenamiento de la RNMD de p clases 2 dimensiones en FPGA

Para este problema se toma como base el algoritmo usado para la RNMD de una dimensión, aumentando el número de columnas a la matriz PR, que ahora no sólo se compone de las columnas x_{i1}, x_{i2}, c_i, sino que se compone también de $x_{i1}, x_{i2}, y_{i1}, y_{i2}$, y c_i, que son el par de valores de la partición i de la primera dimensión, el par de valores de la partición i de la segunda dimensión y su clase asociada, respectivamente.

A partir de este momento se puede analizar la posibilidad de expandir el algoritmo a n dimensiones, pero existe un problema para la implementación en FPGA, dado que si se hablará por ejemplo de un problema con quince dimensiones, entonces siguiendo la línea de trabajo anterior se tendría que crear en la FPGA una memoria de $2^{15}x$ *longitud de* I, y en caso de entrenar la red con un conjunto de entrenamiento de 200 elementos, el tamaño de la memoria debería ser de 6,553,600 elementos lo que es sería un uso desmedido e innecesario de recursos, ya que la cantidad de elementos útiles de esos 6,553,600 con seguridad ronda la longitud de I, entendiendo elementos útiles como aquellos que representan particiones reales que contienen al menos a un elemento de I dentro de ellas.

Por tal motivo, se realizaron algunas modificaciones al algoritmo presentado anteriormente, para hacer un uso eficiente de los recursos y así tener un tamaño de PR mucho más pequeño. Para esto, se agrega una columna a la matriz PR la cuál indicará si la partición contiene sólo elementos que pertenecen a la misma clase cuando su valor sea 1 ó si aún contiene elementos de más de una clase cuando su valor sea 0, a esta columna se le denomina *check*.

4.2.1. Algoritmo con ahorro de memoria

Dada la matriz de entrada I

1. Se crea la matriz PR de tamaño fijo y se inicializa con solo una partición que encierra el conjunto total de datos dejando un margen M alrededor de ellos, y se asocia con cualquiera de las clases del conjunto de datos de entrada, además se establece el

valor de *check* de esa única partición como 0, todos los demás valores de PR son cero.

2. Se inicializan las variables $i, j = 0$, que representarán la fila de PR y la fila de I respectivamente que se estarán usando en determinado punto del algoritmo.

3. Se inicializa la variable $clase = 0$, que representará la clase a la que pertenece la partición PR_i .

4. Se verifica si la partición PR_i tiene que ser revisada, es decir si su valor $check = 0$ así como si no es una fila de ceros.

5. Se verifica si la fila I_j pertenece o se encuentra dentro de la partición PR_i

   5.1. Si pertenece y $clase = 0$, lo que indica que la primera fila de I pertenece a la partición PR_i.

      5.1.1. Se asigna a *clase* y a la partición PR_i el valor de la clase de la fila I_j .

      5.1.2. Se asigna el valor uno a check de PR_i.

   5.2. Si pertenece y $clase \neq 0$, lo que indica que otras filas de I pertenecen a PR.

      5.2.1. Sí clase es diferente a la clase la fila I_j, quiere decir que hay más de una clase de elementos en la partición PR_i.

         5.2.1.1. Dividir PR_i en 2^n particiones, donde n representa el número de dimensiones. PR_i será la primera partición creada y las otras se agregan como una nueva fila de PR.

         5.2.1.2. Guardar el valor de i en $auxi$, hacer $i = 0$, para revisar de nuevo la matriz PR modificada.

         5.2.1.3. Hacer $j = filas$ de I para comenzar de nuevo desde I_0.

   5.3. Si $j < filas$ de I, entonces incrementar j y regresar a 4.

   5.4. Si $clase = 0$ entonces la partición PR_i no tiene elementos, por lo tanto, se hace ceros la fila i para aprovechar el espacio con otra partición.

   5.5. Si $i < filas$ de PR hacer $clase = 0, j = 0, auxi = i$ e incrementar i, regresar a 4.

6. Fin

El diagrama de flujo que representa el algoritmo anterior es el mostrado en la figura 4.15

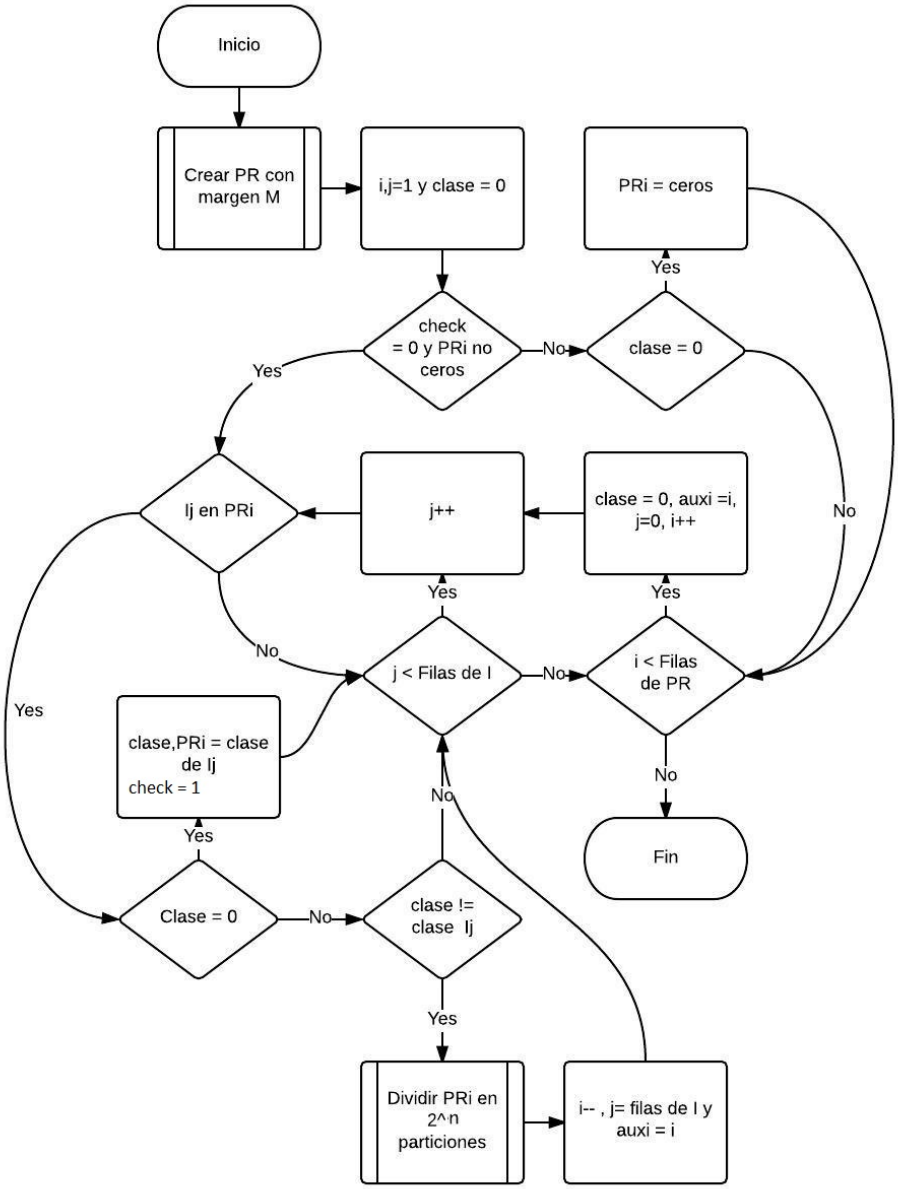

Figura 4.15: Diagrama de flujo para el algoritmo de entrenamiento con ahorro de memoria en FPGA

Dando seguimiento al algoritmo anterior, se puede observar que el ahorro de memoria radica en hacer uso de espacios en la memoria que almacenaban particiones que no contenían ningún dato dentro de ellas, de esta manera dichas particiones se eliminan poniéndolas a cero, para que posteriormente al hacer el recorrido a través de la matriz PR, estas puedan ser utilizadas almacenando una partición que si representa un dato real, dentro de los resultados obtenidos por el algoritmo de entrenamiento.

A continuación, se presenta un ejemplo del uso del algoritmo presentado en 4.2.1. Dado el siguiente conjunto de datos de entrada (figura 4.16) el cuál representa a un XOR:

```
I =
     x     y     clase
     0     0       1
     0     1       2
     1     0       2
     1     1       1
```

Figura 4.16: Conjunto de datos de entrenamiento para 2 dimensiones p clases

Se crea la matriz PR de tamaño fijo, la cual contiene una sola partición y encierra a todo el conjunto de datos con un margen M:

```
PR =
     x1    x2    y1    y2   clase  check
    -1     2    -1     2     1      1
     0     0     0     0     0      0
     0     0     0     0     0      0
     0     0     0     0     0      0
     0     0     0     0     0      0
     0     0     0     0     0      0
     0     0     0     0     0      0
     0     0     0     0     0      0
     0     0     0     0     0      0
```

Figura 4.17: Matriz inicial PR

Se procede a verificar que la partición almacenada en la fila j de PR pueda ser checada, es decir que $check = 0$ y que además no sea una fila de ceros, una vez hecho lo anterior, se revisa cada fila de I para revisar si el dato se encuentra dentro de la partición y de ser así verificar que su clase sea la misma a la almacenada en *clase*, si la clase no es la misma se proceda a dividir la PR (figura 4.18).

```
I =
     x     y     clase
     0     0       1
     0     1       2
     1     0       2
     1     1       1

clase =

     1
```

```
PR =
     x1    x2    y1    y2   clase  check
    -1     2    -1     2     0      0
     0     0     0     0     0      0
     0     0     0     0     0      0
     0     0     0     0     0      0
     0     0     0     0     0      0
     0     0     0     0     0      0
     0     0     0     0     0      0
     0     0     0     0     0      0
     0     0     0     0     0      0
```

```
PR =

    x1         x2         y1         y2         clase      check
-1.0000    0.5000   -1.0000    0.5000         0          0
-1.0000    0.5000    0.5000    2.0000         0          0
 0.5000    2.0000   -1.0000    0.5000         0          0
 0.5000    2.0000    0.5000    2.0000         0          0
      0         0         0         0         0          0
      0         0         0         0         0          0
      0         0         0         0         0          0
      0         0         0         0         0          0
      0         0         0         0         0          0
```

Figura 4.18: Matriz *PR* particionada XOR

Siguiendo el algoritmo, se lleva a cabo el mismo procedimiento para todas las demás particiones generadas, hasta obtener la matriz resultante *PR* que es la que contiene el pesos de las dendritas de la RNMD (figura 4.19).

```
PR =

    x1         x2         y1         y2         clase      check
-1.0000    0.5000   -1.0000    0.5000    1.0000    1.0000
-1.0000    0.5000    0.5000    2.0000    2.0000    1.0000
 0.5000    2.0000   -1.0000    0.5000    2.0000    1.0000
 0.5000    2.0000    0.5000    2.0000    1.0000    1.0000
      0         0         0         0         0          0
      0         0         0         0         0          0
      0         0         0         0         0          0
      0         0         0         0         0          0
      0         0         0         0         0          0
```

Figura 4.19: Matriz *PR* resultante

Finalmente, en la figura 4.20 se puede observar como se generarón las particiones para el conjunto de datos de entrenamiento.

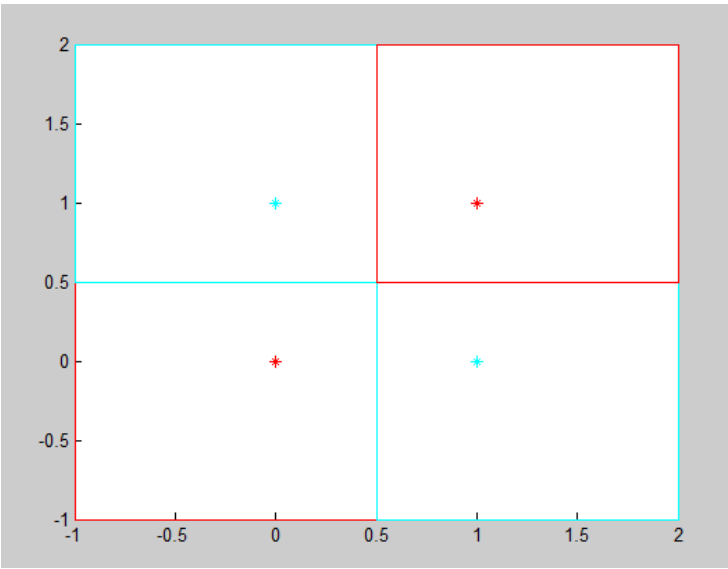

Figura 4.20: Cajas generadas a partir del algoritmo de entrenamiento

4.2.2. Reducción del número de dendritas

Si bien las particiones encontradas en la matriz PR, son correctas y puede crearse la *RNMD* con dichas particiones tomándolas como los pesos de las dendritas de la red, se puede realizar una reducción y optimización de este resultado, ya que existen particiones cuyos límites de $n-1$ dimensiones son iguales, a estas particiones se les llamó *particiones contiguas* y serán enlazadas en un ciclo hasta que no existan más de estas particiones. A continuación, se decribe el algoritmo implementado.

Dada la matriz PR:

1. Inicializar una variable $i = 1$ que indica la partición PR_i que se está comparando y $j = i + 1$ que indica la partición PR_j contra la que que se está comparando PR_i.
2. Si PR_i no es una fila de ceros o vacía
   2.1. Si PR_i es contigua a PR_j.
      2.1.1. Unir PR_i y PR_j en PR_i, y hacer PR_j ceros.
      2.1.2. Regresar a 1
   2.2. Incrementar j e ir a 2.1
3. Incrementar i e ir a 2.
4. FIN

El diagrama de flujo que representa el algoritmo anterior es el mostrado en la figura 4.21

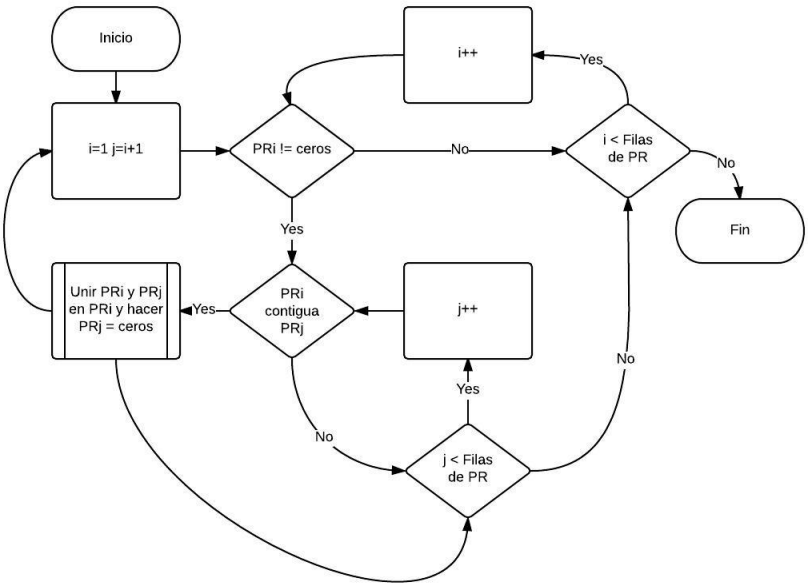

Figura 4.21: Diagrama de flujo para el algoritmo de reducción del número de dendritas

Un ejemplo de lo descrito anteriormente se pude observar en la fig. 4.22. Se puede observar que en la figura de la derecha ya no existen las divisiones entre el área C3 y C1.

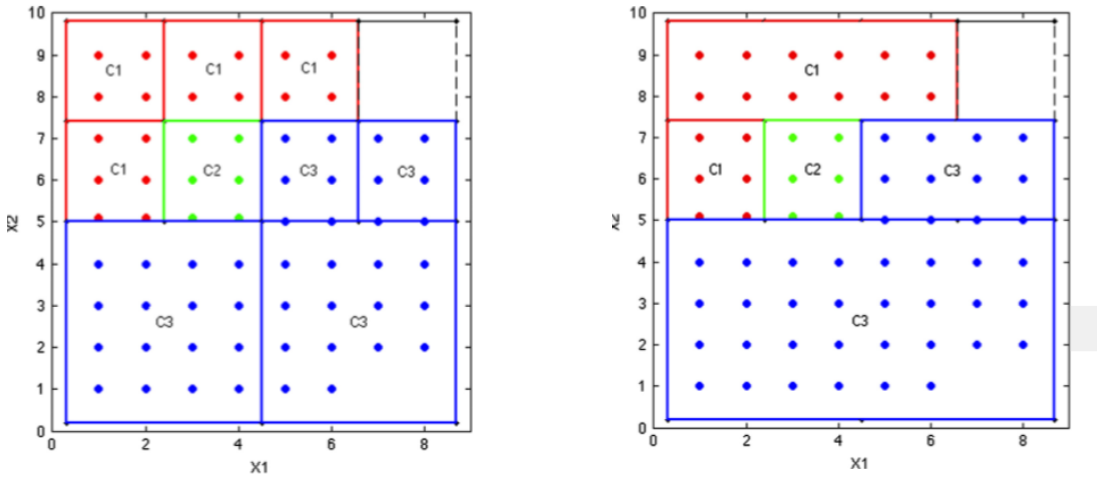

Figura 4.22: Unión de las cajas generadas para minimizar el número de dendritas generadas

4.3. Diseño del algoritmo de entrenamiento de la RNMD de p clases n dimensiones en FPGA

El algoritmo diseñado para la implementación de la RNMD y su entrenamiento en un FPGA, está pensado para funcionar de la misma forma para 1, 2 ó n dimensiones, salvo algunas generalizaciones y adecuaciones que se tienen que realizar.

Lo primero es considerar el aumento en el tamaño de las matrices, el número de columnas de la matriz I sería $2^n + 1$ y el de la matriz PR seria $2^n + 2$, realmente esta consideración no conlleva más problemas al momento de la implementación.

De manera general, la base del algoritmo está pensada para trabajar con n dimensiones, solamente hay que fijar la atención en dos de los pasos de este algoritmo para poder extenderlo de manera adecuada.

El primer paso a modificar es el 5 del algoritmo descrito en la sección 4.2.1 definido como: *Si la fila I_j pertenece a PR_i*; esté paso es mapeado directamente como una función en VHDL que recibe como parámetros de entrada el valor de la fila I_j, el valor de la partición PR_i y el número de dimensión que se está revisando, de esta forma para que funcione con una RNMD de n dimensiones sólo se tiene que llamar n veces a la función de pertenencia, y verificar que en todas las llamadas a la función se obtenga un resultado verdadero, lo que significa que el elemento I_j pertenece a la partición PR_i.

El segundo aspecto a tomar en cuenta es el de dividir PR_i en 2^n particiones, ya que en las implementaciones realizadas se acomodarón manualmente los índices de las particiones para una y dos dimensiones. Ejemplo, creación de particiones para dos dimensiones:

```
for k=1:length(PR(:,1))
    if PR(k,1) ==  PR(k,2)
        PR(k,1) = PAUX(1,3);
        PR(k,2) = PAUX(1,4);
        PR(k,3) = PAUX(2,3);
        PR(k,4) = PAUX(2,4);
        PR(k,5) = 0;
        PR(k,6) = 0;
        break;
    end
end
```

En el ejemplo anterior se muestra el fragmento de código en el que se crean las particiones para 2 dimensiones, se puede observar que el número de columnas se expresa manualmente del 1 al 6 lo que cambia para n dimensiones, también se observa una matriz $PAUX$, usada para almacenar los valores que se asignarán a las nuevas particiones, los índices de la matriz $PAUX$ son asignados también manualmente.

A continuación, se muestra un ejemplo de la implementación de la división de PR_i en 2^n particiones.

```
for  m=1:2^Ndim-1
    for k=1:LPR
        if PR(k,1) ==  PR(k,2)
            bin = dec2bin(m,3);
            for  f=1:Ndim
                PR(k,2*f-1) = PAUX(f,c(str2num(bin(f))+1,1));
                PR(k,2*f) = PAUX(f,c(str2num(bin(f))+1,2));
            end
            PR(k,CLASE) = 0;
            PR(k,CHECK) = 0;
            break;
        end
   end
end
```

Esta implementación no fue realizada en este trabajo de tesis, debido a que algunas cuestiones técnicas con el FPGA no se pudierón solucionar. Lógicamente al aumentar el número de dimensiones, se incrementaba la complejidad y el tiempo de sintésis, por lo que la computadora dejaba de responder. En un trabajo futuro se propone trabajar con recursos de cómputo más potente para poder eliminar esta limitante.

Capítulo 5

Resultados experimentales

En este capítulo se describen las pruebas realizadas para comprobar el correcto funcionamiento del algoritmo de entrenamiento de la red neuronal morfológica con procesamiento en sus dendritas en un dispositivo FPGA. Para esto, se tomarón diversos conjuntos de datos de entrenamiento para una y dos dimensiones, con sus respectivos datos de prueba, para verificar que efectivamente la red está siendo entrenada correctamente.

5.1. Prueba 1: implementación del algoritmo de entrenamiento de la RNMD de p clases y 1 dimensión

Se tomó como base de prueba un conjunto de datos como se muestra en la figura 5.1, con el cual se procedió a realizar el entrenamiento de la RNMD de una dimensión y p clases. Los datos en color verde pertenecen a la clase 2, los datos en color rojo a la clase 1 y, finalmente los datos en color azul a la clase 3.

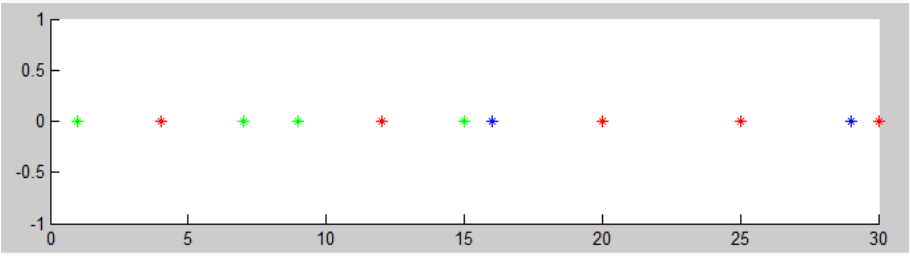

Figura 5.1: Conjunto de datos de 1 dimensión para entrenar RNMD

La matriz PR resultante, es la que se muestra en la figura 5.2, en la que se puede apreciar que la particiones se realizarón correctamente. Una vez teniendo la matriz de particiones

es necesario realizar un prueba en la que se pueda observar que dado un valor x, este pueda ser clasificado de manera correcta. El valor x=*3.2*, será el dato a probar, como se puede apreciar en la matriz de particiones este dato se encuentra dentro de la partición número 5, que encierra datos dentro del rango 2.8125 a 4.6250 y que pertenecen a la clase 1, como era de esperarse.

PR =

X1	X2	Clase	Check
1.0000	2.8125	2.0000	1.0000
15.5000	19.1250	3.0000	1.0000
8.2500	11.8750	2.0000	1.0000
4.6250	8.2500	2.0000	1.0000
2.8125	4.6250	1.0000	1.0000
22.7500	26.3750	1.0000	1.0000
19.1250	22.7500	1.0000	1.0000
11.8750	13.6875	1.0000	1.0000
29.0938	30.0000	1.0000	1.0000
13.6875	15.5000	2.0000	1.0000
28.1875	29.0938	3.0000	1.0000

Figura 5.2: Matriz PR resultante del entrenamiento de la RNMD de 1 dimensión p clases

En la figura 5.3 se muestran los resultados obtenidos, que como se dijo anteriormente este dato sería clasificado dentro de la clase 1, que es a la clase que pertenece. La matriz llamada *resultado* almacena los valores obtenidos al aplicar la operación de máximo a los resultados derivados de aplicar la operación mínimo para cada una de las dendritas. El valor máximo será el que determine a que clase pertenece el dato de prueba, ya que representa el número de dendrita en la cuál se clasificó.

resultado =

Dato	
1.0000	-0.3875
2.0000	-12.3000
3.0000	-5.0500
4.0000	-1.4250
5.0000	0.3875
6.0000	-19.5500
7.0000	-15.9250
8.0000	-8.6750
9.0000	-25.8938
10.0000	-10.4875
11.0000	-24.9875

max_j =

0.3875

Figura 5.3: Resultado obtenido dado un dato de prueba para la RNMD

Los datos anteriores fueron extraídos de la implementación del algoritmo de entrenamiento y de la RNMD en FPGA. Para ambos casos se muestran los resultados obtenidos al llevar a cabo la simulación en el *Model Sim* de *Altera*. En la figura 5.4, se muestran los valores para la matriz PR, que son los valores presentados anteriormente en la figura 5.2, estos datos representan los pesos de las dendritas obtenidas a partir del algoritmo de entrenamiento. Cabe mencionar que los datos fueron tratados como números enteros, dado que el procesamiento y la velocidad de las operaciones resulta ser más eficiente de esta manera que con el uso de números de punto flotante.

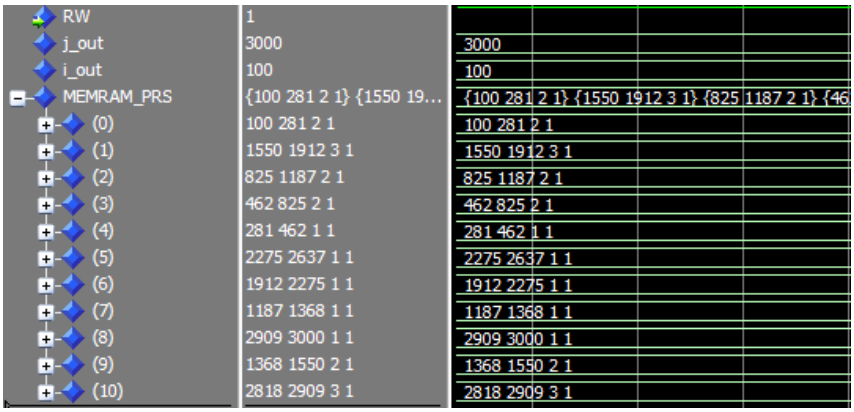

Figura 5.4: Matriz PR resultante del entrenamiento de la RNMD de 1 dimensión p clases en FPGA

La matriz *PR* se encuentra representada por la señal *MEMRAM_PRS*. También se muestran los resultados obtenidos para la prueba de pertenencia de un valor dado, a alguna de las dendritas que representan a una clase del conjunto de datos. La figura 5.5 muestra la simulación de los resultados obtenidos en FPGA, que es el mismo caso de ejemplo mencionado anteriormente.

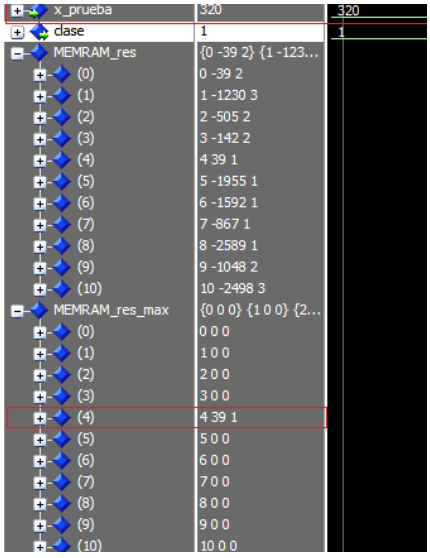

Figura 5.5: Resultado obtenido dado un dato de prueba para la RNMD en FPGA

En las figuras 5.6, 5.7 se muestra el reporte de compilación para la implementación del entrenamiento y de la red neuronal morfológica con procesamiento en sus dendritas.

Flow Summary	
Flow Status	Successful - Tue May 12 22:24:34 2015
Quartus II 64-Bit Version	14.1.0 Build 186 12/03/2014 SJ Web Edition
Revision Name	trainingRNMD2
Top-level Entity Name	trainingRNMD2
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	4,266 / 114,480 (4 %)
Total combinational functions	4,264 / 114,480 (4 %)
Dedicated logic registers	798 / 114,480 (< 1 %)
Total registers	798
Total pins	27 / 529 (5 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

Figura 5.6: Reporte de compilación para entrenamiento de RNMD

Flow Summary	
Flow Status	Successful - Tue May 12 20:35:09 2015
Quartus II 64-Bit Version	14.1.0 Build 186 12/03/2014 SJ Web Edition
Revision Name	pruebaRNMD
Top-level Entity Name	pruebaRNMD
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	1,578 / 114,480 (1 %)
Total combinational functions	1,578 / 114,480 (1 %)
Dedicated logic registers	0 / 114,480 (0 %)
Total registers	0
Total pins	24 / 529 (5 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

Figura 5.7: Reporte de compilación para implementación de RNMD

Finalmente, en la figura 5.8 se puede observar el tiempo que tarda el algoritmo implementado en FPGA para lograr entrenar el conjunto de datos de entrenamiento descrito anteriormente. En cada ciclo de reloj se van generando los valores parciales, que son resultado de las operaciones descritas por el algoritmo y que finalmente permitirán obtener los pesos sinápticos de las dendritas de la RNMD. Para este caso se toma como referencia el reloj de 50Mhz con el que trabaja la tarjeta de desarrollo, el cuál genera ciclos de reloj de 20ns.

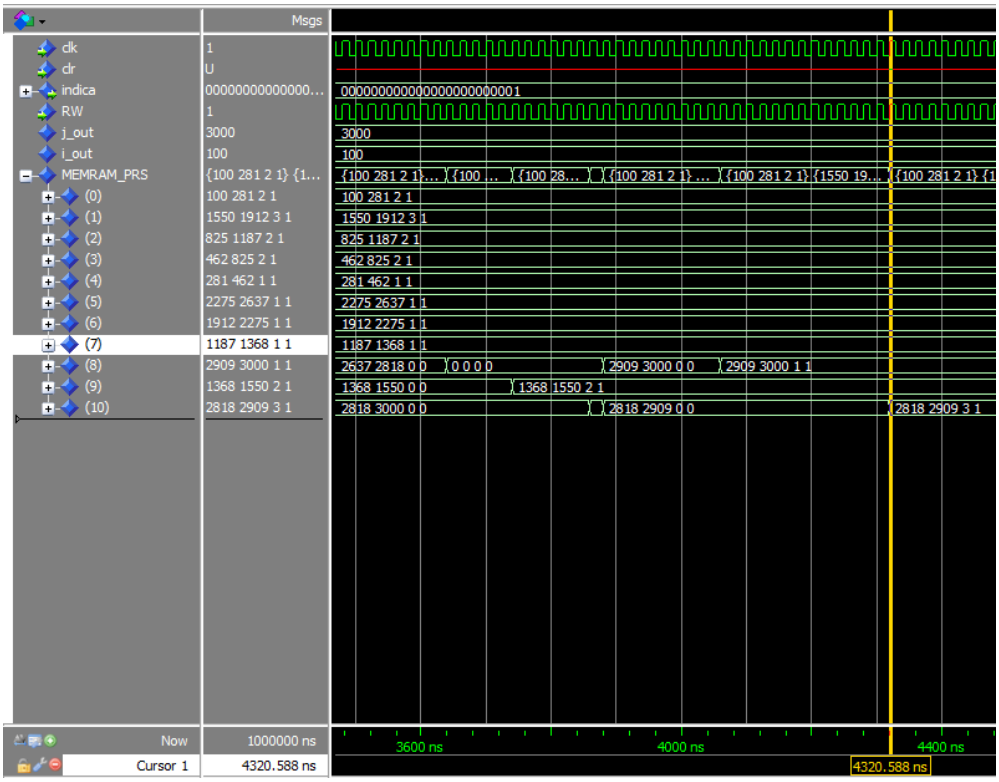

Figura 5.8: Tiempo de entrenamiento para la RNMD de p clases 1 dimensión

Como se puede observar, el tiempo de entrenamiento es de 0.00432ms el cual esta marcado con la fraja color amarillo en la figura 5.9. El tiempo de entrenamiento resulta ser mucho menor que el arrojada por la implementación en MATLAB, el cuál resulta ser de 92.87ms.

```
PR =

    1.0000    1.5938    2.0000    1.0000
   10.5000   15.2500    3.0000    1.0000
    5.7500    6.9375    2.0000    1.0000
    3.3750    5.7500    2.0000    1.0000
    2.1875    3.3750    1.0000    1.0000
    1.5938    2.1875    3.0000    1.0000
   15.2500   20.0000    1.0000    1.0000
    8.1250   10.5000    3.0000    1.0000
    6.9375    8.1250    1.0000    1.0000

Elapsed time is 0.092878 seconds.
```

Figura 5.9: Tiempo de entrenamiento para la RNMD de p clases 1 dimensión MATLAB

5.2. Prueba2: implementación del algoritmo de entrenamiento de la RNMD de p clases y 2 dimesiones en FPGA

En esta sección se muestran las pruebas realizadas con cuatro conjuntos de datos de entrenamiento. Se obtuvierón resultados equivalentes al hacer pruebas para MATLAB y FPGA, con la implementación del algoritmo descrito en el capítulo anterior (4.2.1). Se reporta el porcentaje de clasificación acertada así como el tiempo de entrenamiento para cada uno de ellos.

5.2.1. Primer conjunto de prueba

El primer conjunto de datos de entrenamiento es el que se muestra en la figura 5.10 el cuál consta de 52 valores, y tiene datos pertenecientes a dos clases diferentes. Como se puede observar, este conjunto es linealmente no separable.

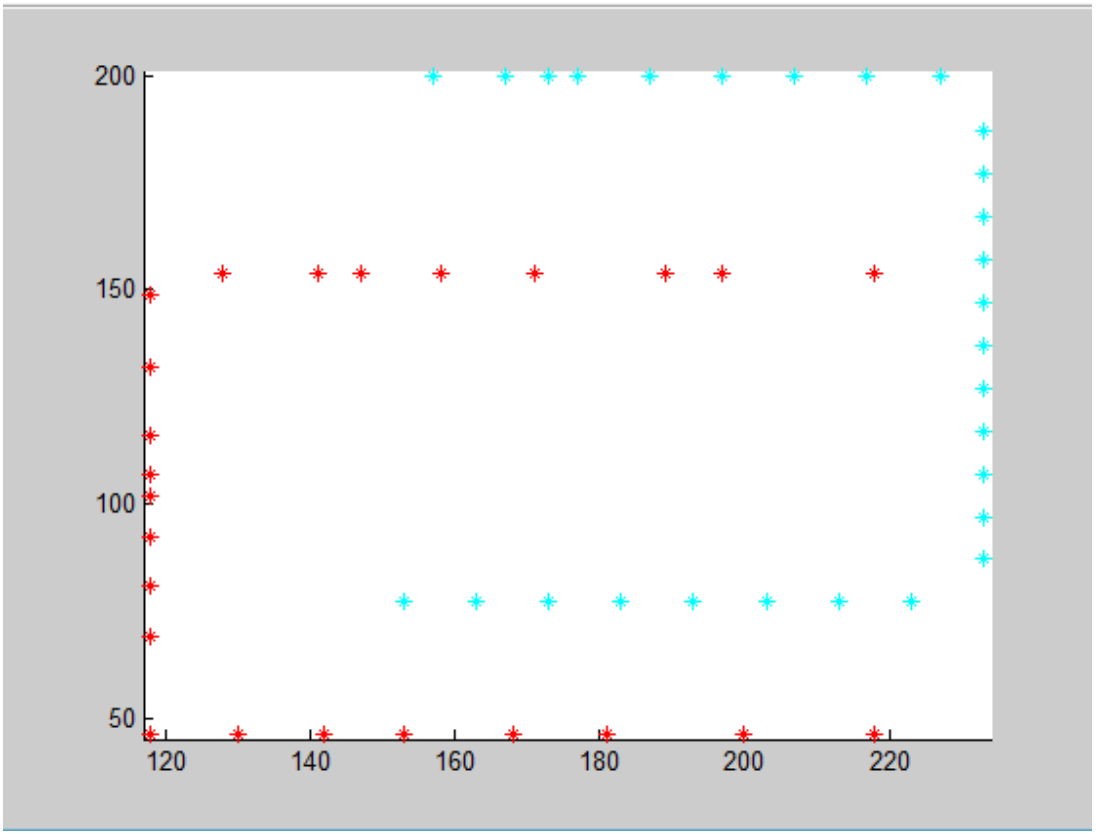

Figura 5.10: Conjunto de datos de entrenamiento 1

Una vez aplicado el algoritmo de entrenamiento, los resultados obtenidos en MATLAB son los mostrados en la figura 5.11, la cual muestra a la matriz *PR* que es en donde se almacena el valor de los pesos sinápticos de las dendritas generadas.

PR =

X1	X2	Y1	Y2	Clase	Check
117.0000	146.2500	45.0000	84.0000	1.0000	1.0000
117.0000	146.2500	123.0000	162.0000	1.0000	1.0000
175.5000	190.1250	45.0000	64.5000	1.0000	1.0000
175.5000	204.7500	123.0000	162.0000	1.0000	1.0000
117.0000	146.2500	84.0000	123.0000	1.0000	1.0000
146.2500	160.8750	45.0000	64.5000	1.0000	1.0000
204.7500	219.3750	64.5000	84.0000	2.0000	1.0000
204.7500	219.3750	142.5000	162.0000	1.0000	1.0000
146.2500	175.5000	123.0000	162.0000	1.0000	1.0000
146.2500	175.5000	162.0000	201.0000	2.0000	1.0000
219.3750	234.0000	64.5000	84.0000	2.0000	1.0000
204.7500	219.3750	45.0000	64.5000	1.0000	1.0000
204.7500	234.0000	84.0000	123.0000	2.0000	1.0000
175.5000	190.1250	64.5000	84.0000	2.0000	1.0000
190.1250	204.7500	45.0000	64.5000	1.0000	1.0000
190.1250	204.7500	64.5000	84.0000	2.0000	1.0000
175.5000	204.7500	162.0000	201.0000	2.0000	1.0000
0	0	0	0	0	0
204.7500	234.0000	162.0000	201.0000	2.0000	1.0000
146.2500	160.8750	64.5000	84.0000	2.0000	1.0000
160.8750	175.5000	45.0000	64.5000	1.0000	1.0000
160.8750	175.5000	64.5000	84.0000	2.0000	1.0000
219.3750	234.0000	123.0000	142.5000	2.0000	1.0000
219.3750	234.0000	142.5000	162.0000	2.0000	1.0000

Figura 5.11: Matriz de particiones generada por Matlab 1

La matriz *PR* esta conformada por seis columnas, las primeras cuatro definen los margenes de cada hiper–cubo generado, la quinta columna es la clase a la que pertenece, y finalmente en la sexta solumna se define a la bandera *check* con la cual se verifica si la partición generada contiene datos dentro de ella o no, de no contener datos, como se definió en el algoritmo de entrenamiento con ahorro de memoria, será posible utilizar dicha partición para almacenar un nuevo valor de dendrita válido.

Se llevó a cabo el entrenamiento de la RNMD en el FPGA con las mismas condiciones dadas en MATLAB, obteniendo los mismos resultados tal y como se muestra en la figura 5.12, en la cual se define de igual manera una matriz *PR* conformada por las mismas columnas. Cabe señalar que en el FPGA todas las operaciones fueron realizadas con números enteros ajustando los datos flotantes del conjunto de entrenamiento, multiplicandolos por 1×10^m en donde m es el número de decimales de la mantisa.

j_out	23300
i_out	11800
MEMRAM_PRS	{11700 14625 4500 8400 1 1} {11700 14625 12300 16200 1 ...
(0)	11700 14625 4500 8400 1 1
(1)	11700 14625 12300 16200 1 1
(2)	17550 19012 4500 6450 1 1
(3)	17550 20475 12300 16200 1 1
(4)	11700 14625 8400 12300 1 1
(5)	14625 16087 4500 6450 1 1
(6)	20475 21937 6450 8400 2 1
(7)	20475 21937 14250 16200 1 1
(8)	14625 17550 12300 16200 1 1
(9)	14625 17550 16200 20100 2 1
(10)	21937 23400 6450 8400 2 1
(11)	20475 21937 4500 6450 1 1
(12)	20475 23400 8400 12300 2 1
(13)	17550 19012 6450 8400 2 1
(14)	19012 20475 4500 6450 1 1
(15)	19012 20475 6450 8400 2 1
(16)	17550 20475 16200 20100 2 1
(17)	0 0 0 0 0 0
(18)	20475 23400 16200 20100 2 1
(19)	14625 16087 6450 8400 2 1
(20)	16087 17550 4500 6450 1 1
(21)	16087 17550 6450 8400 2 1
(22)	21937 23400 12300 14250 2 1
(23)	21937 23400 14250 16200 2 1
(24)	0 0 0 0 0 0
(25)	0 0 0 0 0 0

Figura 5.12: Matriz de particiones generada en FPGA

Posteriormente, se graficaron los hiper–cubos generados a partir del entrenamiento de la red con el conjunto de datos, en la figura 5.13 se muestra el resultado obtenido. Como se puede observar, cada uno de los datos queda dentro de un hiper–cubo, comprobando así la perfecta clasificación del conjunto de datos de entrenamiento.

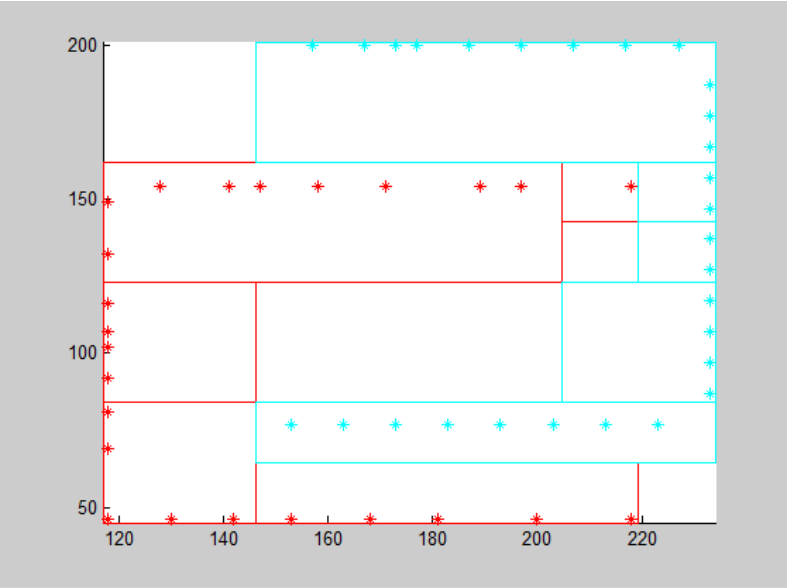

Figura 5.13: Conjunto de datos con los hiper–cubos generados del entrenamiento

Una vez entrenada la RNMD y habiendo obtenido ya los pesos sinápticos de las dendritas, se probó la red con un conjunto de datos de prueba el cual consta de un total de 189 valores, obteniendo un porcentaje del 93 % de clasificación acertada. En la figura 5.14 se observa como es que los datos fueron clasificados, los datos en color azul representan los valores que fueron asignados a una clase distinta a la que pertenecen.

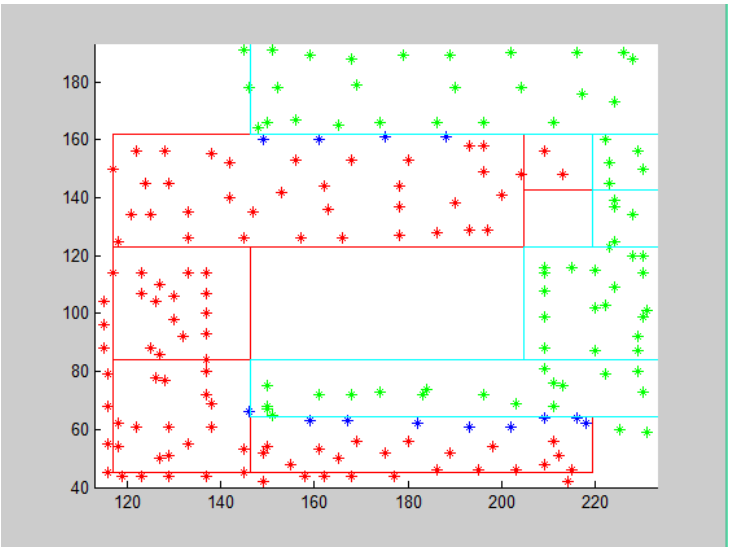

Figura 5.14: Clasificación del conjunto de datos de prueba

Se presenta en la figura 5.15, los resultados obtenidos de la simulación de la implementación de la RNMD una vez dada la matriz de pesos y el conjunto de datos de prueba. En la matriz *MEMRAM_res*, se almacena en la primera columna el número de dato a probar y en la segunda la clase a la que fue clasificado. Estos datos son los gráficados en la imagen 5.14.

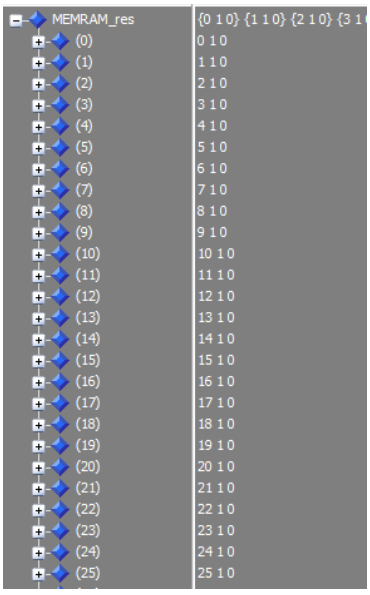

Figura 5.15: Simulación de la clasificación del conjunto de datos de prueba en FPGA

Finalmente, se muestra en 5.16 el tiempo de entrenamiento obtenido en FPGA para este conjunto de datos, de igual manera se toma como referencia el reloj de 50Mhz con el que trabaja la tarjeta de desarrollo, el cuál genera ciclos de reloj de 20ns. El tiempo de entrenamiento oscila entre los 0.0352ms, lo cual es un tiempo mucho menor que el obtenido en MATLAB el cuál es de alrededor de 53.319ms.

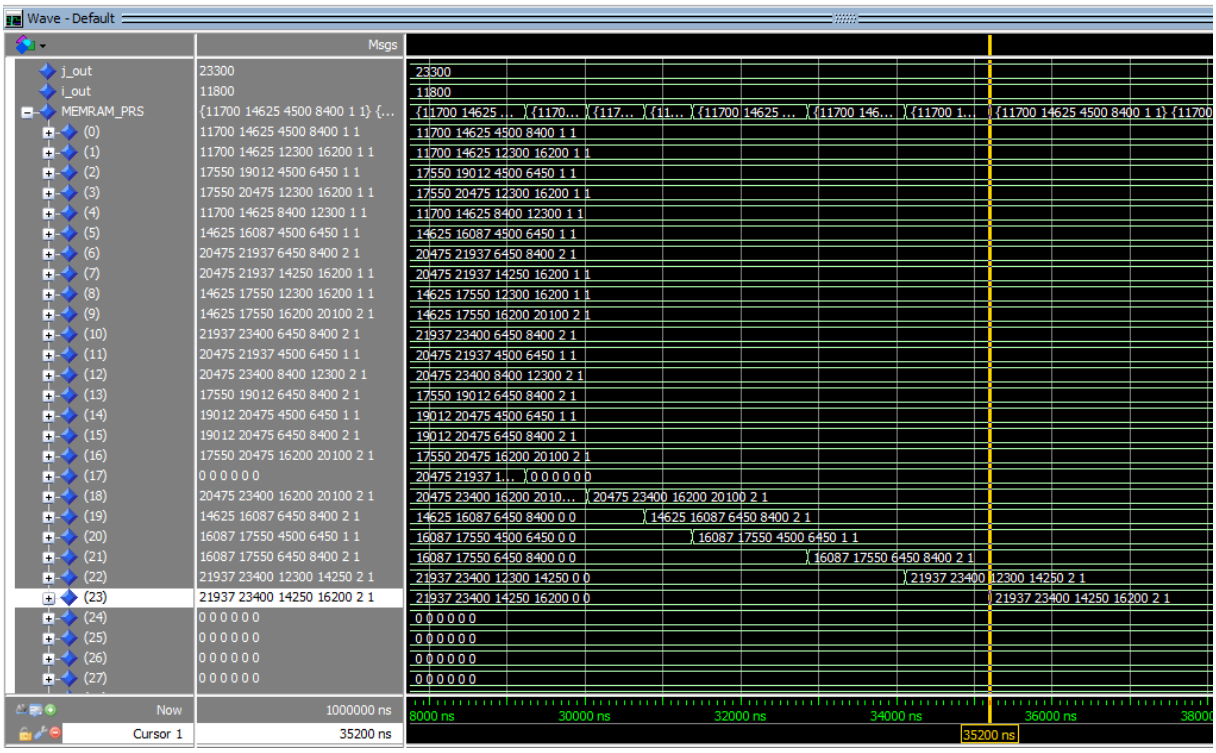

Figura 5.16: Tiempo de entrenamiento para la RNMD de p clases 2 dimensiones

A continuación, se presentan las pruebas realizadas en otros tres conjuntos de datos de entrenamiento, obteniendo de igual manera resultados satisfactorios.

5.2.2. Segundo conjunto de prueba

El segundo conjunto de datos de entrenamiento es el que se muestra en la figura 5.17 el cual consta de 66 valores. Puede observarse que es un conjunto linealmente no separable y que esta conformado por datos de 3 clases distintas.

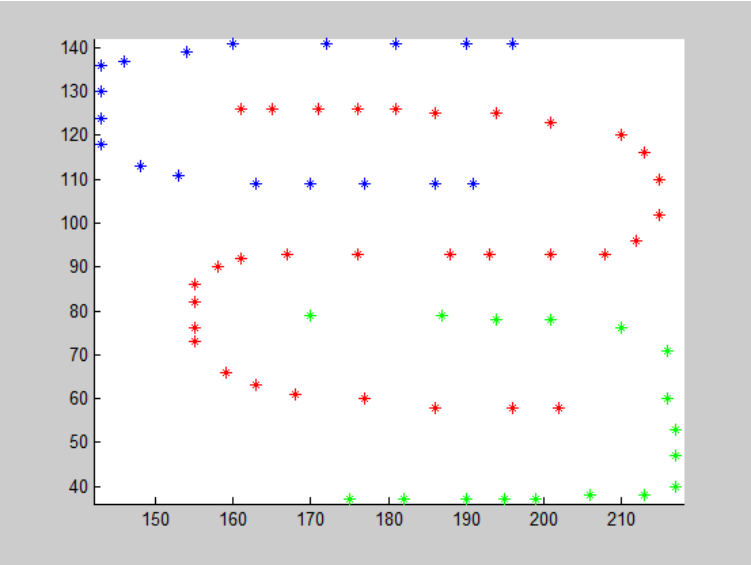

Figura 5.17: Conjunto de datos de entrenamiento (2)

Una vez aplicado el algoritmo de entrenamiento, los resultados obtenidos en MATLAB son los que se muestran en la figura 5.18

PR =

X1	X2	Y1	Y2	Clase	Check
142.0000	151.5000	115.5000	128.7500	3.0000	1.0000
180.0000	199.0000	62.5000	89.0000	2.0000	1.0000
180.0000	189.5000	36.0000	49.2500	2.0000	1.0000
180.0000	189.5000	89.0000	102.2500	1.0000	1.0000
142.0000	161.0000	62.5000	89.0000	1.0000	1.0000
161.0000	170.5000	75.7500	89.0000	2.0000	1.0000
161.0000	170.5000	62.5000	75.7500	1.0000	1.0000
161.0000	170.5000	89.0000	102.2500	1.0000	1.0000
161.0000	170.5000	115.5000	128.7500	1.0000	1.0000
142.0000	151.5000	128.7500	142.0000	3.0000	1.0000
151.5000	161.0000	115.5000	128.7500	1.0000	1.0000
151.5000	161.0000	128.7500	142.0000	3.0000	1.0000
142.0000	151.5000	102.2500	115.5000	3.0000	1.0000
151.5000	161.0000	89.0000	102.2500	1.0000	1.0000
151.5000	161.0000	102.2500	115.5000	3.0000	1.0000
199.0000	208.5000	36.0000	49.2500	2.0000	1.0000
199.0000	218.0000	62.5000	89.0000	2.0000	1.0000
180.0000	189.5000	49.2500	62.5000	1.0000	1.0000
189.5000	199.0000	36.0000	49.2500	2.0000	1.0000
189.5000	199.0000	49.2500	62.5000	1.0000	1.0000
180.0000	189.5000	115.5000	128.7500	1.0000	1.0000
199.0000	218.0000	89.0000	115.5000	1.0000	1.0000
199.0000	218.0000	115.5000	142.0000	1.0000	1.0000
180.0000	189.5000	102.2500	115.5000	3.0000	1.0000
189.5000	199.0000	89.0000	102.2500	1.0000	1.0000
189.5000	199.0000	102.2500	115.5000	3.0000	1.0000
161.0000	170.5000	49.2500	62.5000	1.0000	1.0000
170.5000	180.0000	36.0000	49.2500	2.0000	1.0000
170.5000	180.0000	49.2500	62.5000	1.0000	1.0000
0	0	0	0	0	0
0	0	0	0	0	0

Figura 5.18: Matriz de particiones generada por Matlab (2)

Se llevó a cabo el entrenamiento de la red en el FPGA con las mismas condiciones dadas en MATLAB, obteniendo los mismos resultados tal y como se muestra en la figura 5.19

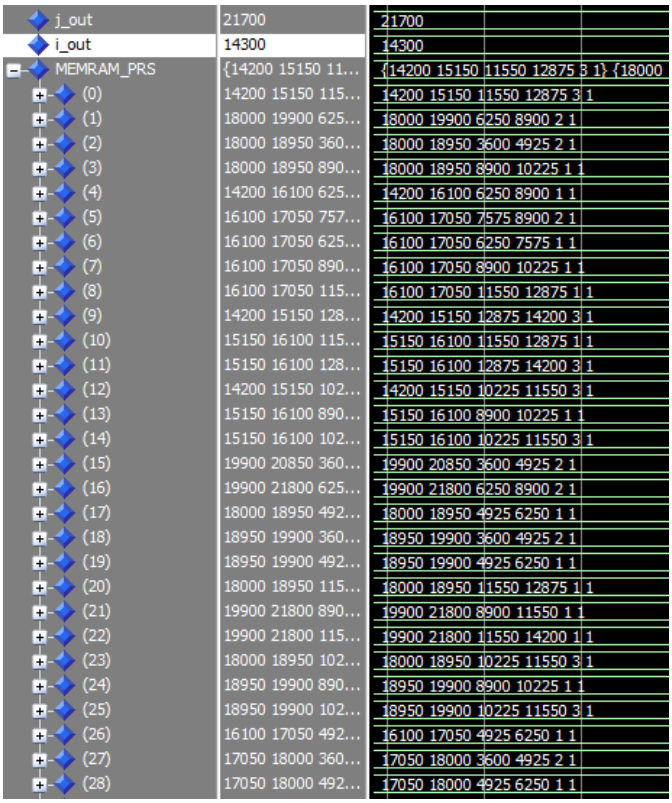

Figura 5.19: Matriz de particiones generada en FPGA (2)

Posteriormente, se graficaron los hiper–cubos generados a partir del entrenamiento de la red con el conjunto de datos, tal y como se muestra en la figura 5.20.

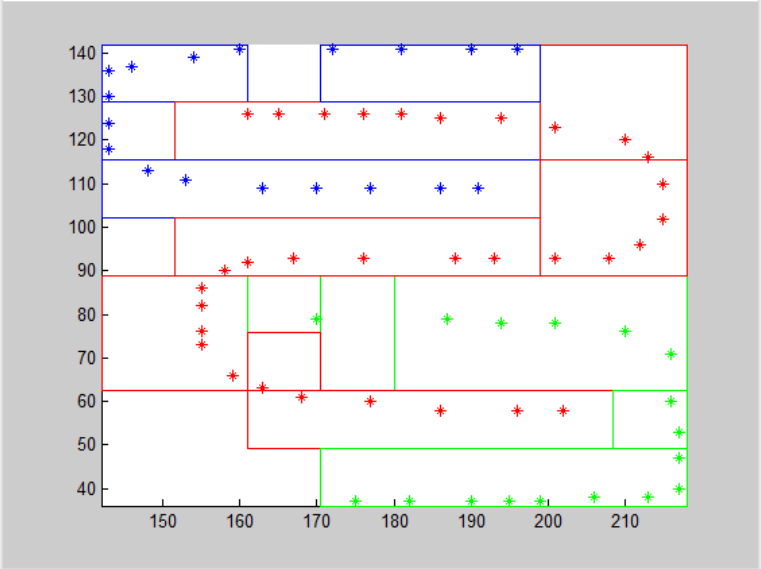

Figura 5.20: Conjunto de datos con los hiper–cubos generados del entrenamiento (2)

Una vez entrenada la RNMD y habiendo obtenido ya los pesos sinápticos de las dendritas, se probó la red con un conjunto de datos de prueba el cual tiene un total de 271 valores obteniendo un porcentaje del 97.41 % de clasificación acertada. En la figura 5.21 se observa como es que los datos han sido clasificados, los datos en color morado representan los valores que fueron asignados a una clase distinta a la que pertenecen.

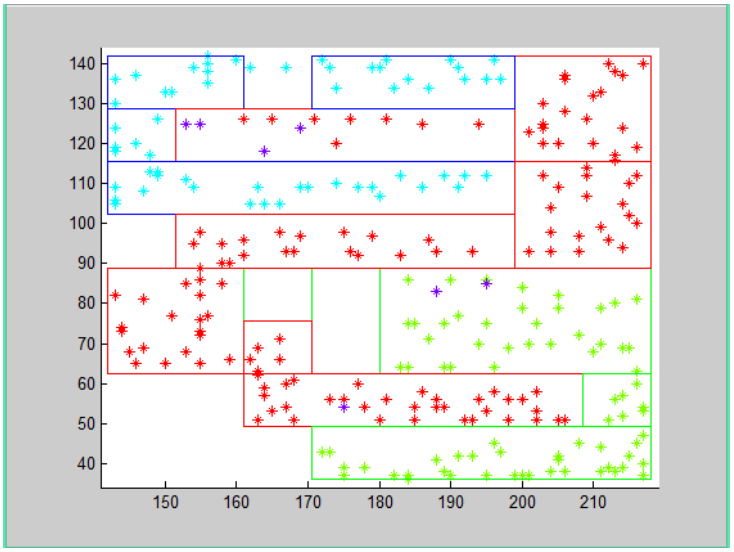

Figura 5.21: Clasificación del conjunto de datos de prueba (2)

Se presenta en la figura 5.22, los resultados obtenidos de la simulación de la implementación de la RNMD una vez dados la matriz de pesos y el conjunto de datos de prueba. En la matriz *MEMRAM_res*, se almacena en la primera columna el número de dato a probar y en la segunda la clase a la que fue clasificado. Estos datos son los graficados en la imagen 5.21.

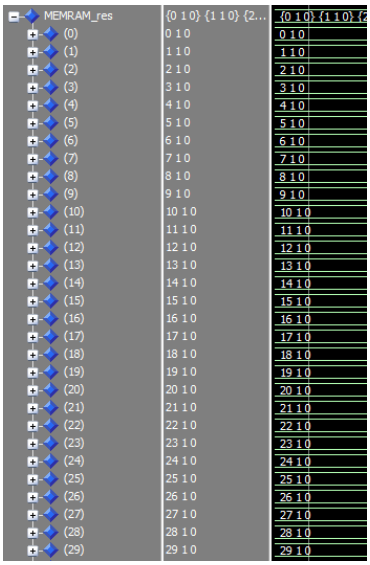

Figura 5.22: Simulación de la clasificación del conjunto de datos de prueba en FPGA (2)

Finalmente, se muestra en 5.23 el tiempo de entrenamiento obtenido en FPGA para este conjunto de datos, de igual manera se toma como referencia el reloj de 50Mhz con el que trabaja la tarjeta de desarrollo, el cual genera ciclos de reloj de 20ns. El tiempo de entrenamiento oscila entre los 0.078ms, lo cual es un tiempo mucho menor que el obtenido en MATLAB el cual es de alredor de 84.69ms.

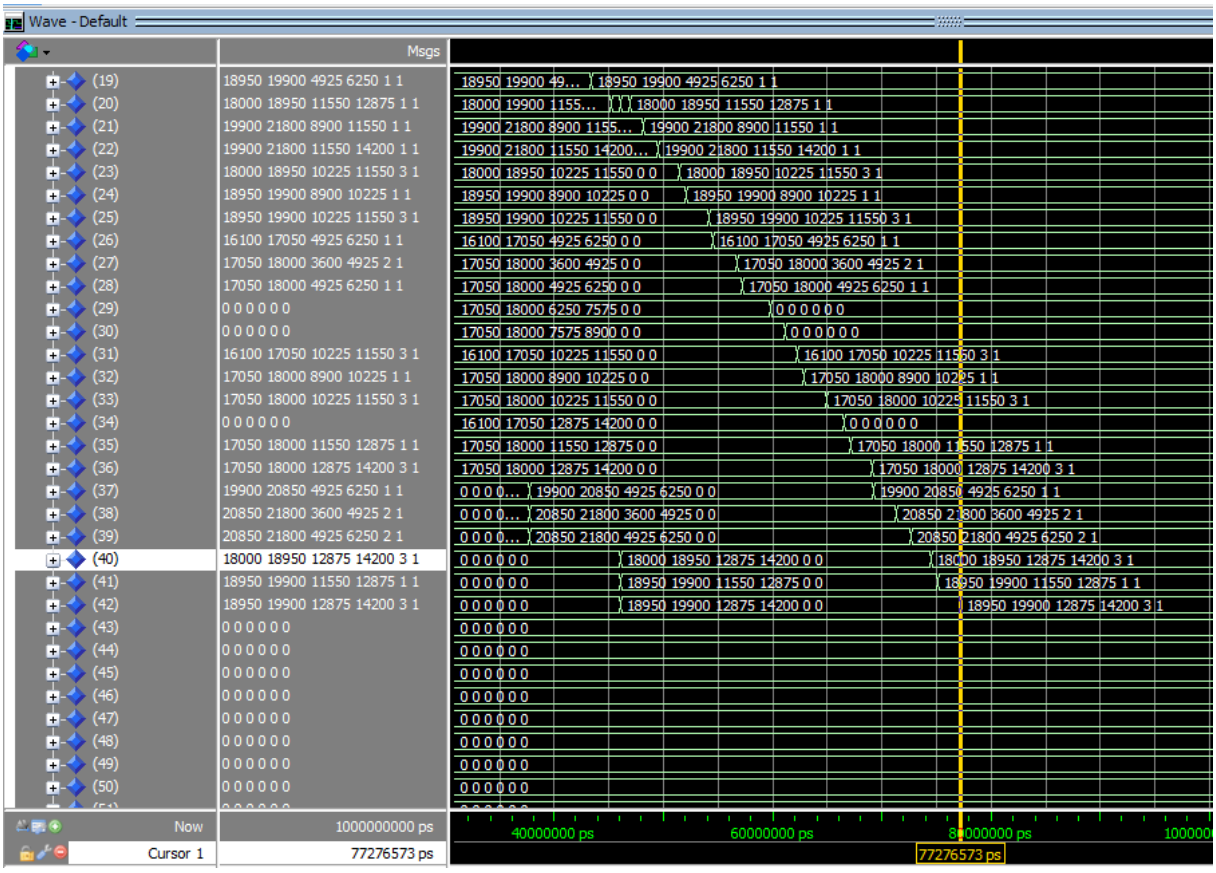

Figura 5.23: Tiempo de entrenamiento para la RNMD de p clases 2 dimensiones (ejemplo 2)

5.2.3. Tercer conjunto de prueba

El tercer conjunto de datos de entrenamiento es el que se muestra en la figura 5.24 el cual consta de 124 valores. Puede observarse que es un conjunto linealmente no separable y que está conformado por datos de 2 clases distintas.

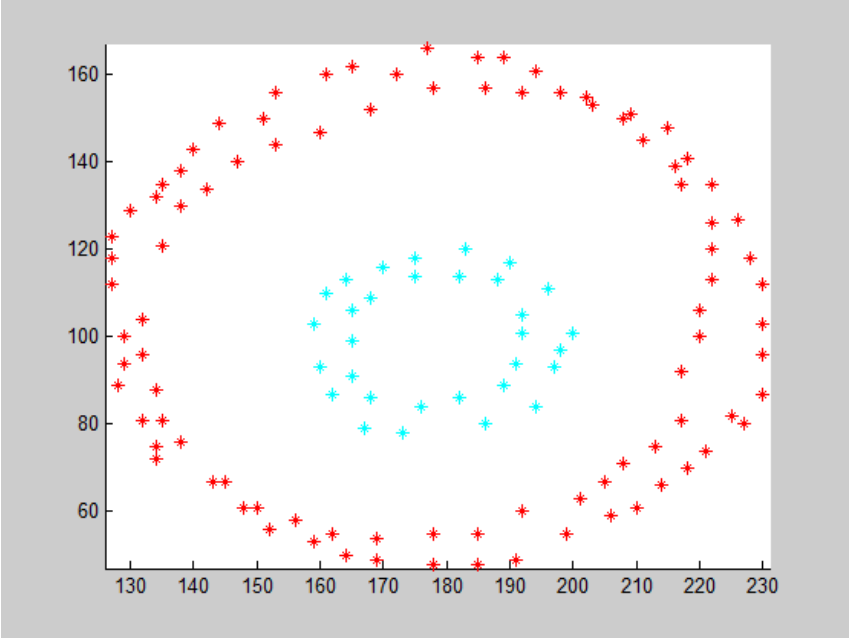

Figura 5.24: Conjunto de datos de entrenamiento (3)

Una vez aplicado el algoritmo de entrenamiento, los resultados obtenidos en MATLAB son los que se muestran en la figura 5.25.

PR =

X1	X2	Y1	Y2	Clase	Check
126.0000	152.2500	47.0000	77.0000	1.0000	1.0000
126.0000	152.2500	107.0000	137.0000	1.0000	1.0000
178.5000	204.7500	47.0000	77.0000	1.0000	1.0000
178.5000	204.7500	107.0000	137.0000	2.0000	1.0000
126.0000	152.2500	77.0000	107.0000	1.0000	1.0000
152.2500	178.5000	47.0000	77.0000	1.0000	1.0000
152.2500	178.5000	77.0000	107.0000	2.0000	1.0000
126.0000	152.2500	137.0000	167.0000	1.0000	1.0000
152.2500	178.5000	107.0000	137.0000	2.0000	1.0000
152.2500	178.5000	137.0000	167.0000	1.0000	1.0000
178.5000	204.7500	77.0000	107.0000	2.0000	1.0000
204.7500	231.0000	47.0000	77.0000	1.0000	1.0000
204.7500	231.0000	77.0000	107.0000	1.0000	1.0000
178.5000	204.7500	137.0000	167.0000	1.0000	1.0000
204.7500	231.0000	107.0000	137.0000	1.0000	1.0000
204.7500	231.0000	137.0000	167.0000	1.0000	1.0000
0	0	0	0	0	0
0	0	0	0	0	0

Figura 5.25: Matriz de particiones generada por Matlab (3)

Se llevó a cabo el entrenamiento de la red en el FPGA con las mismas condiciones dadas en MATLAB, obteniendo los mismos resultados tal y como se muestra en la figura 5.26.

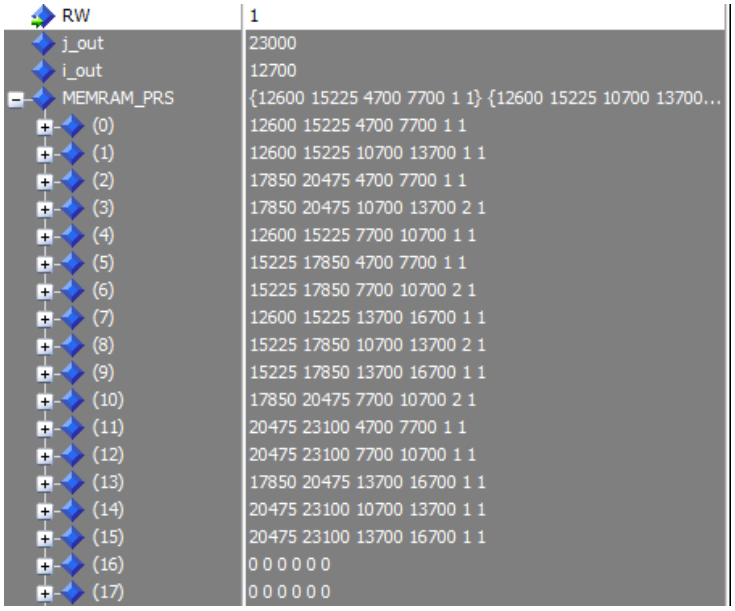

Figura 5.26: Matriz de particiones generada en FPGA (3)

Posteriormente, se graficaron los hiper–cubos generados a partir del entrenamiento de la red con el conjunto de datos, tal y como se muestra en la figura 5.27.

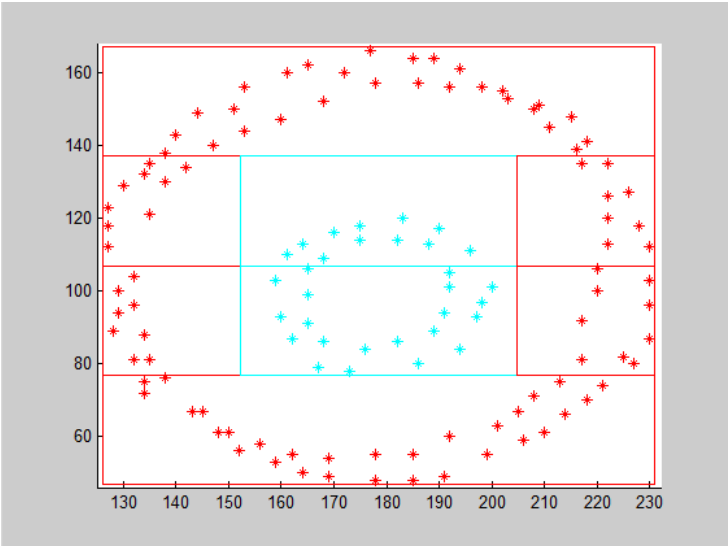

Figura 5.27: Conjunto de datos con los hiper–cubos generados del entrenamiento (3)

Una vez entrenada la RNMD y habiendo obtenido ya los pesos sinápticos de las dendritas, se probó la red con un conjunto de datos de prueba el cual tiene un total de 237 valores, obteniendo un porcentaje del 96.61 % de clasificación acertada. En la figura 5.28 se observa como es que los datos han sido clasificados, los datos en color azul representan los valores que fueron asignados a una clase distinta a la que pertenecen.

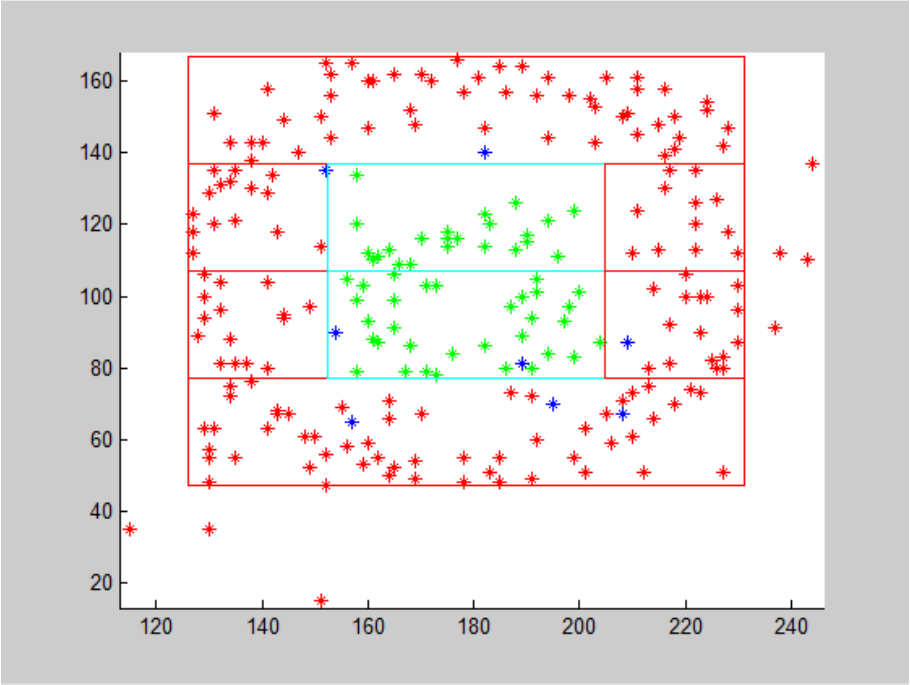

Figura 5.28: Clasificación del conjunto de datos de prueba (3)

Se presenta en la figura 5.29, los resultados obtenidos de la simulación de la implementación de la RNMD una vez dados la matriz de pesos y el conjunto de datos de prueba. En la matriz *MEMRAM_res*, se almacena en la primera columna el número de dato a probar y en la segunda la clase a la que fue clasificado. Estos datos son los graficados en la imagen 5.28.

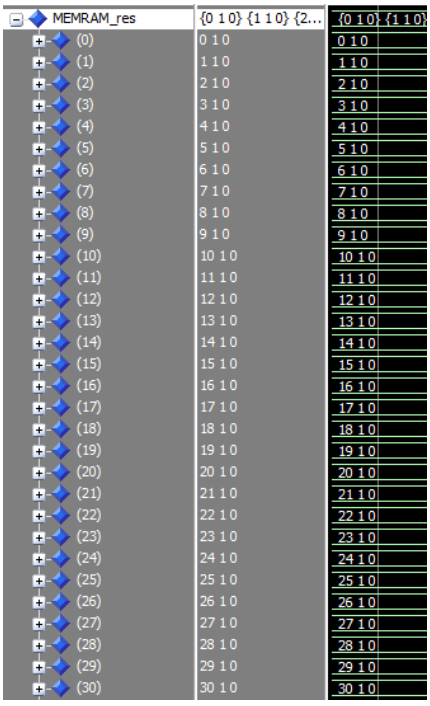

Figura 5.29: Simulación de la clasificación del conjunto de datos de prueba en FPGA (3)

Finalmente, se muestra en 5.30 el tiempo de entrenamiento obtenido en FPGA para este conjunto de datos, de igual manera se toma como referencia el reloj de 50Mhz con el que trabaja la tarjeta de desarrollo, el cual genera ciclos de reloj de 20ns. El tiempo de entrenamiento oscila entre los 0.048ms, lo cual es un tiempo mucho menor que el obtenido en MATLAB el cual es de alrededor de 64ms.

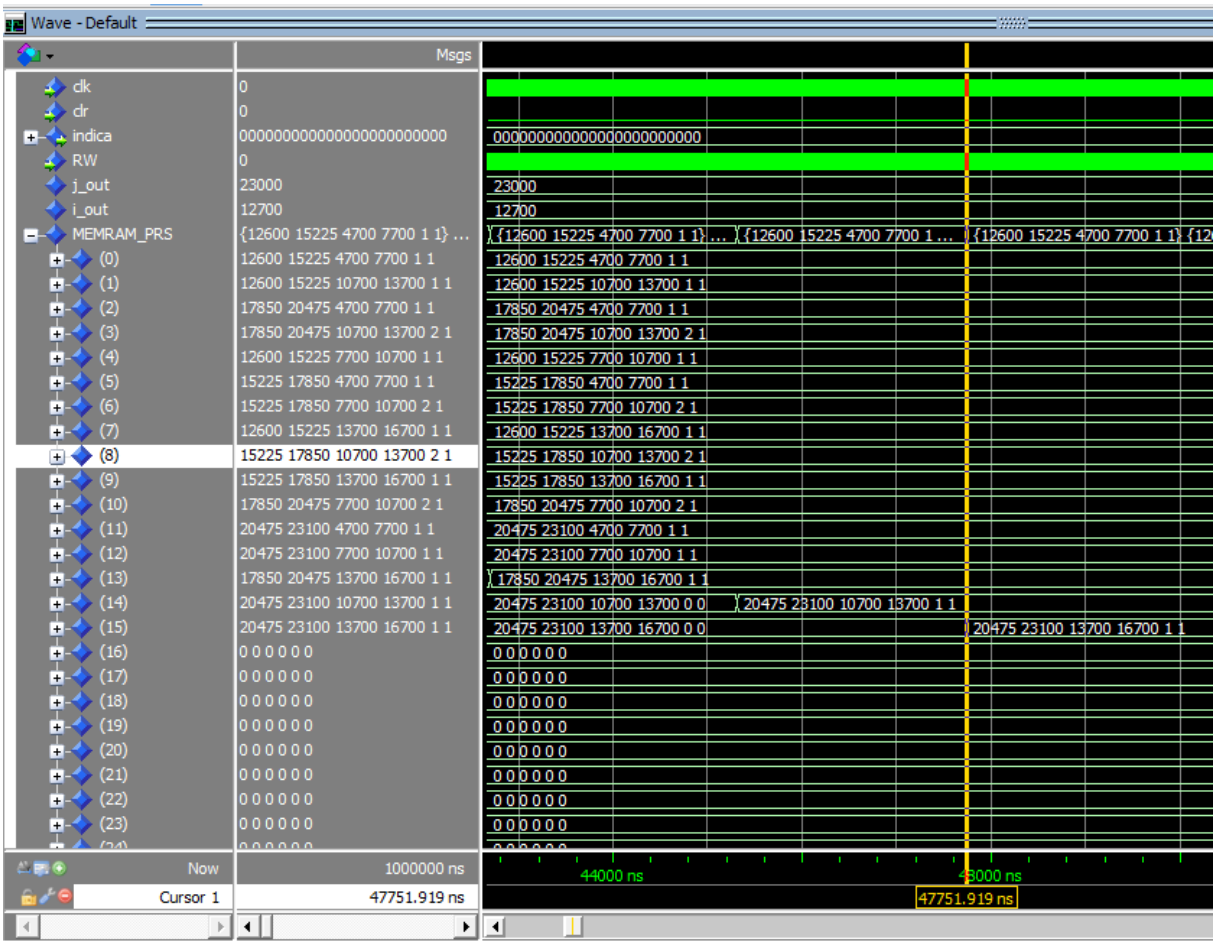

Figura 5.30: Tiempo de entrenamiento para la RNMD de p clases 2 dimensiones (ejemplo 3)

5.2.4. Cuarto conjunto de prueba

El cuarto conjunto de datos de entrenamiento es el que se muestra en la figura 5.31 el cual consta de 106 datos. Puede observarse que es un conjunto linealmente no separable y que está conformado por datos de 2 clases distintas.

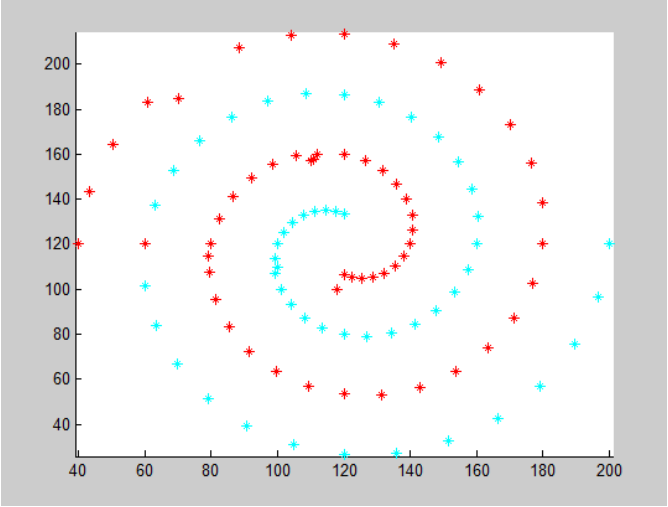

Figura 5.31: Conjunto de datos de entrenamiento (4)

Una vez aplicado el algoritmo de entrenamiento, los resultados obtenidos en MATLAB son los que se muestran en la figura 5.32.

PR =

X1	X2	Y1	Y2	Clase	Check
39.0000	79.5000	25.6667	72.8333	2.0000	1.0000
39.0000	59.2500	120.0000	143.5833	1.0000	1.0000
120.0000	140.2500	25.6667	49.2500	2.0000	1.0000
79.5000	99.7500	49.2500	72.8333	1.0000	1.0000
99.7500	120.0000	25.6667	49.2500	2.0000	1.0000
79.5000	99.7500	25.6667	49.2500	2.0000	1.0000
79.5000	99.7500	72.8333	96.4167	1.0000	1.0000
39.0000	79.5000	167.1667	214.3333	1.0000	1.0000
79.5000	99.7500	120.0000	143.5833	1.0000	1.0000
79.5000	99.7500	167.1667	190.7500	2.0000	1.0000
39.0000	59.2500	143.5833	167.1667	1.0000	1.0000
59.2500	69.3750	120.0000	131.7917	1.0000	1.0000
59.2500	79.5000	143.5833	167.1667	2.0000	1.0000
120.0000	140.2500	72.8333	96.4167	2.0000	1.0000
160.5000	201.0000	25.6667	72.8333	2.0000	1.0000
160.5000	180.7500	72.8333	96.4167	1.0000	1.0000
120.0000	140.2500	49.2500	72.8333	1.0000	1.0000
140.2500	160.5000	25.6667	49.2500	2.0000	1.0000
140.2500	160.5000	49.2500	72.8333	1.0000	1.0000
120.0000	140.2500	167.1667	190.7500	2.0000	1.0000
160.5000	180.7500	120.0000	143.5833	1.0000	1.0000
160.5000	201.0000	167.1667	214.3333	1.0000	1.0000
120.0000	140.2500	143.5833	167.1667	1.0000	1.0000
140.2500	150.3750	120.0000	131.7917	1.0000	1.0000
140.2500	160.5000	143.5833	167.1667	2.0000	1.0000
120.0000	130.1250	131.7917	143.5833	2.0000	1.0000
130.1250	140.2500	120.0000	131.7917	1.0000	1.0000
130.1250	140.2500	131.7917	143.5833	1.0000	1.0000
59.2500	79.5000	72.8333	96.4167	2.0000	1.0000
59.2500	69.3750	96.4167	108.2083	2.0000	1.0000
99.7500	120.0000	49.2500	72.8333	1.0000	1.0000

Figura 5.32: Matriz de particiones generada por Matlab (4)

Se llevó a cabo el entrenamiento de la red en el FPGA con las mismas condiciones dadas en MATLAB, obteniendo los mismos resultados tal y como se muestra en la figura 5.33.

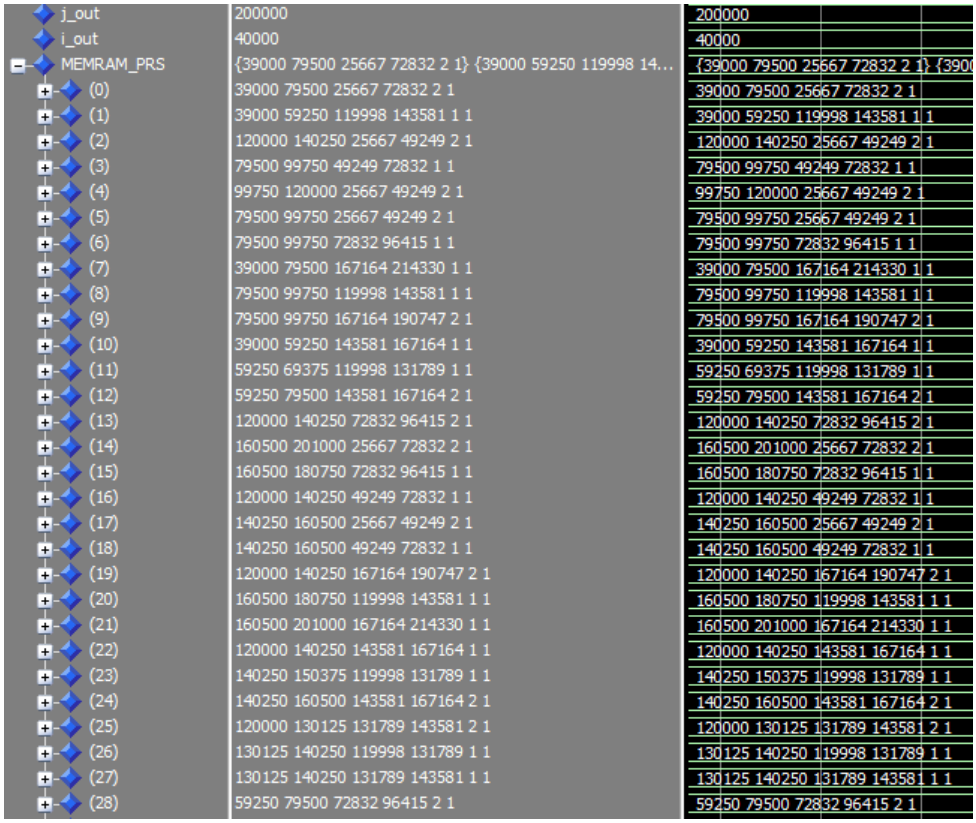

Figura 5.33: Matriz de particiones generada en FPGA (4)

Posteriormente, se graficaron los hiper–cubos generados a partir del entrenamiento de la red con el conjunto de datos, tal y como se muestra en la figura 5.34.

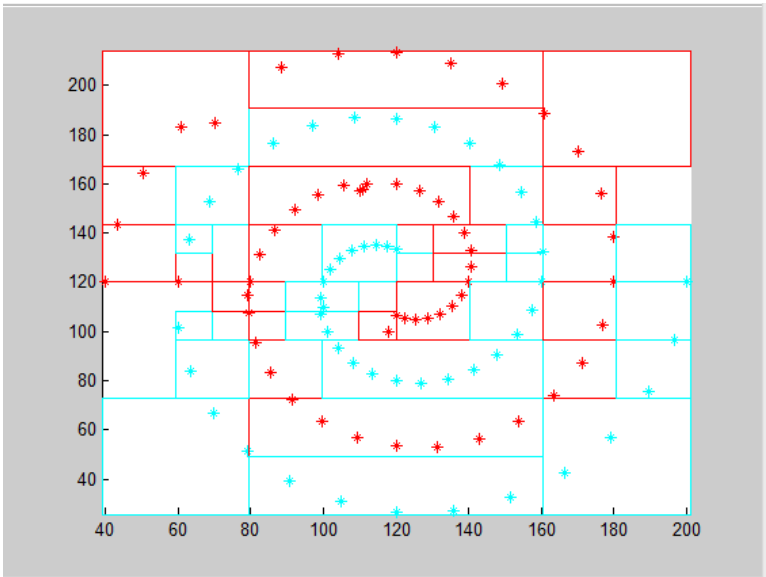

Figura 5.34: Conjunto de datos con los hiper–cubos generados del entrenamiento (4)

Una vez entrenada la RNMD y habiendo obtenido ya los pesos sinápticos de las dendritas, se probó la red con un conjunto de datos de prueba el cual tiene un total de 496 valores obteniendo un porcentaje del 94.6 % de clasificación acertada. En la figura 5.35 se observa como es que los datos han sido clasificados, los datos en color azul representan los valores que fueron asignados a una clase distinta a la que pertenecen.

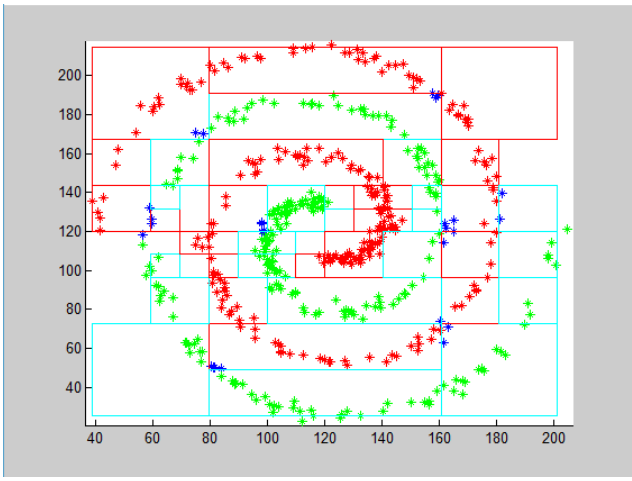

Figura 5.35: Clasificación del conjunto de datos de prueba (4)

Se presenta en la figura 5.36, los resultados obtenidos de la simulación de la implementación de la RNMD una vez dados la matriz de pesos y el conjunto de datos de prueba. En la matriz *MEMRAM_res*, se almacena en la primera columna el número de dato a probar y en la segunda la clase a la que fue clasificado. Estos datos son los gráficados en la imagen 5.35.

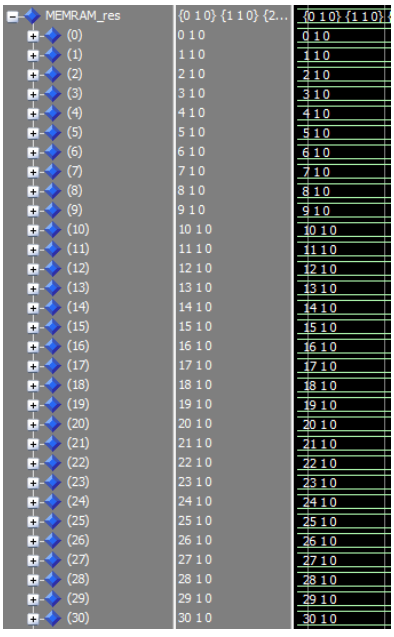

Figura 5.36: Simulación de la clasificación del conjunto de datos de prueba en FPGA (4)

Finalmente, se muestra en 5.37 el tiempo de entrenamiento obtenido en FPGA para este conjunto de datos, de igual manera se toma como referencia el reloj de 50Mhz con el que trabaja la tarjeta de desarrollo, el cual genera ciclos de reloj de 20ns. El tiempo de entrenamiento oscila entre los 0.186ms, lo cual es un tiempo mucho menor que el obtenido en MATLAB el cual es de alredor de 217ms.

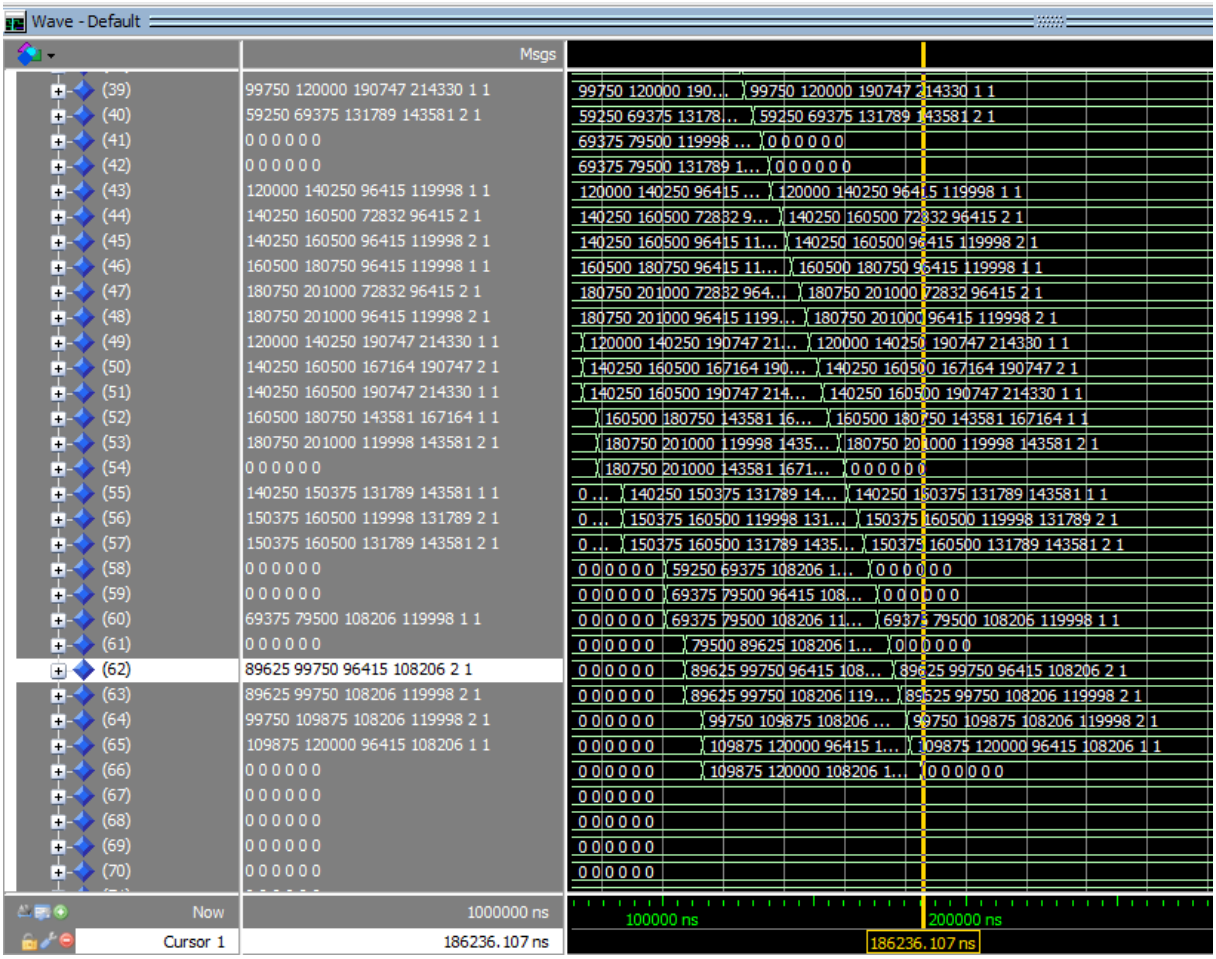

Figura 5.37: Tiempo de entrenamiento para la RNMD de p clases 2 dimensiones (ejemplo 4)

En las figuras 5.38, 5.39 se muestra el reporte de compilación para la implementación del entrenamiento y de la red neuronal morfológica con procesamiento en sus dendritas de 2 dimensiones y p clases, para el caso del cuarto conjunto de prueba, el cual define a dos espirales entrelazadas 5.31.

Como puede verificarse el uso de recursos utilizados es igual al 8 % del total de recursos del FPGA, por lo tanto se observa que sería posible implementar una RNMD con un número mayor de dimensiones y de datos de entrenamiento, dado el uso de recursos alcanzado.

Flow Summary	
Flow Status	Successful - Wed May 13 19:15:12 2015
Quartus II 64-Bit Version	14.1.0 Build 186 12/03/2014 SJ Web Edition
Revision Name	trainingRNMD2
Top-level Entity Name	trainingRNMD2
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	8,995 / 114,480 (8 %)
Total combinational functions	8,990 / 114,480 (8 %)
Dedicated logic registers	1,639 / 114,480 (1 %)
Total registers	1639
Total pins	27 / 529 (5 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

Figura 5.38: Reporte de compilación para entrenamiento de RNMD de 2 dimensiones

Flow Summary	
Flow Status	Successful - Tue Jun 02 03:25:15 2015
Quartus II 64-Bit Version	14.1.0 Build 186 12/03/2014 SJ Web Edition
Revision Name	pruebaRNMD2
Top-level Entity Name	pruebaRNMD2
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	9,170 / 114,480 (8 %)
Total combinational functions	9,170 / 114,480 (8 %)
Dedicated logic registers	0 / 114,480 (0 %)
Total registers	0
Total pins	69 / 529 (13 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

Figura 5.39: Reporte de compilación para implementación de RNMD de 2 dimensiones

Capítulo 6

Conclusión y trabajo a futuro

En este capítulo se presentan las conclusiones obtenidas a partir de la elaboración del presente trabajo de investigación. Así mismo, se menciona el trabajo a futuro que puede resultar de este tema de tesis.

6.1. Conclusiones

La conclusiones obtenidas a partir de los objetivos planteados fuerón las siguientes:

1. Se logró demostrar que es posible implementar en una arquitectura tipo FPGA una RNMD y su algoritmo de entrenamiento, compuesta de una sola neurona y K dendritas para el caso de p clases de 1 y 2 dimensiones.

2. Al verificar los resultados obtenidos en la FPGA con los obtenidos en la implementacion de MATLAB se comprobó su equivalencia por simulación.

3. Con los experimentos realizados se pudo verificar la obtención de resultados plenamente satisfactorios en el entrenamiento de los mismos, con una eficacia superior al 90 % en todos ellos y con tiempos que van entre 10 y 200 μs.

6.2. Trabajo a futuro

Esta implementación contiene mucho material sustancioso para continuar con una investigación detallada y bien fundamentada de cada aspecto relacionado, además cada etapa del proceso de diseño tiene aspectos interesantes que deben ser abordados detalladamente, y que desafortunadamente escapan al presente trabajo de tesis, tales como:

- Aunque el tamaño de las redes manejadas en el presente trabajo es más que suficiente para demostrar un correcto funcionamiento del algoritmo y una buena implementación, en el mundo real existen condiciones más exigentes y para realizar esas pruebas se necesita un mejor equipo con mayor capacidad para la compilación y síntesis de los diseños en la FPGA. Por lo tanto, si se cuentan con estos recursos se deberían realizar estas pruebas cuyos resultados influyen directamente en el diseño e implementación de la red.

- La implementación lograda en la presente tesis es eficiente y funciona correctamente, sin embargo, conforme va pasando el tiempo se mejoran las tecnologías y el diseño propuesto aquí no es la excepción, la implementación de un hardware reconfigurable podría mejorar el uso de recursos y el tiempo de compilación y síntesis. El diseño de una arquitectura robusta pensada especialmente para la implementación de RNMD con el manejo de memoria necesario, el manejo de matrices y de los ciclos seria ideal para una generalización del problema, esta implementación es ambiciosa y se deja como propuesta para algún buen trabajo de investigación.

- Si bien está limitado en cuanto a las dimensiones por la capacidad de procesamiento necesario para la compilación y síntesis, este desarrollo fue bien diseñado y tiene un correcto funcionamiento por lo que puede ser aplicado en diversos trabajos como un módulo clasificador reajustable *entrenable* en tiempo real. Las aplicaciones son sumamente interesantes, variadas y útiles, están van desde el aprendizaje de robots, dispositivos biométricos inteligentes, equipo médico, reconocimiento de patrones, análisis de imágenes.

6.3. Conclusión general

En este trabajo se logró la implementación exitosa de la Red Neuronal Morfológica con Procesamiento en sus Dendritas(RNMD) hasta para dos dimensiones, además del entrenamiento de la red dentro de la misma FPGA lo que supuso una serie de problemáticas que se lograrón solucionar exitosamente.

Este trabajo servirá no sólo de base para futuras implementaciones de este tipo de red en FPGA, sino que también es la primera en su tipo. Se demostró en los resultados que es un buen camino para seguir adelante, además de la tendencia mundial del uso de las FPGA como sustitutos del cómputo paralelo, debido a que en muchas áreas demuestra mejores resultados y un mayor rendimiento.

Aunque aún queda mucho por hacer en cuanto al desarrollo de las RNMD en dispositivos FPGA, la presente implementación podría servir de base no sólo en este campo, también puede ser aplicado directamente a otros objetivos, como por ejemplo el aprendizaje de robots, ya que al tener el entrenamiento y la red en la misma implementación se puede tener una retroalimentación o recalibración continua de la información que se obtenga del ambiente, y así usarlo directamente como entrada del entrenamiento para que en tiempo real reajuste sus pesos y consiga la clasificación correcta de los objetos, tareas o situaciones, que es para lo que está destinada esa red neuronal.

Finalmente, las pruebas antes mencionadas escapan al presente trabajo, pero el diseño e implementación propuestos aquí están listos para ser probados en este tipo de implementaciones. Los resultados obtenidos demuestran que la implementación de la red es eficiente y han sido los mismos que los obtenidos en las implementaciones en un entorno como MATLAB.

Bibliografía

[1] Arias, E. and Torres, *Sistemas Inteligentes en un chip utilizando FPGA: Aplicaciones a la visión por computadora*, X Congreso Internacional de Electrónica, Comunicaciones y Computadoras, pp 37–41, Cholula, México, Febrero, 2000.

[2] Baratta, Bo, Caviglia, Diotalevi and Valle, *Microelectronic Implementacion of ANN*, 5th Electronic Devices and Systems Intl Conf., Rep. Checa, 1998.

[3] Pérez-Uribe, A.,*FAST: A Neural Network with Flexible Adaptable-Size Topology*, `http://lslwww.epfl.ch/pages/tutorials/`, 1999.

[4] Lenne, P.,*Digital Connectionist Hardware: Current Problems and Future Challenges*, International Wor–Conference on Atificial and Natural Neural Networks, pp 688–713, Lanzarote, España, 1997.

[5] M. Krips, T. Lammert, A. Kummert, *FPGA implementation of a neural network for a real-time hand tracking system*, In proceedings of First IEEE International Workshop on Electronic Design Test and Applications, pp 313–317, 2002.

[6] F. Yang, M. Paindavoine, *Implementation of an RBFneural network on embedded systems: real..time face tracking and identity verification*, IEEE Transaction on Neural Networks, pp 1162–1175, 2003.

[7] Y, Maeda, T. Tada, *FPGA Implementation of a pulse density neural network with learning ability using simultaneous pertubation*, IEEE Transaction on Neural Networks 14, pp 688–695, 2003.

[8] R. Gadea, J. Cerda, F. Ballester, A. Macholi, *Artificial Neural Network implementation on a single FPGA of a pipelined on–line back–propagation*, In proceeings of the 13th International Symposium on System Synthesis, pp 225–230, 2008.

[9] Xilinx, *Virtex–ii pro and Virtex–ii pro x platform fpgas: complete data sheet*, `http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf`, 2007.

[10] S. Himavathi, D. Anitha, A. Muthuramalingam, *Feedforward neural network implementation in FPGA using layer multiplexing for effective resorce utilization*, pp 880–888, 2007.

[11] K. Rice, T. Taha, C. Vutsinas, *Scaling analysis of a neocortex inspired congnitive model on the Cray XD1*, The Journal of Supercomputing, pp 21–43, 2009.

[12] D. George, J. Hawkins, *A hierarchical bayesian model of invariant pattern recognition in the visual cortex*, Proceedings of the IEEE International Joint Conference on Neural Networks, vol. 3, pp 1812–1817, 2005.

[13] S. Johnston, G. Prasad, L.P. Maguire, T. T. McGinnity, *Comparative investigation into classical and spiking neuron implementations on FPGAs*, in ICANN, pp 269–274, 2005.

[14] B.R. Gaines, *Stochastic computing systems*, Advances in Information Systems Science, pp 37–162, 1969.

[15] M.V. Daalen, P. Jeavons, J. Shawe–Taylor, *A stochastic neural architecture that exploits dynamically reconfigurable FPGAs*, IEEE Workshop on FPGAs for Custom Computing Machines, pp 202–211, Loas Alamitos CA, 1994.

[16] N. Nedjah, L. de Macedo Mourelle, *Reconfigurable hardware for neural networks: binary versus stochastic*, Neural Computing and Applications, pp 249–255, 2007.

[17] T. Szabo, L. Antoni, G. Horvath, B. Feher, *A full–parallel digital implementation for pre–trained NNs*, IEEE–INNS–ENNS International Joint Conference on Neural Networks, vol.2, pp 2049, 2000.

[18] G. Palm, F. Kurfess, F. Schwenker, *Neural associative memories*, Associative Processing and Processors, pp 284–306, 2008.

[19] U. Ruckert, *An associative memory with neural architecture and its VLSI implementation*, Proceedings of Hawaii International Conference on System Sciences, pp 212–218, 2001.

[20] A. Heittmann, U. Ruckert, *Mixed mode VLSI implementation of a neural associative memory*, Analog Integrated Circuits and Signal Processing, pp 159–172, 2002.

[21] D.J. Willshaw, O.P. Buneman, H.C. Longuet–Higgins, *Non–holographic associative memory*, Nature, pp 960–962, 1969.

[22] M. Weeks, M. Freeman, A. Moulds, J. Austin, *Developing hardware–based applicatons using PRESENCE-2*, Perspectives in Pervasice Computing Savoy Place, pp 469–474, London, 2005.

[23] W. Bledsoe, I. Browning, *Pattern recognition and reading by machine*, Proceedings of Eastern Joint Computer Conference, vol.2, pp 225–232, Boston, 1959.

[24] I. Aleksander, W.V. Thomas, P.A. Bowden, *WISARD: a radical step forward in image recognition*, Sensor Review, pp 120–124, 1984.

[25] T.G. Clarkson, C.K. Ng, D. Gorse, J.G. Taylor, *Learning probabilistic RAM nets using VLSI structures*, IEEE Transactions on Computers, pp 1552–1561, 2002.

[26] G.X. Ritter, Gonzalo Urcid, *Lattice Algebra Approach to Single Neuron Computation*, IEEE Transactions on Neural Networks, pp 282–295, 2008.

[27] G.X. Ritter, I. Iancu, G. Urcid, *Morphological perceptrons with dendritic structure*, 12th IEEE International Conference in Fuzzy Systems, vol. 2, pp 1296–1301, 2003.

[28] G.X. Ritter, Gonzalo Urcid, *Learning in lattice neural networks that employ dendritic computing*, Comput. Intell. Based on Lattice Theory, pp 25–44, 2007.

[29] H. Sossa, E. Guevara, *Efficient training for dendrite morphological neural networks*, Neurocomputing, vol. 131, pp 132–142, 2014.

[30] J. Parra, D. Ramos, A. Tigreros *Implementación de redes neuronales utilizando dispositivos lógicos programables*, Visión de Caso, pp 48–55, 2011.

[31] L. Navarría, J. Rapallini, A. Quijano *Desarrollo de redes neuronales en FPGA*, Codiseño Hardware/Software, 2011.

[32] H. Sossa, E. Guevara, *Modified Dendrite Morphological Neural Network Applied to 3D Object Recognition on RGB-D Data*, Hybrid Artificial Intelligent Systems: Lecture Notes in Computer Science, Volume 8073, pp 304–313, 2013.

[33] J. Davidson, F. Hummen, *Morphology neural networks: An introduction with applications*, Circuits, Systems and Signal Processing, Volume 12, pp 177–210, 1993.

[34] J. Misra, I. Saha, *Artificial neural networks in hardware: A survey of two decades of progress*, Neurocomputing, Volume 74, pp 239–255, 2010.

Apéndice A

Características del FPGA Altera DE2–115

En este apartado se describen las especificaciones de FPGA modeo Cyclone II serie 2C35.

- Cyclone II 2C35:
  - Contiene 32,216 elementos lógicos.
  - 475 entradas/salidas para el usuario.
  - 105 bloques de RAM M4K y 483 Kbits de RAM.
  - 35 multiplicadores embebidos y 4 PLLs.
- Configuración serial (EPCS16) y circuito USB Blaster:
  - USB Blaster embebido para programación y controlador API para el usuario.
  - Modo JTAG y AS soportados.
  - Configuración serial de EPCS16
- 8Mbyte SDRAM:
  - Organizada en 1M x 4 x 16.
  - Soporta el acceso de NIOS II y el controlador Terasic multi–puerto de alta velocidad.
- 1 Mbyte de memoria flash (ampliable hasta 4):
  - Equipado con 4 Mbyte de memoria flash NAND.
  - 8 bits de bus de datos.

- Entrada para tarjeta SD:
  - Acceso a tarjetas en modo SPI.
  - Soporta acceso de NIOS II con el driver de la SD de Terasic.
- Botones *pushbutton*:
  - 4 *push botton*
  - Normalmente en alta y genera una señal de baja cuando se presionan.
- DPDT *Switches*:
  - Contiene 18 *switches* para que el usuario pueda interactuar con el dispositivo FPGA.
  - Cuando se encuentra en posición *down* causa un 0 lógico y cuando esta en posición *up* causa un 1 lógico.
- Relojes:
  - 1 oscilador de 50 MHz.
  - 1 reloj de 27 MHz.
  - Contiene una conexión SMA para poder conectar un reloj externo.
- Codificador de audio:
  - Utiliza Wolfson WM 8731 de 24 bits.
  - Tiene línea de entrada, línea de salida y micrófono.
  - Frecuencia de operación: 8 KHz a 96KHz.
  - Aplicaciones para reproductores MP3, PDA y teléfonos inteligentes.
- Salida VGA:
  - Utiliza el ADV7123 a 240 MHz triple a 10 bits video DAC de alta velocidad.
  - Con conector de 15 pines D–sub.
  - Soporta hasta 1600 x 1200 a 100 MHz.
  - Puede ser empleado para implementar un codificador de TV.
- Circuito decodificador de NTSC/PAL:
  - Usa ADI 7181B multiformato decodificador de video SDTV.
  - Soporta NTSC-M,J,4.43, PAL-D,B,G,H,I,M,N, SECAM.

  - Integra tres 54MHz ADCs de 9 bits.
  - Entrada de oscilador de 27 MHz.
  - Múltiples formatos de entrada analógica: Video compuesto (CVBS), SVideo(Y/C) y componentes YPrPb.
  - Aplicaciones en: grabadores de DVD, televisores LCD, televisión digital, dispositivos portátiles.

- Controlador 10/100 ethernet:
  - MAC y PHY interfaz integrada.
  - Soporta aplicaciones 10Base–T y 100Base–T.
  - Operaciones full dúplex a 10 y 100 Mb/s con auto MDIX.
  - Compatible con las especificaciones 802.3u de IEEE.
  - Soporta generación de checksum IP/TCP/UDP.

- Controlador USB Esclavo/Maestro:
  - Cumple completamente las especificaciones del Universal Serial Bus Rev. 2.0.
  - Soporta transferencia de datos en alta y en baja velocidad.
  - Soporta ambos tipos USB maestro y esclavo.
  - Soporta dos puertos USB (un tipo A para maestro y un tipo B para dispositivos conectados a la DE2).
  - Provee una interfaz paralela de alta velocidad para gran mayoría de los CPUs.
  - Soporta programación de entrada/salida (PIO) o Acceso directo a Memoria (DMA).

- Puerto serial:
  - Provee de dos puertos seriales: un RS–232 y otro PS/2.
  - Tiene un conector serial tipo DB–9 para el puerto RS–232.
  - Tiene conector PS/2 para conectar un ratón o un teclado a la tarjeta DE2.

- Transceptor IrDA:
  - Contiene un transceptor de 115.2 Kb/s.
  - LED de 32 mA.
  - EMI blindado integrado.

- Detector de entradas de filo.

- Dos cabezas de expansión de 40 pines con diodos de protección:
  - Un total de 72 Cyclone II entradas/salidas en extensión de dos conectores de 40 pines.
  - La cabeza de 40 pines fue diseñada para aceptar el cable estándar de 40 pines usado para el estándar de dispositivos IDE.

- Configuración del dispositivo y circuito USB Blaster:
  - Circuito USB Blaster embebido en la tarjeta.
  - Provee los modos de programación JTAG y AS.
  - Contiene 16 Mbit (EPCS16) para configuración del dispositivo.

Apéndice B

Código Implementado en FPGA

En este apartado se muestra el código implementado para el algoritmo de entrenamiento y prueba de la red neuronal morfológica con procesamiento en sus dendritas, de una y dos dimensiones.

B.1. Código para el entrenamiento de la RNMD de 1 dimensión y p clases

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

entity trainingRNMD2 is
generic(lI:integer:=10;Ndim:integer:=1);
    Port (
                                clk, clr: in std_logic;
                                indica: out std_logic_vector(23 downto 0);
                                RW: in std_logic
                            );
end  trainingRNMD2;

architecture Behavioral of trainingRNMD2 is
signal j_out, i_out : integer := 0;

TYPE ARREGLO4 IS ARRAY (0 TO (lI+1)*Ndim-1, 0 to 3) OF integer;
SIGNAL MEMRAM_PRS: ARREGLO4;
--Conjunto de datos de entrenamiento
TYPE ARREGLO IS ARRAY (lI downto 0, 0 to 1) OF integer;
CONSTANT MEMROM_I: ARREGLO := (
        (100,2),
        (700,2),
        (400,1),
        (1200,1),
        (900,2),
        (1500,2),
        (1600,3),
        (2000,1),
        (2500,1),
        (2900,3),
        (3000,1),
        OTHERS => (0,0)
);

function pertenece (a,b,i:integer;X:ARREGLO; c:integer) return integer is
variable per:integer:=0;
```

```
begin
        if(X(i,c)>= a and X(i,c)<=b)then
                        per := 1;
        else
                        per := 0;
        end if;
        return (per);
end pertenece;

begin

-- COMENZAR CON EL ENTRENAMIENTO DE LA RNMPD
-- BLOQUE WHILES ANIDADOS
bloque_i:process(clk, RW, clr)
variable i, j, K,clase, aux,auxi,auxj,check,per,nuevo:integer:=0;
variable max:integer:=0;
variable min:integer:=100;
variable MEMRAM_PRV: ARREGLO4;
variable first,inc:integer:=1;
begin

if(clr='1')then
                for l in 0 to lI loop
                        MEMRAM_PRV(l,0) := 0;
                        MEMRAM_PRV(l,1) := 0;
                        MEMRAM_PRV(l,2) := 0;
                        MEMRAM_PRV(l,3) := 0;
                end loop;
                indica <= (OTHERS =>'0');
elsif(clk' event and clk ='1')then

        if (first = 1)then

                -- Obetener el maximo y el minimo
                for m in 0 to lI loop
                        if(max < MEMROM_I(m,0))then
                                max := MEMROM_I(m,0);
                        end if;
                        if(min > MEMROM_I(m,0))then
                                min := MEMROM_I(m,0);
                        end if;
                end loop;

                --Inicializar todo PR
                for n in 0 to (lI+1)*Ndim-1 loop
                        MEMRAM_PRS(n,0) <= 0;
                        MEMRAM_PRS(n,1) <= 0;
                        MEMRAM_PRS(n,2) <= 0;
                        MEMRAM_PRS(n,3) <= 0;

                        MEMRAM_PRV(n,0) := 0;
                        MEMRAM_PRV(n,1) := 0;
                        MEMRAM_PRV(n,2) := 0;
                        MEMRAM_PRV(n,3) := 0;
                end loop;

                --Crear la primera particionde PR
                if(RW='1')then
                        MEMRAM_PRV(0,0) := min;
                        MEMRAM_PRV(0,1) := max;
                        MEMRAM_PRV(0,2) := 0;
                        MEMRAM_PRV(0,3) := 0;
                end if;
        end if;

                first := 0;
                if(i<(lI+1)*Ndim)then
                        if(j=lI+1)then
                        --While afuera
                                        if (check=1) then
                                                if(clase=0) then
                                                        MEMRAM_PRV(auxi,0) := 0;
                                                        MEMRAM_PRV(auxi,1) := 0;
                                                end if;
                                        end if;

                                        check:=0;
                                        clase := 0;
```

```
                                        j := 0;
                                        i := i+1;

                                        if(i<(lI+1)*Ndim)then
                                                if(MEMRAM_PRV(i,3)=0 and (MEMRAM_PRV(i,0) /= MEMRAM_PRV(i,1)))then
                                                        check:= 1;
                                                else
                                                        check:= 0;
                                                        j:=lI+1;
                                                end if;

                                        end if;
                                end if;
                        end if;


                        if(j<=lI and i<(lI+1)*Ndim)then
                                --While adentro
                                per := pertenece(MEMRAM_PRV(i,0),MEMRAM_PRV(i,1),j,MEMROM_I,0);
                                if(per=1)then
                                        if(clase=0)then
                                                clase := MEMROM_I(j,1);
                                                MEMRAM_PRV(i,2):= clase;
                                                MEMRAM_PRV(i,3):= 1;
                                        else
                                                if(clase/=MEMROM_I(j,1))then

                                                        --modificamos la particion actual
                                                        aux := MEMRAM_PRV(i,1);
                                                        MEMRAM_PRV(i,1) := (MEMRAM_PRV(i,0) + aux)/2;
                                                        MEMRAM_PRV(i,2) := 0;
                                                        MEMRAM_PRV(i,3) := 0;

                                                        --Ocupamos los primeros espacios disponibles
                                                        for k in 0 to (lI+1)*Ndim-1 loop
                                                                if(MEMRAM_PRV(k,0) =MEMRAM_PRV(k,1))then
                                                                        MEMRAM_PRV(k,0) := MEMRAM_PRV(i,1);
                                                                        MEMRAM_PRV(k,1) := aux;
                                                                        MEMRAM_PRV(k,2) := 0;
                                                                        MEMRAM_PRV(k,3) := 0;
                                                                        exit;
                                                                end if;
                                                        end loop;

                                                        auxi:=i;
                                                        i := -1;
                                                        j:=lI; -- break;
                                                end if;
                                        end if;
                                end if;
                                j := j+1;
                        end if;
                        --w int end
                end if; -- end clk

                -- Asignar senales de salida
                i_out <= min;
                j_out <= max;

                for l in 0 to lI loop
                                MEMRAM_PRS(l,0) <= MEMRAM_PRV(l,0);
                                MEMRAM_PRS(l,1) <= MEMRAM_PRV(l,1);
                                MEMRAM_PRS(l,2) <= MEMRAM_PRV(l,2);
                                MEMRAM_PRS(l,3) <= MEMRAM_PRV(l,3);
                end loop;

                indica <= CONV_STD_LOGIC_VECTOR((MEMRAM_PRV(0,3)), 24);

end process;
end Behavioral;
```

B.2. Código para la implementación de la RNMD de 1 dimensión y p clases

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


entity pruebaRNMD is
generic(PI:integer:=11; PJ:integer:=2); -- Numero de dendritas
    Port (
                                x_prueba: in std_logic_vector(15 downto 0);
                            clase: out STD_LOGIC_VECTOR(7 downto 0);
                                dis_0: out STD_LOGIC_VECTOR(6 downto 0);
                                dis_1: out STD_LOGIC_VECTOR(6 downto 0)
                            );
end pruebaRNMD;

architecture Behavioral of pruebaRNMD is



TYPE ARREGLO2 IS ARRAY (0 TO PI-1, 0 to PJ) OF integer;
SIGNAL MEMRAM_res: ARREGLO2;
SIGNAL MEMRAM_res_max: ARREGLO2;


TYPE ARREGLO IS ARRAY (0 TO PI-1, 0 to PJ) OF integer;
CONSTANT MEMROM_MATRIZ_P: ARREGLO := (
        (-100,-281,2),
        (-1550,-1912,3),
        (-825,-1187,2),
        (-462,-825,2),
        (-281,-462,1),
        (-2275,-2637,1),
        (-1912,-2275,1),
        (-1187,-1368,1),
        (-2909,-3000,1),
        (-1368,-1550,2),
        (-2818,-2909,3),

        OTHERS => (0,0)
);

signal indica:std_logic;
signal aux:std_logic_vector(7 downto 0);
begin


gen_particiones:for i in 0 to PI-1 generate
        gen1:process(x_prueba)
        variable min_i: integer;

        begin
        for j in 0 to PJ-1 loop
                if( j = 0)then
                        min_i := ((-1)**(j+1))*(-1*(MEMROM_MATRIZ_P(i,j)+CONV_INTEGER(x_prueba)));
                else
                -- encontrar el minimo
                        if(min_i < (((-1)**(j+1)) * (-1*(MEMROM_MATRIZ_P(i,j)+CONV_INTEGER(x_prueba))))then
                                min_i := min_i;
                        else
                                min_i := ((-1)**(j+1)) * (-1*(MEMROM_MATRIZ_P(i,j)+CONV_INTEGER(x_prueba)));
                        end if;
                end if;
        end loop;
        MEMRAM_res(i,0) <= i;
        MEMRAM_res(i,1) <= min_i;
        MEMRAM_res(i,2) <= MEMROM_MATRIZ_P(i,2);
   end process;
 end generate;
```

```
    EncontarMaximo:process(MEMRAM_res, aux)
        variable max_j: integer:= MEMRAM_res(0,1);
        variable max_j_aux: integer;
        variable max_j_aux2: integer;
        begin
                for i in 0 to PI-1 loop
                                max_j_aux := MEMRAM_res(i,1);

                                if(max_j < max_j_aux ) then
                                                max_j := max_j_aux;
                                                max_j_aux2 := MEMRAM_res(i,2);
                                else
                                                max_j := 0;
                                                max_j_aux2 := 0;
                                end if;
                                MEMRAM_res_max(i,0) <= i;
                                MEMRAM_res_max(i,1) <= max_j;
                                MEMRAM_res_max(i,2) <= max_j_aux2;

                                if(max_j_aux2/=0)then
                                        clase <= CONV_STD_LOGIC_VECTOR(max_j_aux2,8);
                                        aux <= CONV_STD_LOGIC_VECTOR(max_j_aux2,8);
                                end if;

                end loop;


        end process;

        decoSal: process(aux)
        begin
                case(aux) is
                        when "00000000" => dis_0 <= "1000000"; --0
                        when "00000001" => dis_0 <= "1111001"; --1
                        when "00000010" => dis_0 <= "0100100"; --2
                        when "00000011" => dis_0 <= "0110000"; --3
                        when "00000100" => dis_0 <= "0011001"; --4
                        when "00000101" => dis_0 <= "0010010"; --5
                        when "00000110" => dis_0 <= "0000010"; --6
                        when "00000111" => dis_0 <= "0111000"; --7
                        when "00001000" => dis_0 <= "0000000"; --8
                        when "00001001" => dis_0 <= "0011000"; --9
                        when others => dis_0 <= "1111111";
                end case;

                dis_1 <= "1000110";

        end process;


end Behavioral;
```

B.3. Código para el entrenamiento de la RNMD de 2 dimensiones y p clases

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


entity trainingRNMD2 is
generic(lI:integer:=104;Ndim:integer:=2);
    Port (
                                clk, clr: in std_logic;
                                indica: out std_logic;
                                RW: in std_logic
                                --MEMRAM_PRS_OUT: out array_UI
                        );
end  trainingRNMD2;

architecture Behavioral of trainingRNMD2 is
signal j_out, i_out : integer := 0;
```

```
TYPE ARREGLO4 IS ARRAY (0 TO (lI+1)*Ndim-1, 0 to 5) OF integer;
SIGNAL MEMRAM_PRS: ARREGLO4;

TYPE ARREGLO5 IS ARRAY (0 TO Ndim-1, 0 to 3) OF integer;

TYPE ARREGLO IS ARRAY (0 to lI , 0 to 2) OF integer;
CONSTANT MEMROM_I: ARREGLO := (
(40000, 120000, 1),
(43399, 143310, 1),
(50497, 164630, 1),
(60804, 182940, 1),
(70000, 185000, 1),
(79610, 107710, 1),
(79000, 115000, 1),
(80000, 120000, 1),
(81387, 95207,  1),
(82396, 131440, 1),
(85376, 83184,  1),
(86535, 141490, 1),
(88388, 207520, 1),
(91498, 72346,  1),
(92077, 149690, 1),
(98623, 155740, 1),
(99545, 63368,  1),
(104110,        212850, 1),
(105740,        159470, 1),
(109190,        56862,  1),
(112000,        160000, 1),
(111000,        158000, 1),
(110000,        157000, 1),
(120000,        53333,  1),
(118000,        100000, 1),
(120000,        106670, 1),
(120000,        160000, 1),
(120000,        213330, 1),
(122540,        105140, 1),
(125580,        104550, 1),
(126360,        157140, 1),
(128910,        105110, 1),
(131440,        53148,  1),
(131780,        152610, 1),
(132290,        106940, 1),
(135260,        209140, 1),
(135450,        110080, 1),
(136030,        146810, 1),
(138110,        114490, 1),
(138990,        140190, 1),
(140000,        120000, 1),
(140590,        133220, 1),
(140890,        126360, 1),
(142930,        56503,  1),
(149130,        200660, 1),
(153850,        63410,  1),
(160970,        188500, 1),
(163560,        73684,  1),
(170260,        173440, 1),
(171480,        86942,  1),
(176630,        156360, 1),
(177100,        102620, 1),
(179890,        138230, 1),
(180000,        120000, 1),
(60000, 120000, 1),
(60112, 101770, 2),
(62897, 137380, 2),
(63368, 83636,  2),
(68516, 153060, 2),
(69739, 66558,  2),
(76441, 166320, 2),
(79028, 51497,  2),
(86153, 176590, 2),
(90868, 39342,  2),
(97066, 183500, 2),
(99109, 113640, 2),
(99406, 106780, 2),
(100000,        110000, 2),
(100000,        120000, 2),
(101010,        99811,  2),
```

```
(101890,        125510, 2),
(103970,        93194,  2),
(104550,        129920, 2),
(104740,        30864,  2),
(107710,        133060, 2),
(108220,        87394,  2),
(108560,        186850, 2),
(111090,        134890, 2),
(113640,        82860,  2),
(114420,        135450, 2),
(117460,        134860, 2),
(120000,        26667,  2),
(120000,        80000,  2),
(120000,        133330, 2),
(120000,        186670, 2),
(126990,        79146,  2),
(130810,        183140, 2),
(134260,        80529,  2),
(135890,        27150,  2),
(140450,        176630, 2),
(141380,        84259,  2),
(147920,        90310,  2),
(148500,        167650, 2),
(151610,        32477,  2),
(153460,        98512,  2),
(154620,        156820, 2),
(157600,        108560, 2),
(158610,        144790, 2),
(160000,        120000, 2),
(160390,        132290, 2),
(166320,        42561,  2),
(179200,        57057,  2),
(189500,        75372,  2),
(196600,        96688,  2),
(200000,        120000, 2),
        OTHERS => (0,0,0)
);

function pertenece (a,b,i:integer;X:ARREGLO; c:integer) return integer is
variable per:integer:=0;
begin
        if(X(i,c)>= a and X(i,c)<=b)then
                        per := 1;
        else
                        per := 0;
        end if;
        return (per);
end pertenece;

begin
-- COMENZAR CON EL ENTRENAMIENTO DE LA RNMPD
-- BLOQUE WHILES ANIDADOS
bloque_i:process(clk, RW, clr)
variable i, j, K,clase, aux,auxi,auxj,check,per,nuevo:integer:=0;
variable max_x:integer:=0;
variable min_x:integer:=200000;
variable max_y:integer:=0;
variable min_y:integer:=200000;
variable MEMRAM_PRV: ARREGLO4;
variable MEMRAM_PAUX: ARREGLO5;
variable first,inc:integer:=1;
variable perCount, m:integer:=0;

begin
if(clr='1')then
                for l in 0 to lI loop
                        MEMRAM_PRV(l,0) := 0;
                        MEMRAM_PRV(l,1) := 0;
                        MEMRAM_PRV(l,2) := 0;
                        MEMRAM_PRV(l,3) := 0;
                        MEMRAM_PRV(l,4) := 0;
                        MEMRAM_PRV(l,5) := 0;

                end loop;

                for l in 0 to Ndim-1 loop
                        MEMRAM_PAUX(l,0) := 0;
                        MEMRAM_PAUX(l,1) := 0;
```

```
                        MEMRAM_PAUX(l,2) := 0;
                        MEMRAM_PAUX(l,3) := 0;
                end loop;

                indica <= '0';

elsif(clk' event and clk ='1')then

        if (first = 1)then

                -- Obetener el maximo y el minimo
                for m in 0 to lI loop
                        if(max_x < MEMROM_I(m,0))then
                                max_x := MEMROM_I(m,0);
                        end if;
                        if(min_x > MEMROM_I(m,0))then
                                min_x := MEMROM_I(m,0);
                        end if;
                end loop;

                for m in 0 to lI loop
                        if(max_y < MEMROM_I(m,1))then
                                max_y := MEMROM_I(m,1);
                        end if;
                        if(min_y > MEMROM_I(m,1))then
                                min_y := MEMROM_I(m,1);
                        end if;
                end loop;

                --Inicializar todo PR
                for n in 0 to (lI+1)*Ndim-1 loop
                        MEMRAM_PRS(n,0) <= 0;
                        MEMRAM_PRS(n,1) <= 0;
                        MEMRAM_PRS(n,2) <= 0;
                        MEMRAM_PRS(n,3) <= 0;
                        MEMRAM_PRS(n,4) <= 0;
                        MEMRAM_PRS(n,5) <= 0;

                        MEMRAM_PRV(n,0) := 0;
                        MEMRAM_PRV(n,1) := 0;
                        MEMRAM_PRV(n,2) := 0;
                        MEMRAM_PRV(n,3) := 0;
                        MEMRAM_PRV(n,4) := 0;
                        MEMRAM_PRV(n,5) := 0;
                end loop;

                for l in 0 to Ndim-1 loop
                        MEMRAM_PAUX(l,0) := 0;
                        MEMRAM_PAUX(l,1) := 0;
                        MEMRAM_PAUX(l,2) := 0;
                        MEMRAM_PAUX(l,3) := 0;
                end loop;

                --Crear la primera particionde PR
                if(RW='1')then
                        MEMRAM_PRV(0,0) := min_x-1000;
                        MEMRAM_PRV(0,1) := max_x+1000;
                        MEMRAM_PRV(0,2) := min_y-1000;
                        MEMRAM_PRV(0,3) := max_y+1000;
                        MEMRAM_PRV(0,4) := 0;
                        MEMRAM_PRV(0,5) := 0;
                end if;
        end if;

                first := 0;
                if(i<(lI+1)*Ndim)then
                        if(j=lI+1)then
                        --While afuera
                                        if (check=1) then
                                                if(clase=0) then
                                                        MEMRAM_PRV(auxi,0) := 0;
                                                        MEMRAM_PRV(auxi,1) := 0;
                                                        MEMRAM_PRV(auxi,2) := 0;
                                                        MEMRAM_PRV(auxi,3) := 0;
                                                end if;
                                        end if;

                                        check:=0;
```

```
                                clase := 0;
                                j := 0;
                                i := i+1;
                                auxi:=i;
                                if(i<(lI+1)*Ndim)then
                                        if(MEMRAM_PRV(i,5)=0 and (MEMRAM_PRV(i,0) /= MEMRAM_PRV(i,1)))then
                                                check:= 1;
                                        else
                                                check:= 0;
                                                j:=lI+1;
                                        end if;

                                end if;
                        end if;
end if;


if(j<=lI and i<(lI+1)*Ndim)then
        --While adentro
        perCount:=0;
        for k in 0 to NDim-1 loop
                m := (k*2);
                per := pertenece(MEMRAM_PRV(i,m),MEMRAM_PRV(i,m+1),j,MEMROM_I,k);
                if(per=1)then
                        perCount:= perCount +1;
                end if;
        end loop;

        if(perCount=NDim)then
                if(clase=0)then
                        clase := MEMROM_I(j,Ndim);
                        MEMRAM_PRV(i,4):= clase;
                        MEMRAM_PRV(i,5):= 1;
                else
                        if(clase/=MEMROM_I(j,Ndim))then

                                --divide el renglon PR de acuerdo al numero de clases
                                for k in 0 to NDim-1 loop
                                        m:= (k*2);
                                        MEMRAM_PAUX(k,0) :=  MEMRAM_PRV(i,m);
                                        MEMRAM_PAUX(k,1) := (MEMRAM_PRV(i,m) + MEMRAM_PRV(i,m+1))/2;
                                        MEMRAM_PAUX(k,2) := (MEMRAM_PRV(i,m) + MEMRAM_PRV(i,m+1))/2;
                                        MEMRAM_PAUX(k,3) :=      MEMRAM_PRV(i,m+1);
                                end loop;

                                --modificamos la particion actual
                                MEMRAM_PRV(i,0) := MEMRAM_PAUX(0,0);
                                MEMRAM_PRV(i,1) := MEMRAM_PAUX(0,1);
                                MEMRAM_PRV(i,2) := MEMRAM_PAUX(1,0);
                                MEMRAM_PRV(i,3) := MEMRAM_PAUX(1,1);
                                MEMRAM_PRV(i,4) := 0;
                                MEMRAM_PRV(i,5) := 0;

                                --Ocupamos los primeros espacios disponibles
                                for k in 0 to (lI+1)*Ndim-1 loop
                                        if(MEMRAM_PRV(k,0) =MEMRAM_PRV(k,1))then
                                                MEMRAM_PRV(k,0) := MEMRAM_PAUX(0,0);
                                                MEMRAM_PRV(k,1) := MEMRAM_PAUX(0,1);
                                                MEMRAM_PRV(k,2) := MEMRAM_PAUX(1,2);
                                                MEMRAM_PRV(k,3) := MEMRAM_PAUX(1,3);
                                                MEMRAM_PRV(k,4) := 0;
                                                MEMRAM_PRV(k,5) := 0;
                                                exit;
                                        end if;
                                end loop;

                                for k in 0 to (lI+1)*Ndim-1 loop
                                        if(MEMRAM_PRV(k,0) =MEMRAM_PRV(k,1))then
                                                MEMRAM_PRV(k,0) := MEMRAM_PAUX(0,2);
                                                MEMRAM_PRV(k,1) := MEMRAM_PAUX(0,3);
                                                MEMRAM_PRV(k,2) := MEMRAM_PAUX(1,0);
                                                MEMRAM_PRV(k,3) := MEMRAM_PAUX(1,1);
                                                MEMRAM_PRV(k,4) := 0;
                                                MEMRAM_PRV(k,5) := 0;
                                                exit;
                                        end if;
                                end loop;
```

```
                                                for k in 0 to (lI+1)*Ndim-1 loop
                                                        if(MEMRAM_PRV(k,0) =MEMRAM_PRV(k,1))then
                                                                MEMRAM_PRV(k,0) := MEMRAM_PAUX(0,2);
                                                                MEMRAM_PRV(k,1) := MEMRAM_PAUX(0,3);
                                                                MEMRAM_PRV(k,2) := MEMRAM_PAUX(1,2);
                                                                MEMRAM_PRV(k,3) := MEMRAM_PAUX(1,3);
                                                                MEMRAM_PRV(k,4) := 0;
                                                                MEMRAM_PRV(k,5) := 0;
                                                                exit;
                                                        end if;
                                                end loop;


                                                auxi:=i;
                                                i := -1;
                                                j:=lI; -- break;
                                        end if;
                                end if;
                        end if;
                        j := j+1;
                end if;
                --w int end
        end if; -- end clk

        -- Asignar senales de salida
        i_out <= min_x;
        j_out <= max_x;

        for l in 0 to (lI+1)*Ndim-1 loop
                        MEMRAM_PRS(l,0) <= MEMRAM_PRV(l,0);
                        MEMRAM_PRS(l,1) <= MEMRAM_PRV(l,1);
                        MEMRAM_PRS(l,2) <= MEMRAM_PRV(l,2);
                        MEMRAM_PRS(l,3) <= MEMRAM_PRV(l,3);
                        MEMRAM_PRS(l,4) <= MEMRAM_PRV(l,4);
                        MEMRAM_PRS(l,5) <= MEMRAM_PRV(l,5);
        end loop;

        indica <= '1';

end process;
end Behavioral;
```

B.4. Código para la implementación de la RNMD de 2 dimensiones y p clases

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


entity pruebaRNMD2 is
generic(PI:integer:=54; PJ:integer:=4); -- Numero de dendritas, tamanio de la matriz P
    Port (
                                clk: in std_logic;
                                x_prueba: in std_logic_vector(7 downto 0);
                                y_prueba: in std_logic_vector(7 downto 0);
                                dis_0: out STD_LOGIC_VECTOR(6 downto 0);
                                dis_1: out STD_LOGIC_VECTOR(6 downto 0);
                                dis_4: out STD_LOGIC_VECTOR(6 downto 0);
                                dis_5: out STD_LOGIC_VECTOR(6 downto 0);
                                dis_6: out STD_LOGIC_VECTOR(6 downto 0);
                                dis_7: out STD_LOGIC_VECTOR(6 downto 0);
                                clase: out std_logic_vector(7 downto 0);
                                RW,LD: in STD_LOGIC
                        );
end  pruebaRNMD2;

architecture Behavioral of pruebaRNMD2 is
```

```
TYPE ARREGLO3 IS ARRAY (0 to PJ-1) OF integer;--(los datos deben ser (x,x,y,y))
SIGNAL MEMROM_X_PRUEBA: ARREGLO3;

TYPE ARREGLO2 IS ARRAY (0 TO PI-1, 0 to PJ-2) OF integer;-- deben ser del mismo tamanio que el arrego de prueba
SIGNAL MEMRAM_res: ARREGLO2;
SIGNAL MEMRAM_res_max: ARREGLO2;
signal x_prueba2: integer;


TYPE ARREGLO IS ARRAY (0 TO PI-1, 0 to PJ) OF integer; --- la matriz debe ser negativa
CONSTANT MEMROM_MATRIZ_P: ARREGLO := (
(-122000,-153500,-13000,-51500,1),
(-122000,-137750,-90000,-109250,1),
(-185000,-200750,-13000,-32250,1),
(-185000,-192880,-90000,-99625,2),
(-122000,-153500,-51500,-90000,1),
(-153500,-169250,-13000,-32250,1),
(-153500,-169250,-51500,-70750,2),
(-122000,-137750,-128500,-147750,1),
(-153500,-161380,-90000,-99625,2),
(-153500,-169250,-128500,-147750,4),
(-122000,-137750,-109250,-128500,1),
(-185000,-200750,-70750,-90000,3),
(-137750,-153500,-109250,-128500,4),
(-185000,-200750,-51500,-70750,2),
(-216500,-232250,-13000,-32250,1),
(-216500,-232250,-51500,-70750,3),
(-185000,-200750,-32250,-51500,3),
(-200750,-216500,-13000,-32250,1),
(-200750,-216500,-32250,-51500,3),
(-185000,-200750,-128500,-147750,4),
(-216500,-248000,-90000,-128500,1),
(-216500,-248000,-128500,-167000,1),
(-185000,-200750,-109250,-128500,2),
(-200750,-216500,-90000,-109250,2),
(-200750,-216500,-109250,-128500,2),
(-185000,-192880,-99625,-109250,4),
(-192880,-200750,-90000,-99625,2),
(-169250,-177130,-99625,-109250,4),
(-169250,-185000,-13000,-32250,1),
(-169250,-185000,-32250,-51500,3),
(-153500,-169250,-70750,-90000,2),
(-169250,-185000,-51500,-70750,2),
(-169250,-185000,-70750,-90000,3),
(-122000,-137750,-147750,-167000,1),
(-137750,-153500,-128500,-147750,4),
(-137750,-153500,-147750,-167000,1),
(-153500,-169250,-109250,-128500,2),
(-169250,-177130,-90000,-99625,2),
(-169250,-185000,-109250,-128500,2),
(-208630,-216500,-51500,-61125,3),
(-161380,-169250,-90000,-99625,2),
(-161380,-169250,-99625,-109250,4),
(-153500,-169250,-147750,-167000,1),
(-169250,-185000,-128500,-147750,4),
(-169250,-185000,-147750,-167000,1),
(-200750,-208630,-51500,-61125,2),
(-200750,-216500,-70750,-90000,3),
(-216500,-232250,-32250,-51500,3),
(-232250,-248000,-13000,-32250,1),
(-232250,-248000,-32250,-51500,1),
(-232250,-248000,-51500,-70750,1),
(-232250,-248000,-70750,-90000,1),
(-185000,-200750,-147750,-167000,1),
(-200750,-216500,-147750,-167000,1),


        OTHERS => (0,0,0,0)
);
signal indica:std_logic;
signal aux:std_logic_vector(7 downto 0);
begin

memrom_prueba:process(x_prueba, y_prueba)
variable x: integer:=0;
variable y: integer:=0;
begin
```

```
        x := CONV_INTEGER(UNSIGNED(x_prueba))*1000;
        y := CONV_INTEGER(UNSIGNED(y_prueba))*1000;

        for i in 0 to PJ-1 loop
                if(i<2)then
                        MEMROM_X_PRUEBA (i)<= x;
                else
                        MEMROM_X_PRUEBA (i)<= y;
                end if;
        end loop;

end process;


gen_particiones:for i in 0 to PI-1 generate
        gen1:process(MEMROM_X_PRUEBA)
        variable min_i: integer;

        begin
                        for j in 0 to PJ-1 loop
                                if( j = 0)then
                                        min_i := ((-1)**(j+1))*(-1*(MEMROM_MATRIZ_P(i,j)+MEMROM_X_PRUEBA(j)));
                                else
                                        -- encontrar el minimo
                                        if(min_i < (((-1)**(j+1)) * (-1*(MEMROM_MATRIZ_P(i,j)+MEMROM_X_PRUEBA(j))))then
                                                min_i := min_i;
                                        else
                                                min_i := ((-1)**(j+1)) * (-1*(MEMROM_MATRIZ_P(i,j)+MEMROM_X_PRUEBA(j)));
                                        end if;
                                end if;
                        end loop;
                        MEMRAM_res(i,0) <= i;
                        MEMRAM_res(i,1) <= min_i;
                        MEMRAM_res(i,2) <= MEMROM_MATRIZ_P(i,4);
   end process;
 end generate;


   EncontarMaximo:process(MEMRAM_res)
        variable max_j: integer:= MEMRAM_res(0,1);
        variable max_j_aux: integer;
        variable max_j_aux2: integer;
        --variable aux:std_logic_vector(7 downto 0);
        begin
                for i in 0 to PI-1 loop
                                max_j_aux := MEMRAM_res(i,1);

                                if(max_j < max_j_aux ) then
                                                max_j := max_j_aux;
                                                max_j_aux2 := MEMRAM_res(i,2);
                                else
                                                max_j := 0;
                                                max_j_aux2 := 0;
                                end if;

                                MEMRAM_res_max(i,0) <= i;
                                MEMRAM_res_max(i,1) <= max_j;
                                MEMRAM_res_max(i,2) <= max_j_aux2;
                end loop;
        end process;

        deco:process(MEMRAM_res_max,aux)
        begin

        for i in 0 to PI-1 loop
                if(MEMRAM_res_max(i,2)/=0)then
                        clase <= CONV_STD_LOGIC_VECTOR((MEMRAM_res_max(i,2)),8);
                        aux <= CONV_STD_LOGIC_VECTOR((MEMRAM_res_max(i,2)),8);
                        exit;
                else
                        clase <= (others =>'0');
                        aux <= (others =>'0');
                end if;
        end loop;

                case(aux) is
                                when "00000000" => dis_0 <= "1000000"; --0
```

```
                when "00000001" => dis_0 <= "1111001"; --1
                when "00000010" => dis_0 <= "0100100"; --2
                when "00000011" => dis_0 <= "0110000"; --3
                when "00000100" => dis_0 <= "0011001"; --4
                when "00000101" => dis_0 <= "0010010"; --5
                when "00000110" => dis_0 <= "0000010"; --6
                when "00000111" => dis_0 <= "0111000"; --7
                when "00001000" => dis_0 <= "0000000"; --8
                when "00001001" => dis_0 <= "0011000"; --9
                when others => dis_0 <= "1111111";
        end case;
        dis_1 <= "1000110";


case(x_prueba(3 downto 0)) is
        when "0000" => dis_4 <= "1000000"; --0
        when "0001" => dis_4 <= "1111001"; --1
        when "0010" => dis_4 <= "0100100"; --2
        when "0011" => dis_4 <= "0110000"; --3
        when "0100" => dis_4 <= "0011001"; --4
        when "0101" => dis_4 <= "0010010"; --5
        when "0110" => dis_4 <= "0000010"; --6
        when "0111" => dis_4 <= "0111000"; --7
        when "1000" => dis_4 <= "0000000"; --8
        when "1001" => dis_4 <= "0011000"; --9
        when "1010" => dis_4 <= "0001000"; --A
        when "1011" => dis_4 <= "0000011"; --B
        when "1100" => dis_4 <= "1000110"; --C
        when "1101" => dis_4 <= "0100001"; --D
        when "1110" => dis_4 <= "0000110"; --E
        when "1111" => dis_4 <= "0001110"; --F
        when others => dis_4 <= "1111111";
end case;

case(x_prueba(7 downto 4)) is
        when "0000" => dis_5 <= "1000000"; --0
        when "0001" => dis_5 <= "1111001"; --1
        when "0010" => dis_5 <= "0100100"; --2
        when "0011" => dis_5 <= "0110000"; --3
        when "0100" => dis_5 <= "0011001"; --4
        when "0101" => dis_5 <= "0010010"; --5
        when "0110" => dis_5 <= "0000010"; --6
        when "0111" => dis_5 <= "0111000"; --7
        when "1000" => dis_5 <= "0000000"; --8
        when "1001" => dis_5 <= "0011000"; --9
        when "1010" => dis_5 <= "0001000"; --A
        when "1011" => dis_5 <= "0000011"; --B
        when "1100" => dis_5 <= "1000110"; --C
        when "1101" => dis_5 <= "0100001"; --D
        when "1110" => dis_5 <= "0000110"; --E
        when "1111" => dis_5 <= "0001110"; --F
        when others => dis_5 <= "1111111";
end case;


case(y_prueba(3 downto 0)) is
        when "0000" => dis_6 <= "1000000"; --0
        when "0001" => dis_6 <= "1111001"; --1
        when "0010" => dis_6 <= "0100100"; --2
        when "0011" => dis_6 <= "0110000"; --3
        when "0100" => dis_6 <= "0011001"; --4
        when "0101" => dis_6 <= "0010010"; --5
        when "0110" => dis_6 <= "0000010"; --6
        when "0111" => dis_6 <= "0111000"; --7
        when "1000" => dis_6 <= "0000000"; --8
        when "1001" => dis_6 <= "0011000"; --9
        when "1010" => dis_6 <= "0001000"; --A
        when "1011" => dis_6 <= "0000011"; --B
        when "1100" => dis_6 <= "1000110"; --C
        when "1101" => dis_6 <= "0100001"; --D
        when "1110" => dis_6 <= "0000110"; --E
        when "1111" => dis_6 <= "0001110"; --F
        when others => dis_6 <= "1111111";
end case;

case(y_prueba(7 downto 4)) is
        when "0000" => dis_7 <= "1000000"; --0
        when "0001" => dis_7 <= "1111001"; --1
```

```
                        when "0010" => dis_7 <= "0100100"; --2
                        when "0011" => dis_7 <= "0110000"; --3
                        when "0100" => dis_7 <= "0011001"; --4
                        when "0101" => dis_7 <= "0010010"; --5
                        when "0110" => dis_7 <= "0000010"; --6
                        when "0111" => dis_7 <= "0111000"; --7
                        when "1000" => dis_7 <= "0000000"; --8
                        when "1001" => dis_7 <= "0011000"; --9
                        when "1010" => dis_7 <= "0001000"; --A
                        when "1011" => dis_7 <= "0000011"; --B
                        when "1100" => dis_7 <= "1000110"; --C
                        when "1101" => dis_7 <= "0100001"; --D
                        when "1110" => dis_7 <= "0000110"; --E
                        when "1111" => dis_7 <= "0001110"; --F
                        when others => dis_7 <= "1111111";
                end case;

        end process;

end Behavioral;
```

Apéndice C

Technology Map Viewer

En este apartado se muestran los reportes RTL generados para la implementación del entrenamiento de la RNMD para una y dos dimensiones con p clases, este tipo de reportes permite ver la estructura interna del diseño realizado.

C.1. Technology Map Viewer para la RNMD de 1 dimensión y p clases

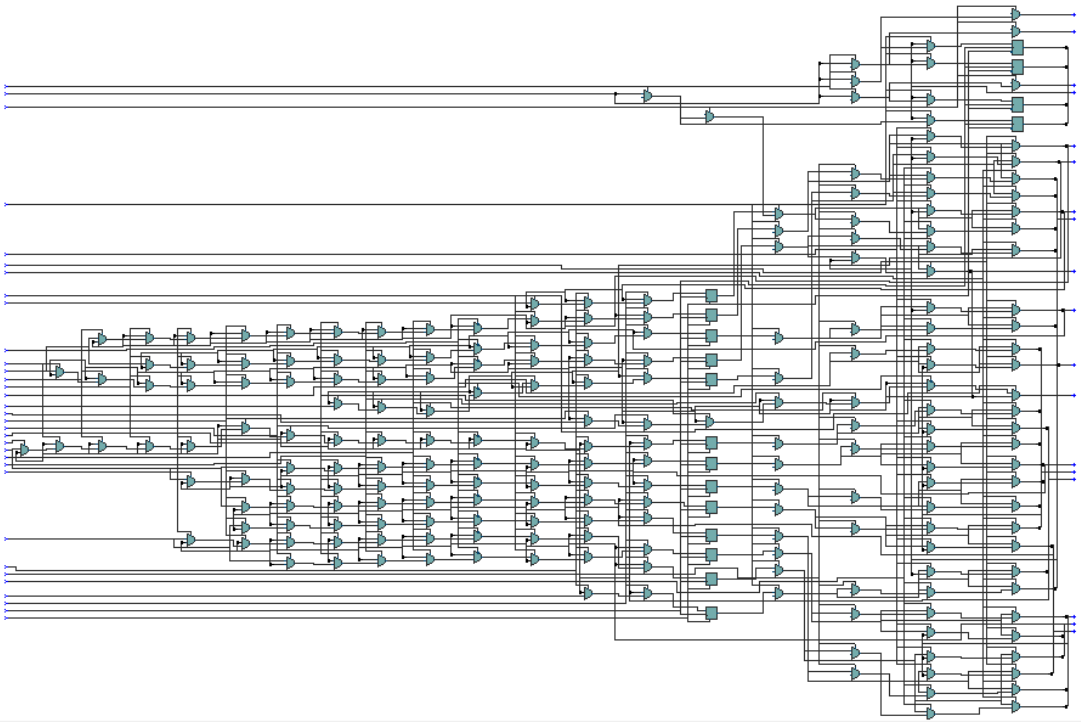

Figura C.1: Reporte RTL para la RNMD de p clases 1 dimensión

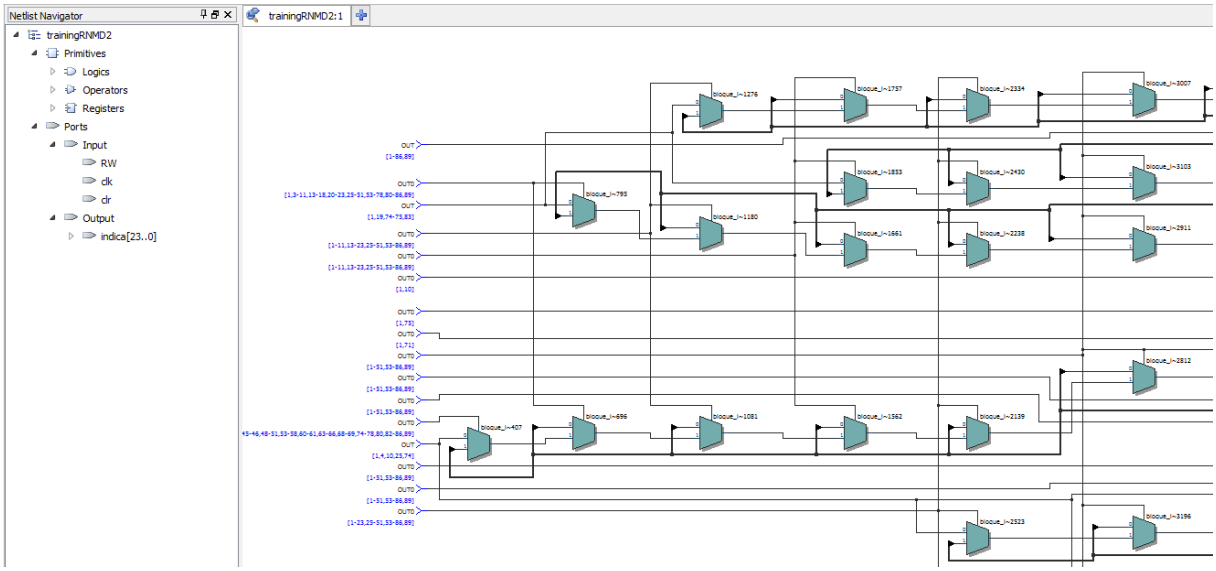

Figura C.2: Zoom reporte RTL para la RNMD de p clases 1 dimensión

C.2. Technology Map Viewer para la RNMD de 2 dimensiones y p clases

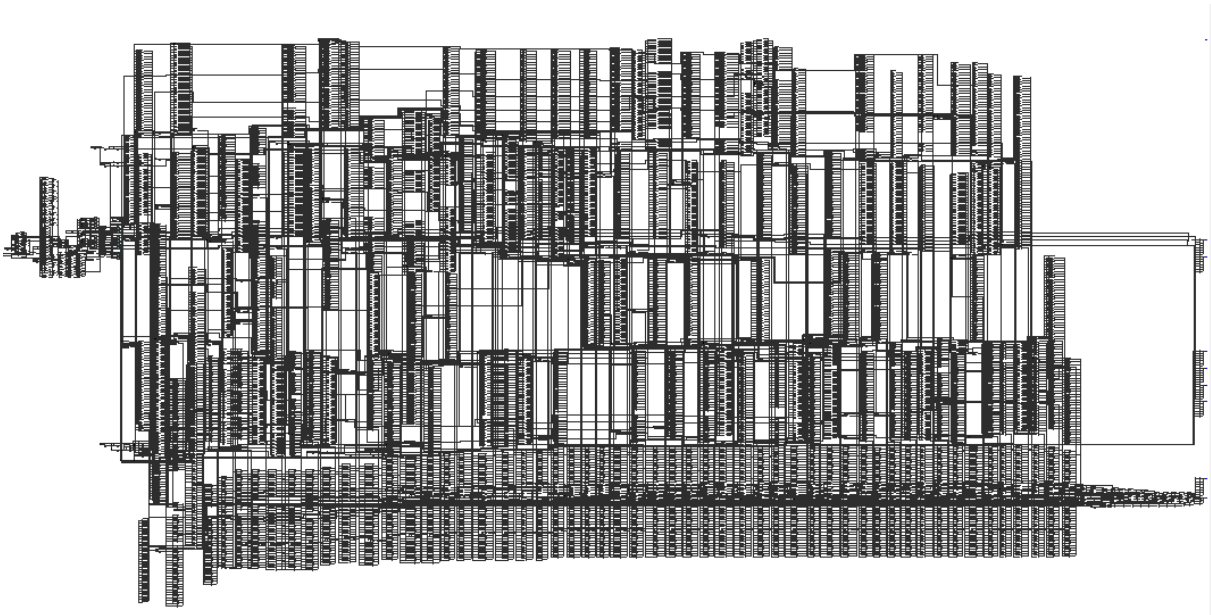

Figura C.3: Reporte RTL para la RNMD de p clases 2 dimensiones

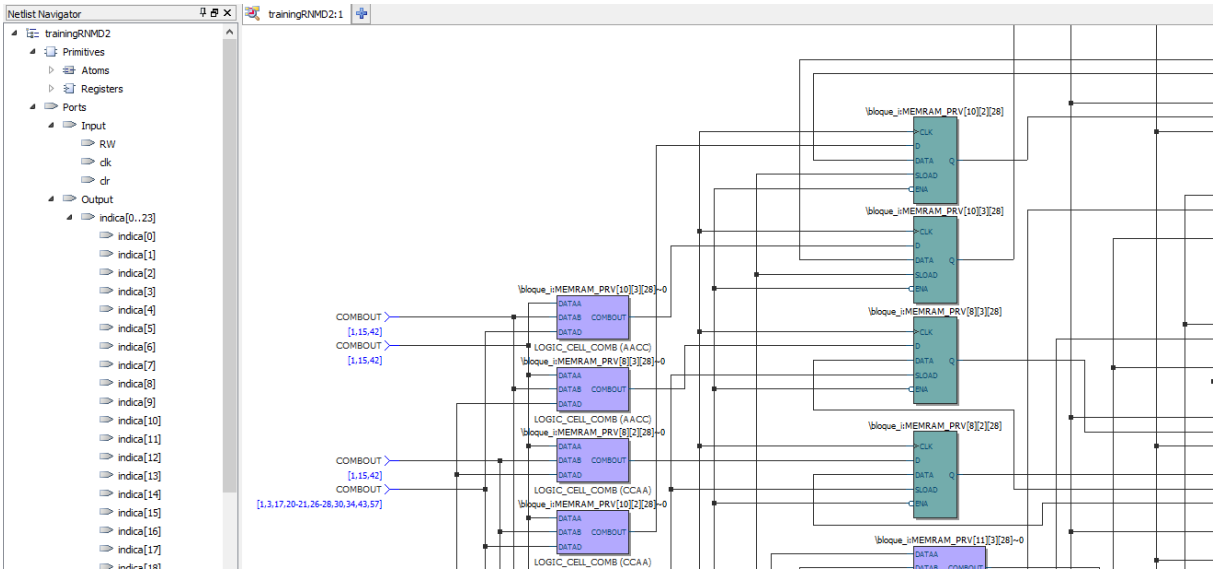

Figura C.4: Zoom reporte RTL para la RNMD de p clases 2 dimensiones