Universidad de Alcalá
Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN INGENIERÍA INDUSTRIAL

Trabajo Fin de Máster

DEFINICIÓN DE UNA ARQUITECTURA CONFIGURABLE PARA LA IMPLEMENTACIÓN DE REDES NEURONALES EN DISPOSITIVOS FPGA

Autor: Juan Carlos Barbero del Olmo

Tutor: Álvaro Hernández Alonso

2019

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN INGENIERÍA INDUSTRIAL

Trabajo Fin de Máster

DEFINICIÓN DE UNA ARQUITECTURA CONFIGURABLE PARA LA IMPLEMENTACIÓN DE REDES NEURONALES EN DISPOSITIVOS FPGA

Autor: Juan Carlos Barbero del Olmo

Director: Álvaro Hernández Alonso

TRIBUNAL:

Presidente: Ignacio Bravo Muñoz

Vocal 1º: Ignacio García Tejedor

Vocal 2º: Álvaro Hernández Alonso

FECHA: 13/07/201

Contenido

LISTA DE FIGURAS

"Siempre que te pregunten si puedes hacer un trabajo,

contesta que sí y ponte enseguida a aprender cómo se hace"

Franklin D. Roosevelt

A mi familia y a mi pareja

por su apoyo incondicional.

A Álvaro, mi tutor,

por creer en mi capacidad

y por su inestimable ayuda

Resumen

En los últimos años, en el mundo de la robótica cada vez es más importante el campo de la Inteligencia Artificial y el Machine Learning, donde, además, cada vez cobran más importancia las redes neuronales. El propósito de este trabajo es el diseño de una arquitectura flexible y eficiente para la implementación de redes neuronales en un dispositivo FPGA; para ello, se atenderá a la maximización en la eficiencia de los recursos, sin reducir la frecuencia de actualización de la red neuronal y la latencia de ésta. Finalmente se mostrarán ejemplos de cómo generar algunas estructuras basadas en la arquitectura propuesta.

Palabras clave

- Red Neuronal.
- FPGA.
- Eficiencia.
- Flexibilidad.
- Robótica.

Abstract

In recent years, in the field of Robotics, Artificial Intelligence and Machine Learning has become increasingly important, and neural networks are even more interesting. The purpose of this work is the design of a flexible and efficient architecture for the implementation of neural networks in an FPGA device; for that , we will focus on maximizing the efficiency of resources, without reducing the update frequency of the neural network and its latency. Finally, some examples will be shown about how to generate certain structures based on the proposed architecture.

Key Words

- Neural Network.
- FPGA.
- Efficiency.
- Flexibility.
- Robotics.

Resumen Extendido

En los últimos años, en el mundo de la robótica cada vez es más importante el campo de la Inteligencia Artificial y el Machine Learning; además, cada vez cobran más importancia las redes neuronales, estas estructuras son susceptibles de implementarse en un dispositivo FPGA debido a su carácter paralelo. Actualmente se utilizan dispositivos de procesado gráfico (GPUs), pero estos dispositivos tienen habitualmente un coste relativamente elevado; esto es un desembolso importante de dinero comparado con el precio de una FPGA que suele ser bastante inferior.

El propósito de este trabajo es la definición y diseño de una arquitectura flexible y eficiente para la implementación de redes neuronales en un dispositivo FPGA; para ello, se atenderá a la maximización en la eficiencia de los recursos, sin reducir la frecuencia de actualización de la red neuronal y la latencia de esta. Se propone una arquitectura descrita en su totalidad en lenguaje VHDL, para una mayor optimización del código. En una fase preliminar, se analizaron varias opciones para la realización de este proyecto:

- Utilizar el generador de código automático a partir de bloques Simulink que Matlab proporciona con sus herramientas; este generador produce modelos directamente en VHDL, pero tiene un nivel muy bajo de optimización, por lo que se descartó.
- Utilizar Vivado HLS, que es una herramienta que permite programar en lenguaje C el comportamiento que se desea para el diseño y la herramienta se encarga de generar automáticamente el código para la FPGA. Se descartó porque se le quería dar más versatilidad a la arquitectura a través de genéricos.
- Finalmente, se propuso realizar la codificación directa en VHDL, que permite una alta versatilidad a través de los genéricos y, además, un alto grado de optimización, ya que usa directamente las primitivas del dispositivo, evitando así que se autoinfieran elementos no previstas.

Además, se desea que en este proyecto se utilicen las arquitecturas de FPGA más modernas; para ello, se ha escogido el fabricante Xilinx y su serie de System-on-Chips denominada “Zynq”, muy interesante debido a que aúna en una sola pastilla dos núcleos ARM Cortex A9 y una FPGA de última generación.

Finalmente se mostrarán ejemplos de cómo generar algunas estructuras basadas en la arquitectura propuesta. Se validarán todas las etapas de la arquitectura mediante simulaciones hechas con la herramienta del fabricante (Vivado) y además se simularán algunas redes neuronales sencillas para poder validar la arquitectura correctamente. Igualmente, se mostrarán los datos de utilización de los recursos de los que se disponen y se hará un informe sobre el desempeño de la arquitectura en cuanto a tiempos de ejecución, frecuencia de funcionamiento, latencia y limitaciones en cuanto a capacidad de representar redes grandes.

1. Introducción

1.1. Presentación

Con el auge de la robótica en nuestros tiempos se ha extendido el uso de robots en muchos ámbitos de nuestra vida. Se pueden ver en nuestra casa, absorbiendo la suciedad de los suelos mientras dormimos, también en los centros de logística, encargados de repartir la paquetería entre las distintas estanterías de los almacenes, por no hablar de las famosas competiciones de robots o los robots más avanzados de investigación que son capaces de mantener conversaciones con humanos.

Todo esto hubiese sido imposible si de la mano no hubiese sufrido también un gran avance otras disciplinas como la inteligencia artificial o el aprendizaje automático, también llamado "Machine Learning". El "Machine Learning" es el responsable de que los robots sean capaces de aprender por sí mismos, que el robot que limpia nuestros suelos sea capaz de pasear por nuestra casa y sepa cuáles son las zonas más sucias, además de ser capaz de confeccionar un mapa de las habitaciones y volver sano y salvo a su estación de carga.

Otro ejemplo sería el de reconocimiento de imágenes: los robots cada vez son más capaces de reconocer ciertos objetos, desde la cámara que permite en las gasolineras pagar directamente cuando reconoce cuál es el número de nuestra matrícula hasta los radares que hacen una foto de la matrícula y son capaces de autogenerar una multa a partir del reconocimiento.

Esto es posible gracias a unos elementos que incorporan la inteligencia del robot que son las redes neuronales; estas redes neuronales actúan igual que el cerebro humano (ya que están inspiradas en la propia biología), son un conjunto de neuronas, cada neurona está interconectada con otras neuronas y cada una de ellas es capaz de enviar un estímulo a las neuronas con las que está conectada, provocando que, si se sobrepasa un cierto umbral, esa neurona se active. Estas conexiones tienen una cierta fuerza, que va a provocar que la neurona sea más o menos sensible a ese estímulo; la variación de esa sensibilidad en cada una de las conexiones es la que va a propiciar el aprendizaje.

Históricamente, estas neuronas son ejecutadas por un ordenador (o procesador). El problema de los procesadores es que son secuenciales, es decir, solo pueden hacer una operación a la vez. En una red neuronal real (biológica), todas estas conexiones y estímulos suceden simultáneamente, formando lo que se denomina la sinapsis.

Actualmente existen ordenadores muy potentes que son capaces de ejecutar operaciones a GHz, pero siguen teniendo el problema de que no son capaces de realizar operaciones simultáneas, además de que el precio de un procesador potente es elevado. Además, se quiere que los robots cada vez sean más baratos, ya que nadie quiere pagar mucho dinero por un robot que barra el suelo, pero todo el mundo querría tener uno a un precio reducido.

En esta situación nace el propósito de este TFG, dotar a este problema de una solución con un precio reducido y que además sea capaz de ejecutar una red neuronas de forma simultánea (concurrente); para ello, se hará uso de un elemento llamado FPGA, que consiste en una pastilla de elementos electrónicos que inicialmente están sin conectar entre ellos y que mediante una configuración permiten interconectarlos formando circuitos electrónicos que pueden hacer operaciones simultáneas. Además, la arquitectura debe ser flexible, para poder adaptarse al mayor número de situaciones y redes que se puedan dar.

1.2. Interés de la propuesta

Tal y como se ha expuesto en la presentación, el interés de la propuesta consiste en conseguir una arquitectura flexible para una red neuronal que se pueda embarcar en una FPGA, ya que una FPGA normalmente va a ser más flexible, adaptable, y con un mayor grado de paralelismo, que un procesador que tenga que ejecutar la misma red a la misma frecuencia. Además, es importante conseguir una arquitectura eficiente, ya que será deseable que valga para una gran variedad de aplicaciones.

Además, es importante conseguir una arquitectura eficiente, ya que queremos que valga para una gran variedad de aplicaciones.

Este TFM se desarrolla en el Departamento de Electrónica de la Universidad de Alcalá, dentro del grupo de investigación GEINTRA, donde existe una gran experiencia previa en el desarrollo de sistemas sensoriales, de comunicaciones y procesamiento de señal en arquitecturas empotradas, muchas veces basadas en dispositivos FPGA. EN [1] se puede encontrar su página web, donde se podrán observar ejemplos de sus trabajos previos.

1.3. Objetivos

En este apartado se van a exponer los objetivos a conseguir en este trabajo de Fin de Master.

El objetivo general de este trabajo es conseguir la funcionalidad requerida (modelar una red neuronal en una FPGA) de tal manera que se adapte a la mayor cantidad posible de casos y que además funcione a la mayor frecuencia posible de reloj, para ello, se proponen los siguientes objetivos:

1. Demostrar que es posible embeber una red neuronal de gran tamaño en una FPGA. Este es el objetivo primario, ya que debe ser posible conseguir modelar una red neuronal en una FPGA para poder continuar el trabajo
2. Optimizar los recursos de la FPGA. Este objetivo es muy importante, ya que permitiría generar redes neuronales mayores y abarcar así más casos de uso.
3. Dotar de flexibilidad a la arquitectura (flexibilidad de datos y de datapath). Igual que en el caso anterior, una mayor flexibilidad permite adaptar la arquitectura a mayor número de casos, aumentando la utilidad de ésta.
4. Conseguir una frecuencia de actualización alta y una latencia baja (en la medida de que no limite la optimización de recursos). Una mayor frecuencia de actualización permite el uso de la arquitectura en casos en los que la velocidad de refresco es importante, como por ejemplo los sistemas en tiempo real que se utilizan en robótica.
5. Conseguir una solución barata. Para cualquier uso comercial, es importante el coste, por eso se propone el uso de un dispositivo FPGA, que es un claro ejemplo de buena relación entre potencia y precio.
6. Conseguir una frecuencia del reloj del sistema elevada. Este indicador repercute directamente en la frecuencia de actualización y en la latencia del sistema, a una mayor frecuencia de reloj, se consiguen mejores resultados de temporización.
7. Diseñar todo el sistema directamente en código VHDL. Este objetivo se plantea a favor de la eficiencia, ya que con código VHDL se consigue un alto grado de optimización de recursos.

En los siguientes apartados se expondrá el trabajo realizado para la consecución de estos objetivos.

1.4. Estructura del documento

En el capítulo 2 se van a exponer los antecedentes de la propuesta, dando una visión general de todas las herramientas, técnicas y dispositivos que participan en la realización de este proyecto, además, se da una visión breve de los trabajos previos que ya se han realizado en esta dirección y se aportan algunos artículos que pueden ser interesantes en esta línea.

En el capítulo 3 se realiza un análisis detallado de la arquitectura propuesta para la implementación de redes neuronales en una FPGA, para ello, se realiza un análisis Bottom-Top, es decir, se exponen una a una los módulos que componen la arquitectura, de menor nivel de jerarquía a mayor nivel de jerarquía.

En el capítulo 4 se muestran los resultados experimentales, una vez expuestos los módulos que componen la arquitectura en el capítulo 3, en el capítulo 4 se muestran simulaciones del comportamiento de estos módulos y se validan los mismos haciendo diversas simulaciones de los mismos.

En el capítulo 5 se realiza una presentación de la plataforma objetivo a la que se ha destinado la arquitectura y se hace un análisis de los consumos de recursos que presentan los distintos elementos de la arquitectura.

En el capítulo 6 se establecen los límites de la arquitectura, una vez que se da a conocer el consumo de recursos de la arquitectura en el capítulo 5, en el capítulo 6 se muestran las limitaciones de tamaño, la frecuencia máxima de funcionamiento, latencias, frecuencia de actualización y demás datos importantes de la arquitectura.

Finalmente, en el capítulo 7 se encuentran las conclusiones que se pueden extraer del trabajo realizado y también se recogen los trabajos futuros que nacen de este proyecto y que pueden ser atacados en un futuro para la mejora de este proyecto.

2. Antecedentes

Para la realización de este Trabajo de Fin de Máster se han utilizado una serie de herramientas y técnicas que van a ser explicadas en este capítulo; consta de una serie de apartados en los que se explica el estado del arte de una temática concreta que es importante para este trabajo.

2.1. Dispositivos FPGAs

En este primer apartado se va a hablar sobre qué es una FPGA (Field Programmable Gate Array). Las FPGAs, como la de la Figura 1, son chips que en una sola pastilla de Silicio contienen bloques lógicos, celdas multiplicadoras/DSP y memoria, y las interconexiones son programables eléctricamente. Esta programación puede ser cambiada cuantas veces se quiera (hasta rotura del chip) y permite una versatilidad que no tienen los procesadores, ya que son muy genéricas y se amoldan a cualquier diseño. Por otro lado, son menos eficientes que un procesador haciendo cálculos en coma flotante y los recursos de una FPGA son limitados.

Las FPGAs también se suelen utilizar como expansores de puertos o como pasarelas hacia otros sistemas porque suelen tener una gran cantidad de pines configurables como entradas y/o salidas.

Figura 1. Ejemplo de una FPGA. Fuente: https://academy.bit2me.com/que-es-fpga/

Además, las FPGAs tienen la ventaja de poder ejecutar muchas operaciones de forma simultánea, ya que la lógica se puede poner en paralelo; ésta es una de las características que más se usará en este trabajo, por lo que se puede usar para acelerar procesos que en un procesador serían más costosos de hacer debido a su proceso lineal.

2.2. Arquitecturas SoC

Por otro lado, se denomina System-on-Chip a aquellos circuitos integrados donde en un solo chip se encapsula el sistema; históricamente, para poder hacer una tarjeta electrónica, usualmente se integraban varios chips, como DSPs (Digital Signal Processors), FPGAs y otro tipo de chips que fuesen específicos de comunicaciones. Actualmente, están en pleno auge los SoC como el Zynq porque en una sola pastilla se incluyen dos núcleos ARM A9 de propósito general, un gran número de periféricos (CAN, SPI, Ethernet, etc.) y una zona de FPGA con posibilidad de interconexión entre estos tres elementos. Estas características dotan al chip de gran versatilidad y utilidad. Por un lado, se tiene la parte de procesadores, que le dan una gran potencia de cálculo al sistema; y, por otro lado, también se tiene la versatilidad de la FPGA, que permite adaptar el diseño a casi cualquier problema.

Estos SoC se utilizan en muchos ámbitos, como la automoción, la aeronáutica, comunicaciones, energía, robótica, etc. De hecho, en robótica es especialmente útil, ya que permiten desarrollar una gran cantidad de periféricos especializados, además tener dos núcleos para todas las tareas numéricas pesadas (como filtros o lazos de control). Un ejemplo de SoC ampliamente conocido y con una gran versatilidad es el que va embebido en una Raspberry Pi como la de la Figura 2.

Figura 2. Ejemplo de SoC (Raspberry Pi). Fuente: https://en.wikipedia.org/wiki/System_on_a_chip

Como en el caso de las FPGA, estos SoC suelen ser vendidos en tarjetas de evaluación de distintas gamas para poder probar su utilidad. En la Figura 3 y en la Figura 3 se muestran dos ejemplos de dos tarjetas de evaluación que contienen el dispositivo Zynq 7000, que es el chip que se utilizará en este trabajo. En la Figura 3 se ve una PicoZed, que está pensada para conectarla sobre otro sistema como elemento central, o formar parte de un sistema con inteligencia distribuida.

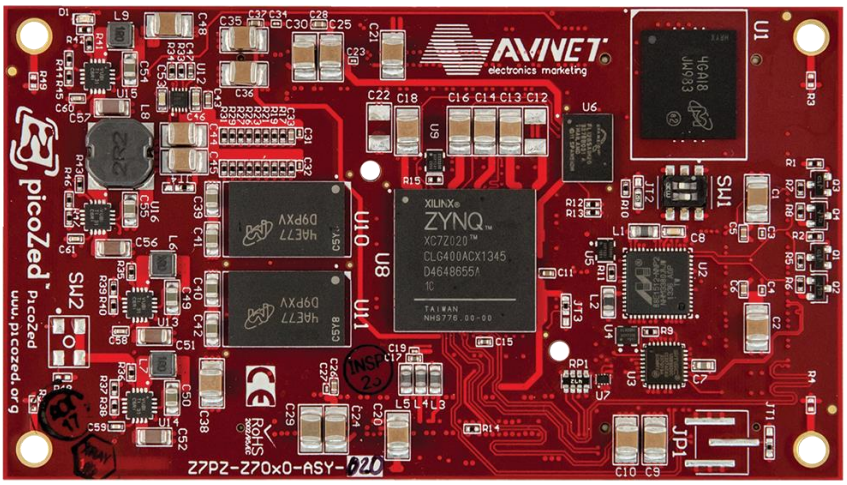

Figura 3. Imagen de la plataforma de desarrollo PicoZed. Fuente: http://zedboard.org/product/picozed

Por otro lado, en la Figura 4 se tiene la tarjeta ZC702, que posee una gran cantidad de conectores y elementos como SD o ethernet, más pensado para el desarrollo de una arquitectura Stand-Alone.

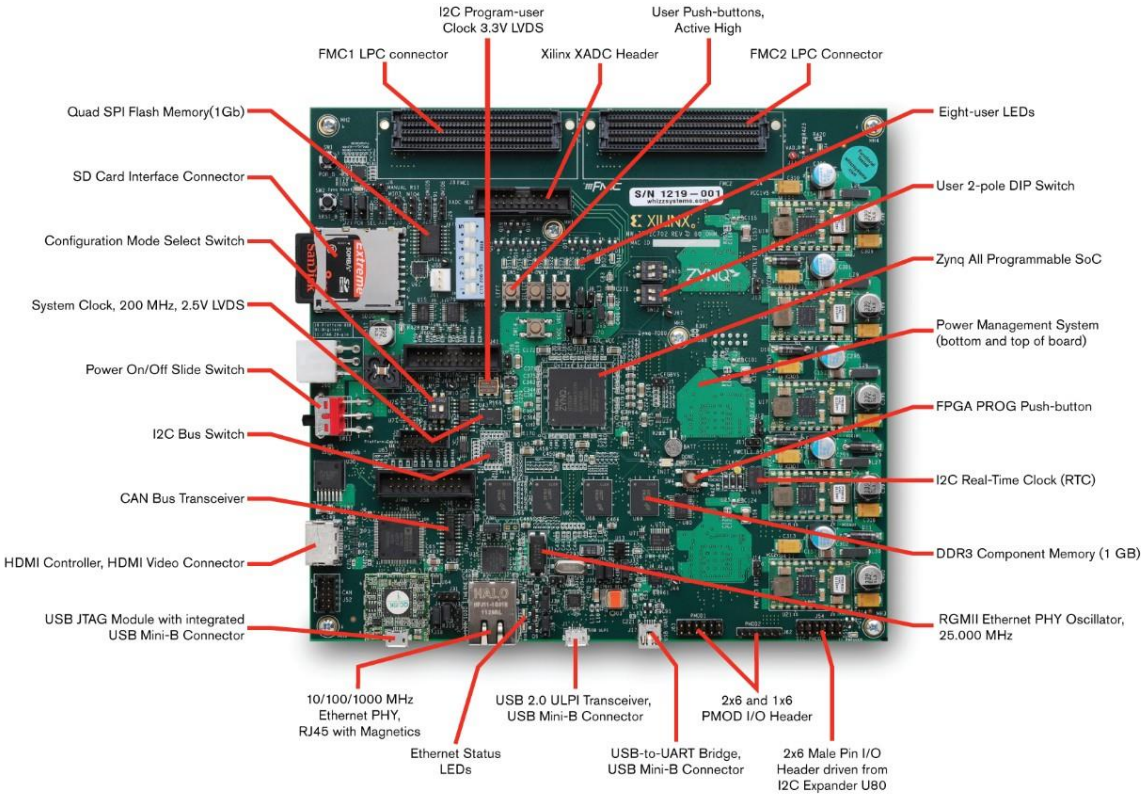

Figura 4. Imagen de la tarjeta de desarrollo Zc702. Fuente: https://www.xilinx.com/

En nuestro caso, se usará una ZedBoard, que es un paso intermedio entre estas dos tarjetas, proporcionando una descripción más detallada más adelante.

2.3. Redes Neuronales

Este trabajo se centra en las redes neuronales; las redes neuronales no son sino una parte pequeña de la inteligencia artificial. Una red neuronal es un conjunto de elementos simples (neuronas) que permiten detectar patrones en sus versiones más simples y modelar comportamientos complejos en sus versiones más avanzadas. En la Figura 5 se puede ver un ejemplo de una red neuronal simple.

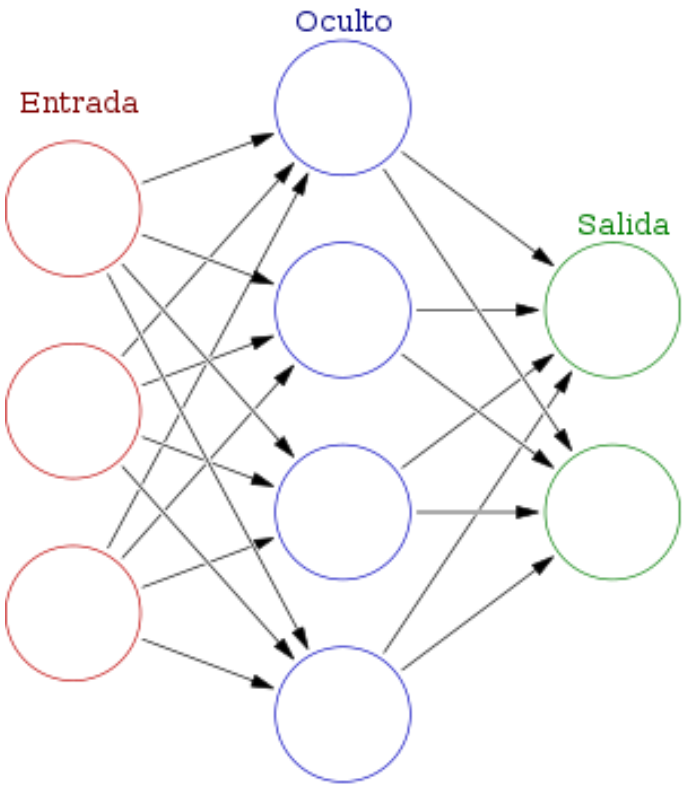

Figura 5. Ejemplo de topología de una red neuronal. Fuente: https://es.wikipedia.org/wiki/Red_neuronal_artificial

Hay muchos tipos de redes neuronales, cada tipo de red neuronal está diseñada para resolver algún tipo de problema, pero al final, todas ellas se componen de los mismos elementos, las neuronas y las interconexiones entre ellas. En la Figura 6 se puede ver una clasificación (del año 2016, actualmente hay más tipos) de los distintos tipos de redes. Este trabajo se centrará en aquellas redes que no tienen realimentación, las Feed Forward, aunque la arquitectura podría adaptarse en un futuro para admitir más tipos de redes.

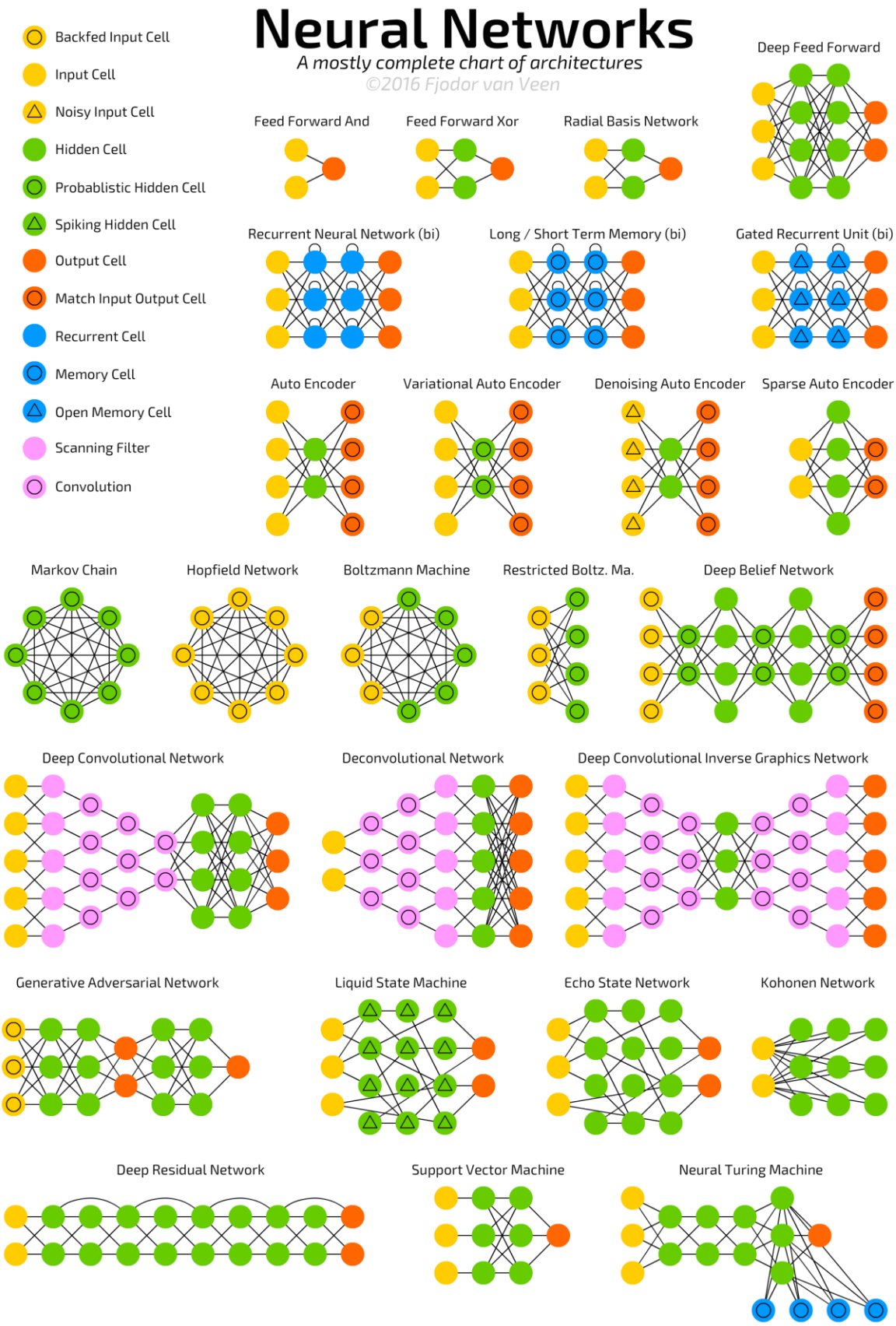

Figura 6. Tipos de red neuronal. Fuente: https://towardsdatascience.com/

Las redes por otro lado son utilizadas en distintos ámbitos; habitualmente se utilizan en la robótica, por ejemplo, los robots domésticos que limpian los suelos llevan redes neuronales preentrenadas para a través de los datos que reciben los sensores (que son las entradas a la red neuronal) decidir si el suelo sobre el que se mueven es madera, plaqueta o alfombra y actuar acorde a ello.

Por otro lado, las redes neuronales también pueden servir para cosas tan dispares como generar melodías que imiten a la música de Bach de manera totalmente independiente (https://www.google.com/doodles/celebrating-johann-sebastian-bach), o también son capaces de mostrar imágenes de personas que no existen en la realidad totalmente realistas de manera independiente (https://thispersondoesnotexist.com/). En el siguiente apartado se hablará de la inteligencia artificial, que es el siguiente salto cuando se habla de redes neuronales.

2.4. Machine Learning

El Machine Learning o Aprendizaje Automático es una rama de la inteligencia artificial que busca generar algoritmos y procedimientos para que un ente artificial aprenda de unos datos de manera autónoma. Actualmente se encuentra en auge junto con el denominado Big Data porque vivimos en una era digital en la que las empresas poseen millones de datos sobre sus clientes y desean sacar provecho de ellos. El Big Data se centra en sacar información útil de esos datos para que el Machine Learning se encargue de aprender de ellos y en muchos casos, obtener beneficios o ventajas con esa información.

Para ello, se suelen definir algoritmos que, a partir de unos datos, detecta tendencias en esos datos, generan un modelo de éstos y son capaces de tomar decisiones de acuerdo a lo aprendido de esos datos. Un ejemplo muy claro del aprendizaje automático son los móviles: los móviles tienen implementados algoritmos de aprendizaje automático, en base a los sensores de luz ambiente y de las intervenciones de subir o bajar brillo que tenga en su historial, es capaz de aprender el nivel de brillo deseado en cada situación, cambiándolo de manera automática cuando detecta que el nivel de iluminación cambia. Otro ejemplo, pero esta vez a gran escala está en el sistema de sugerencias de Amazon; su sistema aprende de las visitas a productos que se hace en su web y cuando detecta la tendencia, es capaz de recomendarte cosas que te gusten, aunque realmente no se le haya dicho que eso te gusta.

En el ámbito de la robótica y la electrónica este aprendizaje automático consiste en que a partir de los datos que se obtienen de una serie de sensores y el comportamiento que se desea, se entrena una red neuronal hasta que el elemento tiene el comportamiento deseado. Una vez que se tiene validado el funcionamiento de la red, esa red se queda estática y es cuando puede ser implantada en el robot (bien sea como un programa que debe ser ejecutado por un procesador o, como en el caso de este TFM, embebida en una FPGA).

2.5. Trabajos previos

En este apartado se va a dar una visión de los trabajos anteriores en este tema. No es el primer proyecto para implementar redes neuronales en FPGAs, en la bibliografía se pueden encontrar libros resumen de décadas de intentos y proyectos como en [2]. En [3] se puede encontrar un artículo sobre la incorporación de una red convolucional en una FPGA, que es un tipo de red que se va a poder realizar con la arquitectura que se propone en este TFM.

En [4] se puede encontrar un artículo sobre la optimización de una red neuronal en una FPGA, donde se realiza una multiplexación a nivel de capa. Es una implementación alternativa, ya que no se sigue la misma metodología que en este trabajo, pero es un buen ejemplo de cómo se puede realizar este trabajo. En la arquitectura de este trabajo se utilizan neuronas en las que no se multiplexan las entradas (consumiendo más recursos), pero, por otro lado, se utiliza una única Look-Up-Table para todas las funciones de activación, reduciendo el número de recursos, pero perjudicando el pipelining.

Como se ha expuesto, hay más opciones en cuanto a la implementación de la arquitectura en la FPGA, por lo que este trabajo se centra en la optimización y el añadir valor a las ya existentes usando las arquitecturas actuales de SoC.

3. Arquitectura propuesta

En este capítulo se va a detallar la arquitectura propuesta para la resolución del problema presentado; para ello, en primer lugar, se presenta un diagrama explicativo de la estructura de una red neuronal (Figura 7). A partir de este esquema se van a explicar cada una de esas unidades funcionales y su funcionamiento interno.

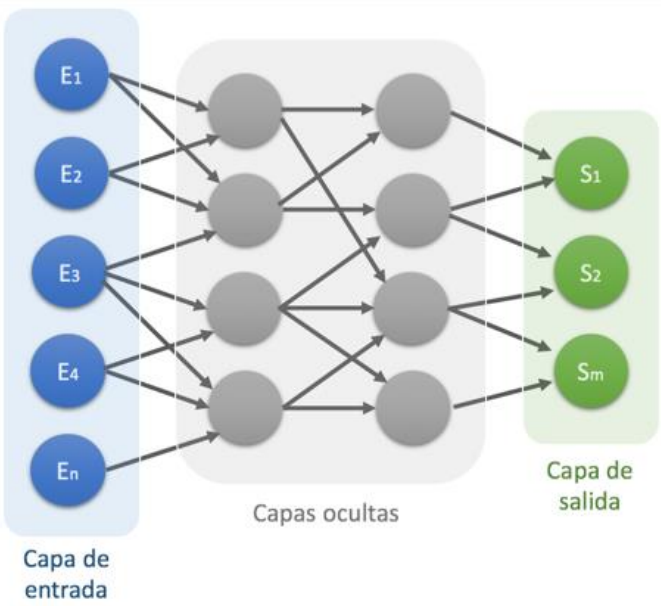

Figura 7. Estructura red neuronal. Fuente: http://www.diegocalvo.es/definicion-de-red-neuronal/

3.1. Neurona

La primera unidad funcional que debe pensarse en una red neuronal es, por supuesto, la neurona. El esquema de una neurona puede verse en la Figura 8.

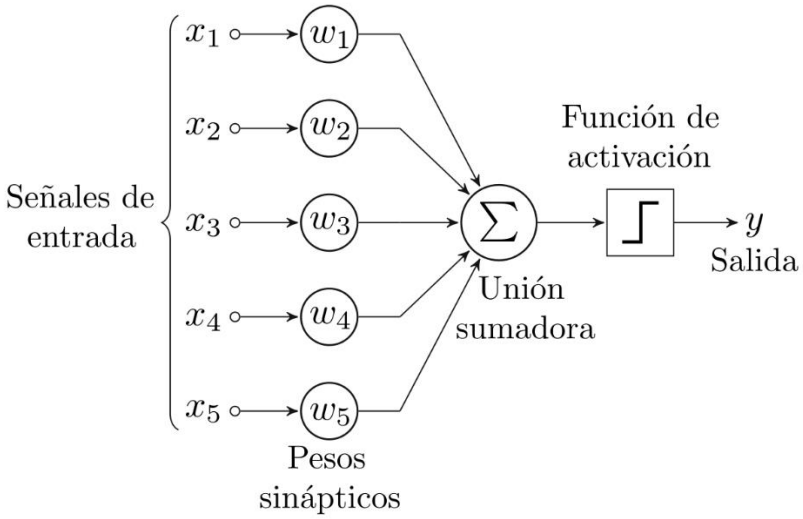

Figura 8. Estructura interna de una neurona. Fuente: https://es.wikipedia.org/wiki/Perceptrón

Como se puede observar, una neurona posee un número determinado de entradas; cada una se pondera por su respectivo peso; el resultado de estas multiplicaciones se suma; a continuación, se aplica una función de activación; y finalmente la salida de esta función es la salida de la neurona.

El número de entradas es un genérico en nuestro diseño VHDL, este número de entradas depende del número de neuronas que tenga la capa anterior o, si se trata de la capa de entrada, de las entradas que tenga la neurona. Como este número es susceptible de cambiar de una capa a otra y de un diseño a otro, es bastante lógico que sea un genérico. El número de pesos de la neurona es el mismo que el número de entradas, por lo que depende del mismo genérico. La unión sumadora no deja de ser un acumulador, tratado más adelante.

La función de activación es un tema interesante en la inteligencia artificial, ya que existen multitud de funciones de activación, como se puede ver en la Figura 9.

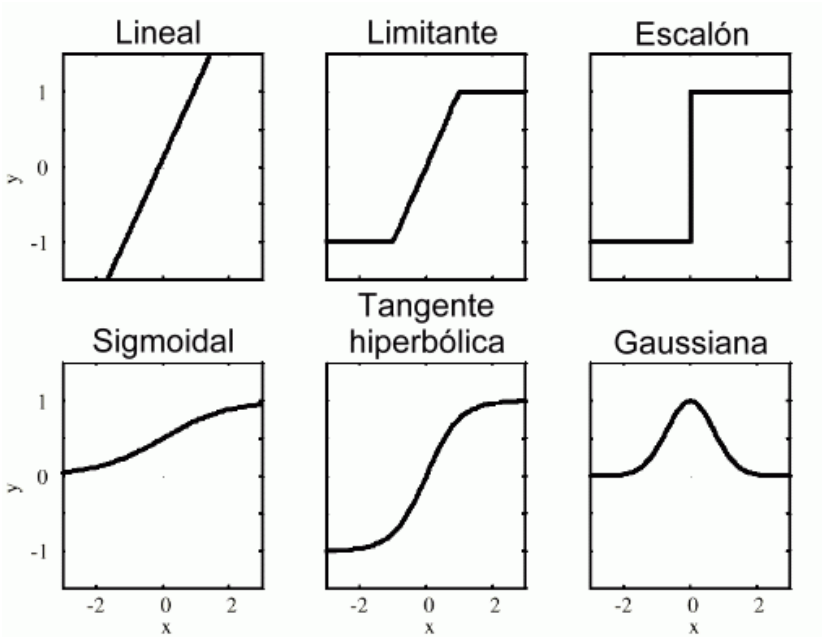

Figura 9. Funciones de activación. Fuente: https://www.um.es/LEQ/Atmosferas/Ch-VI-3/C63s6p2.htm

En este diseño se han implementado 3 funciones de activación distintas:

- Función sigmoidal (Figura 10)

$$y(t) = \frac{1}{1 + e^{-x}}$$

Ecuación 1. Función sigmoidal.

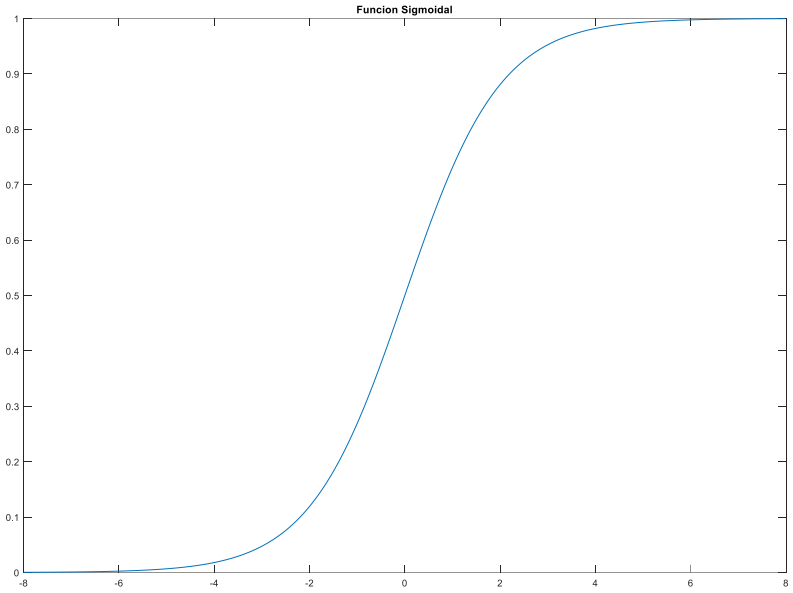

Figura 10. Función sigmoidal.

- Tangente hiperbólica (Figura 11).

$$y(t) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

Ecuación 2. Función de tangente hiperbólica.

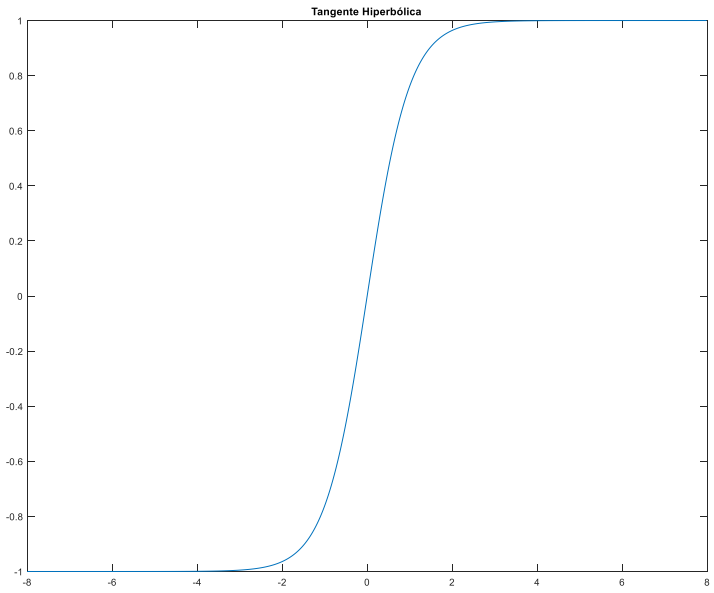

Figura 11. Tangente hiperbólica.

- Tangente sigmoidal (Figura 12).

$$y(t) = \frac{2}{1 + e^{-2x}} - 1$$

Ecuación 3. Función de tangente sigmoidal.

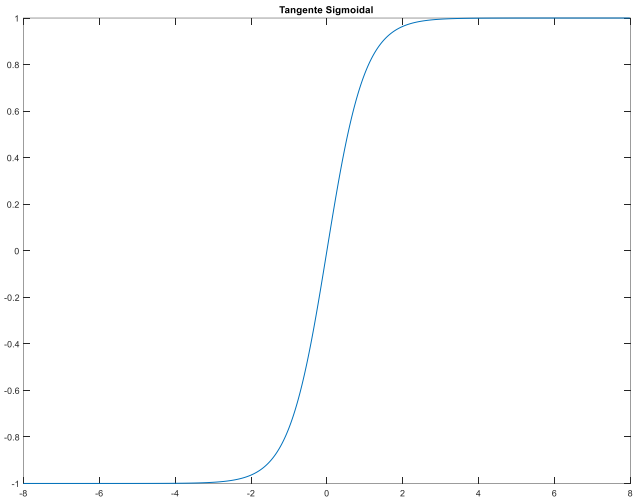

Figura 12. Tangente sigmoidal.

Una vez se tiene presentada la neurona, se tratan a continuación los detalles de su implementación en VHDL.

3.1.1. Entidad y Parámetros genéricos

En este primer apartado, se aborda la entidad que se ha elegido para la neurona y cuáles son los parámetros que recibe como genéricos para su configuración pre-síntesis. En la Figura 13 se puede observar la entidad en lenguaje VHDL.

```
entity Neuron is
    generic(
                NumOfInputs     : integer :=1;
                Datalength      : integer :=18;
                DecimalLength   : integer :=14
            );

    Port    (
                clk             : in    STD_LOGIC;
                rst             : in    STD_LOGIC;
                Neuron_enable   : in    STD_LOGIC;
                Input           : in    STD_LOGIC_VECTOR(Datalength-1 downto 0);
                Weight          : in    STD_LOGIC_VECTOR(Datalength-1 downto 0);
                Output          : out   STD_LOGIC_VECTOR(Datalength-1 downto 0);
                Out_valid       : out   STD_LOGIC
            );
end Neuron;
```

Figura 13. Entidad de la neurona.

A continuación, se realiza una pequeña descripción de los genéricos:

- *NumOfInputs*: este genérico define el número de entradas que va a tener la neurona.
- *DataLength*: este genérico define cuántos bits va a tener la representación de los datos.
- *DecimalLength*: este genérico define cuántos bits del *Datalength* se van a utilizar para representar la parte decimal de los datos.

De forma similar, se proporciona ahora una descripción de las entradas y las salidas:

- **Clk**: señal de reloj del diseño.
- **Rst**: señal de reset activa a nivel alto.
- **Neuron_enable**: señal de habilitación de la neurona.
- **Input**: bus por donde se recibe la entrada a procesar.
- **Weight**: bus por donde se recibe el peso correspondiente a la entrada a procesar.
- **Output**: resultado de la neurona.
- **Out_Valid**: señal que se activa cuando la salida es válida.

3.1.2. Diagrama de bloques

En la Figura 14 se muestra el bloque que se usará posteriormente en la capa; este bloque es el que se deduce de la entidad utilizada en el fichero VHDL.

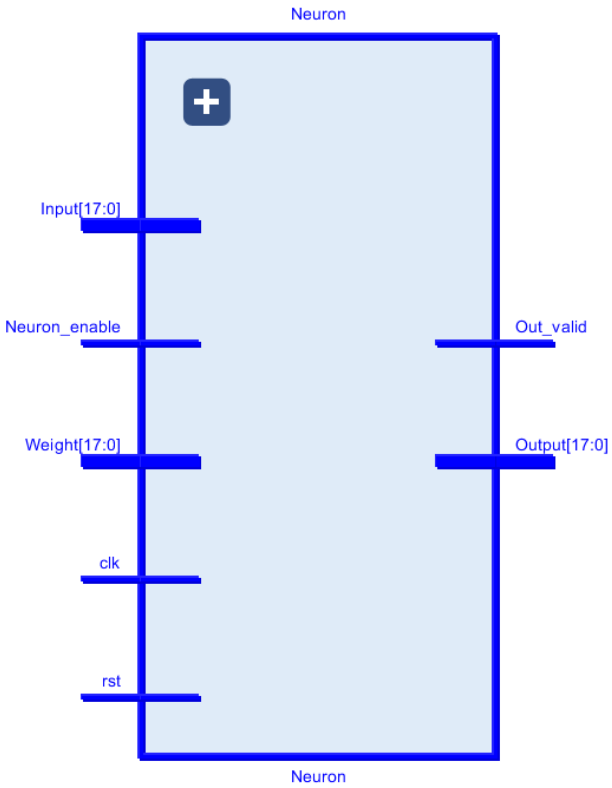

Figura 14. Bloque neurona.

3.1.3. Representación numérica

Para la realización de este proyecto se ha decidido usar una representación numérica en coma fija, es decir, del ancho de bits utilizado para representar un número se escoge una cantidad de bits determinada para la parte entera y el resto para la parte decimal. Este reparto puede ser arbitrario, por lo que se definen dos genéricos, uno para el número de bits con los que se va a representar el dato y otro que permite definir el número de bits que se van a dedicar a la representación de la parte decimal.

Existe una limitación, el número máximo de bits que se pueden utilizar para la representación de un número es de 18, ya que se tienen que poder almacenar en una BRAM que tiene un ancho de bus de

datos de 18 bits, así como ajustar al tamaño de 25x18 bits de los multiplicadores existentes en las arquitecturas habituales de FPGA.

3.1.4. Entradas y pesos

No hay que olvidar que el principal objetivo es que la arquitectura sea flexible y que además se desea que no suponga un consumo de recursos elevado, ya que, en una FPGA, estos recursos son limitados. Por estas razones se ha elegido que la neurona va a multiplexar las entradas y los respectivos pesos; es más, para simplificar al máximo la estructura de ésta, estos pesos y entradas serán alimentados desde fuera de la neurona, por lo que la neurona no va a necesitar ningún tipo de memoria para saber sus pesos, sino que serán alimentados desde el exterior.

3.1.5. Multiplicación y acumulación

Una vez se tiene el peso y la entrada, es necesario multiplicar el peso por la entrada y acumularlo, para realizar la multiplicación es necesario un multiplicador, o, en nuestro caso, ya que se está utilizando un dispositivo de la familia Zynq de la serie 7, una celda DSP48E1. Esta celda se puede ver en la Figura 15.

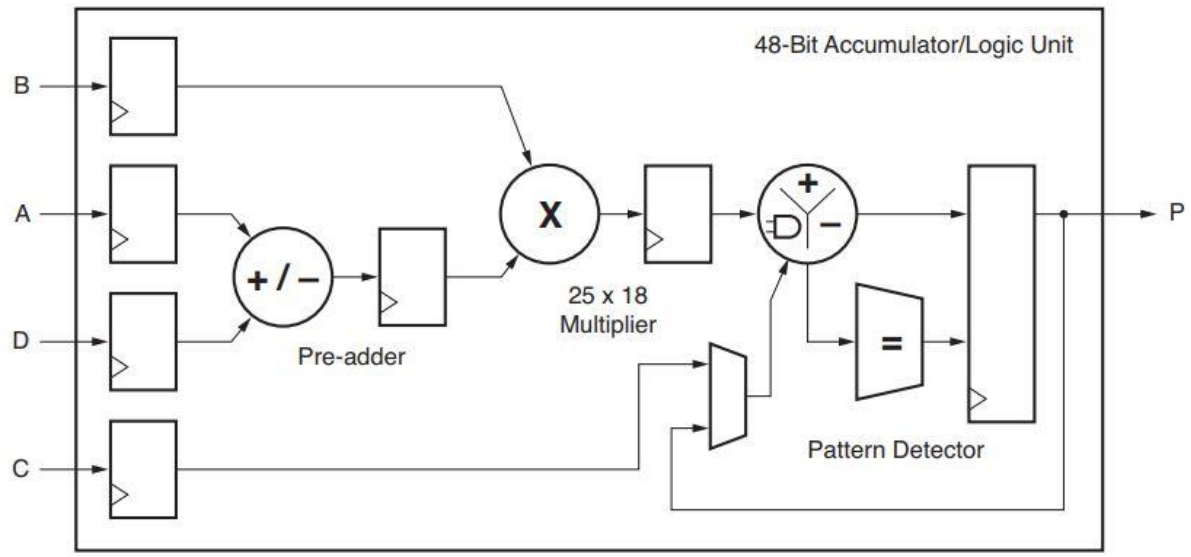

Figura 15. Esquema de una celda DSP48E1. Fuente: UG479_7Series_DSP48E1

Estas celdas DSP se pueden configurar de tal manera que actúen de multiplicador y acumulador, por tanto, resuelve el problema. Para realizar esta configuración, Xilinx proporciona unas macros que facilitan la configuración de esta celda DSP (compleja de por sí): la macro elegida es MACC_MACRO, que permite implementar un multiplicador+acumulador, y el bloque puede verse en la Figura 16.

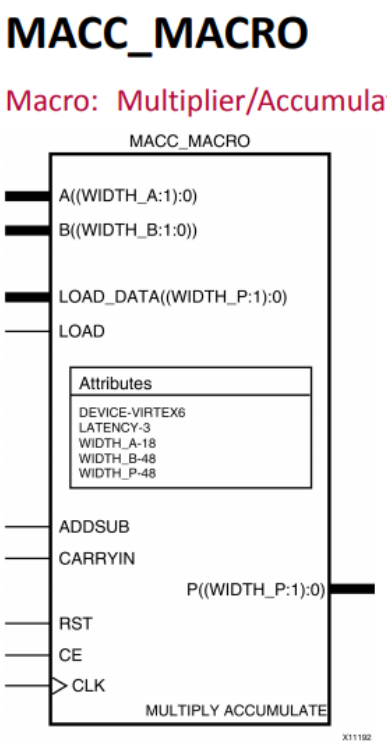

Figura 16. Bloque MACC_MACRO. Fuente: UG953-vivado-7series-libraries

Este bloque precisa de una cierta configuración, ya que posee unos ciertos atributos por defecto que pueden no servir para esta aplicación. En primer lugar, hay que configurar el ancho en bits de los datos que van a ser multiplicados (WIDTH_A y WIDTH_B); en este caso, esto va a ser fijo, va a tener un valor de 18 bits, este valor se debe a que es el máximo ancho que se puede guardar en una BRAM, como se verá posteriormente.

Por otro lado, se tiene el ancho de la salida (WIDTH_P); en este caso, se va a coger todo el ancho que proporciona el DSP, por lo que se elige 48 bits. Esto se hace porque a la salida del multiplicador se quiere detectar si se ha pasado del límite superior o del límite inferior de la función de activación y poder saturar. Finalmente, se configura la latencia deseada, que, en este caso, es de 1, es decir, que el DSP tarde 1 ciclo de reloj en devolver el resultado de la multiplicación y la acumulación. En la Figura 17 se puede observar el bloque que se ha implementado, y sus entradas y salidas.

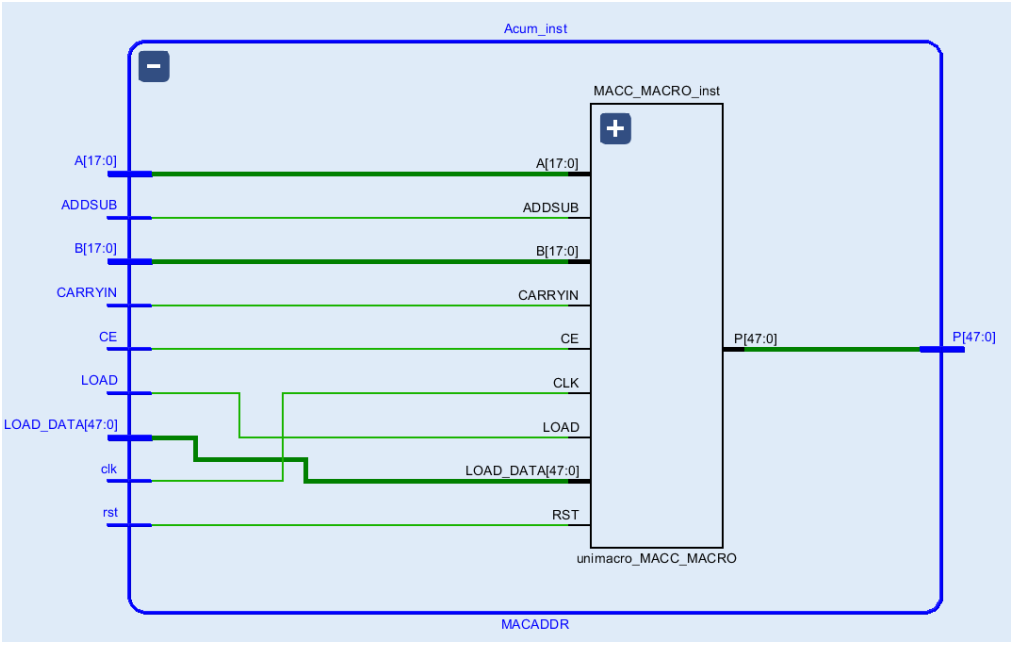

Figura 17. Bloque Multiplicador/Acumulador.

Entonces, después de este bloque ya se tendría el resultado de multiplicar una entrada por su peso y acumularlo con el resto de las entradas y pesos.

3.1.6. Función de activación

Para la implementación de la función de activación se plantearon varias opciones:

- Implementación directa de la fórmula.
- Linealización por tramos.
- Desarrollo en series de Taylor.
- Look-Up table.

La primera opción se descartó por el alto consumo de recursos para la implementación (y en algunos casos, directamente la imposibilidad de implementación). La segunda opción y la tercera consumía muchos recursos DSP, por lo que también se descartó. Finalmente se eligió la opción de Look-Up table porque es la más sencilla de implementar, además de que si se desea cambiar de función de activación es la más sencilla de adaptar al código, ya que se implementa mediante una memoria BRAM, por lo que, si se desea cambiar de función, solo hay que cambiar la inicialización de la memoria, que no es un proceso nada complicado.

Por tanto, una vez que se ha decidido que se va a utilizar una implementación mediante una memoria de tipo BRAM, hay que ver qué características tiene la misma y adaptarlas a nuestras necesidades. La familia que se está usando actualmente de FPGAs posee unos bloques de memoria de 36kb, como aparece en la Figura 18.

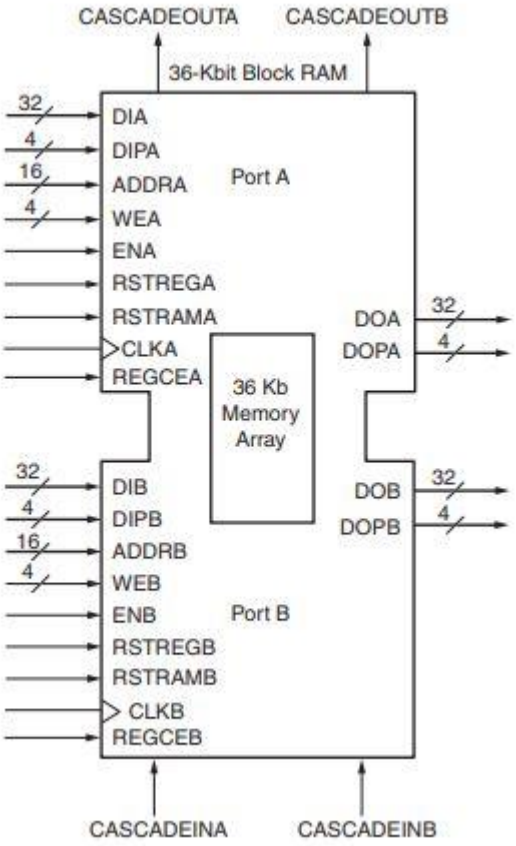

Figura 18. Esquema de una BRAM. Fuente: UG473_7Series_Memory_Resources

Estas BRAM a su vez están divididas internamente en dos bloques de 18 Kb, por lo que se ha decidido que el ancho del bus de datos va a ser de 18 bits y se van a tener 1024 posiciones (bus de datos de 10 bits). Como se puede ver en la Figura 18, estas BRAM son dual port, por lo que se puede acceder a las mismas desde dos puertos a la vez, tanto para escribir como para leer, por lo que habrá que tener esto en cuenta para el diseño.

Volviendo a la función de activación, se necesita una memoria que, introduciendo la salida del multiplicador/acumulador, devuelva el resultado de aplicar la función de transferencia, por lo que, en realidad lo que se necesita es fabricar una ROM. Además, esta ROM no necesita ser dual port, ya que solo va a ser accedida desde un sitio, por lo que, con estos requisitos, se busca una macro que permita configurar de manera sencilla estos bloques BRAM. Se ha determinado que la macro BRAM_SINGLE_MACRO se ajusta perfectamente a lo buscado, ya que implementa una BRAM del tamaño deseado con un solo puerto. El bloque de esta macro se puede ver en la Figura 19.

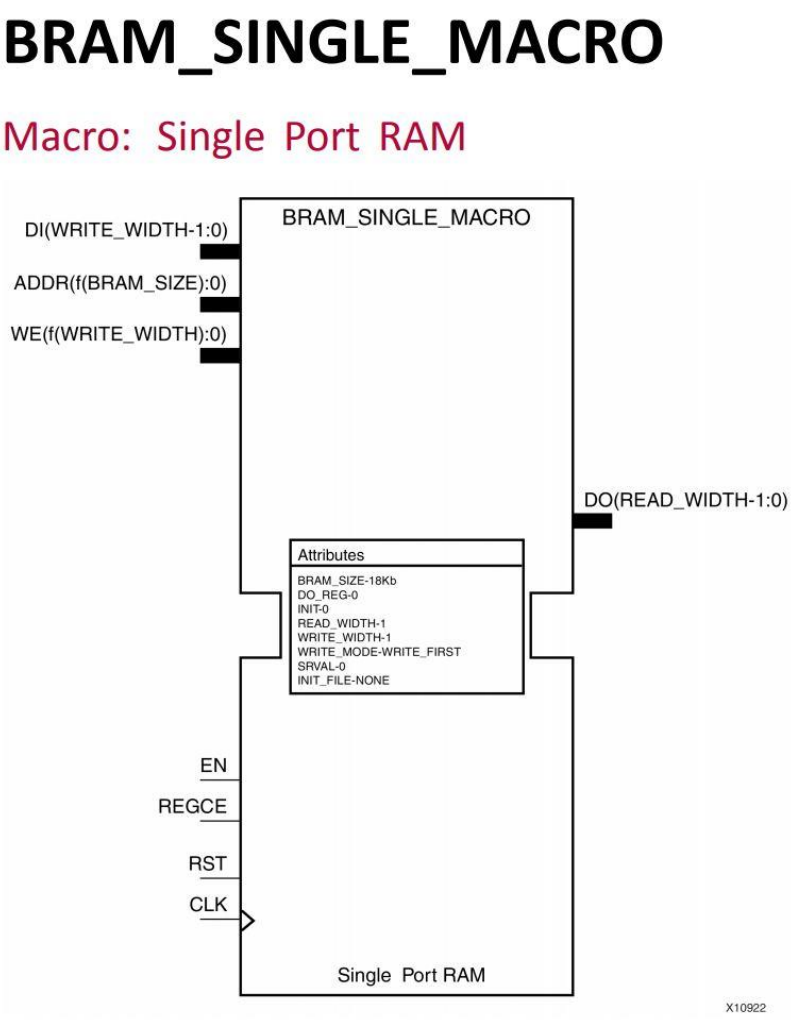

Figura 19. Esquema de BRAM_SINGLE_MACRO. Fuente: UG953-vivado-7series-libraries

Como en el caso anterior, esta macro necesita ser configurada, los atributos elegidos son los siguientes:

- BRAM_SIZE: este atributo permite configurar una BRAM de 36kb o de 18 kb, en nuestro caso, 18 kb.
- DO_REG: habilita un modo de escritura más rápido a costa de la latencia.
- READ_WIDTH: ancho del bus de lectura, en este caso 18 bits.
- WRITE_WIDTH: ancho del bus de escritura, en este caso 18 bits.
- WRITE_MODE: modo de escritura, en este caso, “NO_CHANGE”, ya que no se desea escribir (es una ROM).
- INIT_xx: a continuación, en bloques de 256 bits hay que especificarle la inicialización.

Con estos atributos estaría configurada la BRAM, faltaría resolver con qué valores se va a inicializar la BRAM y cómo se va a acceder a ella.

Empezando por cómo se accede a los datos de la BRAM, la BRAM tiene un ancho de bus de direcciones de 10 bits, por lo que lo primero que hay que hacer es recortar la salida del multiplicador/acumulador a 10 bits, recortando la precisión decimal de la salida del multiplicador/acumulador.

Para calcular los valores que deben ir en la BRAM se ha utilizado Matlab. Se ha escrito un script que indicándole los bits que tiene la representación, los bits de la parte decimal y definiendo la función de activación, utiliza dos cuantificadores (uno para cuantificar la entrada de la BRAM y otro para cuantificar la salida de la BRAM), y genera un fichero que contiene la inicialización que hay que meterle a la BRAM. Como ejemplo, se ha elegido un ancho de 18 bits, con 4 bits de parte entera y 14 de parte decimal, en la Figura 20 se muestra el máximo error generado con esta configuración para la función sigmoidal.

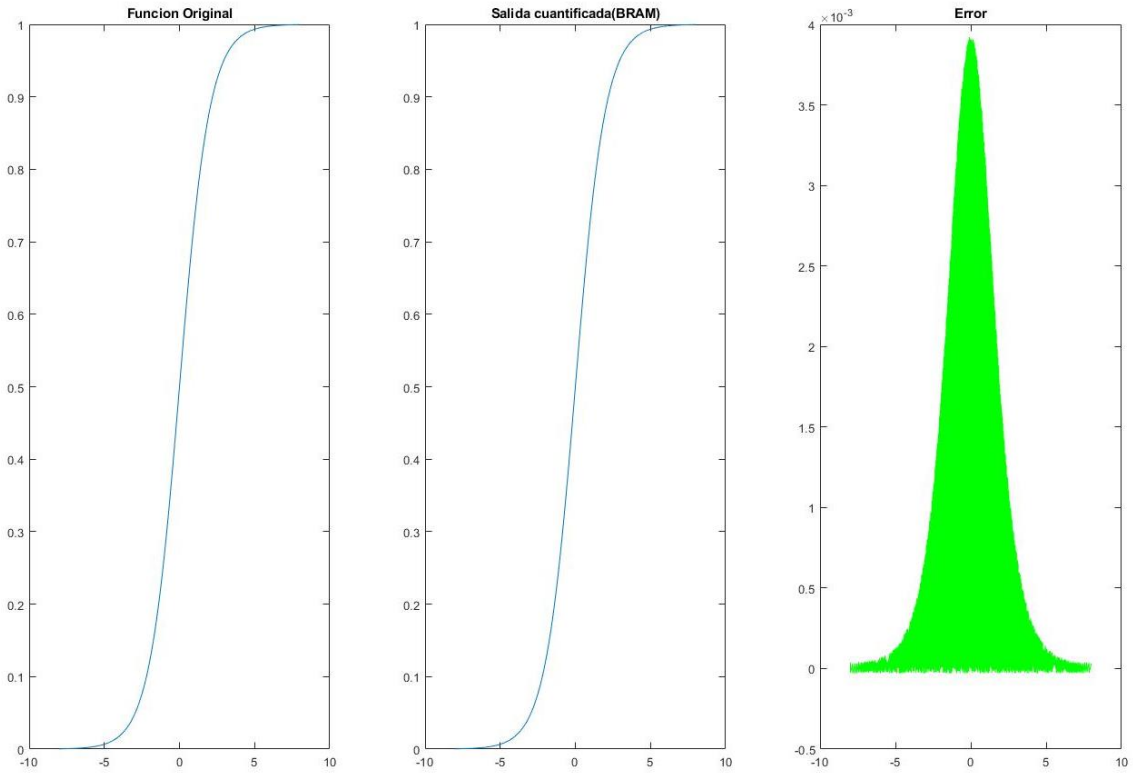

Figura 20. Comparativa función original-función cuantificada.

Este script es reconfigurable para el número de bits que se desee y permite reconfigurar la función de activación. Para este proyecto, tal y como se ha mencionado anteriormente, se han implementado tres scripts distintos para tres funciones de activación. Con la BRAM cargada, a la salida ya se tiene la salida de la neurona. Finalmente, en la Figura 21 se puede observar el bloque que se utiliza para implementar esta BRAM.

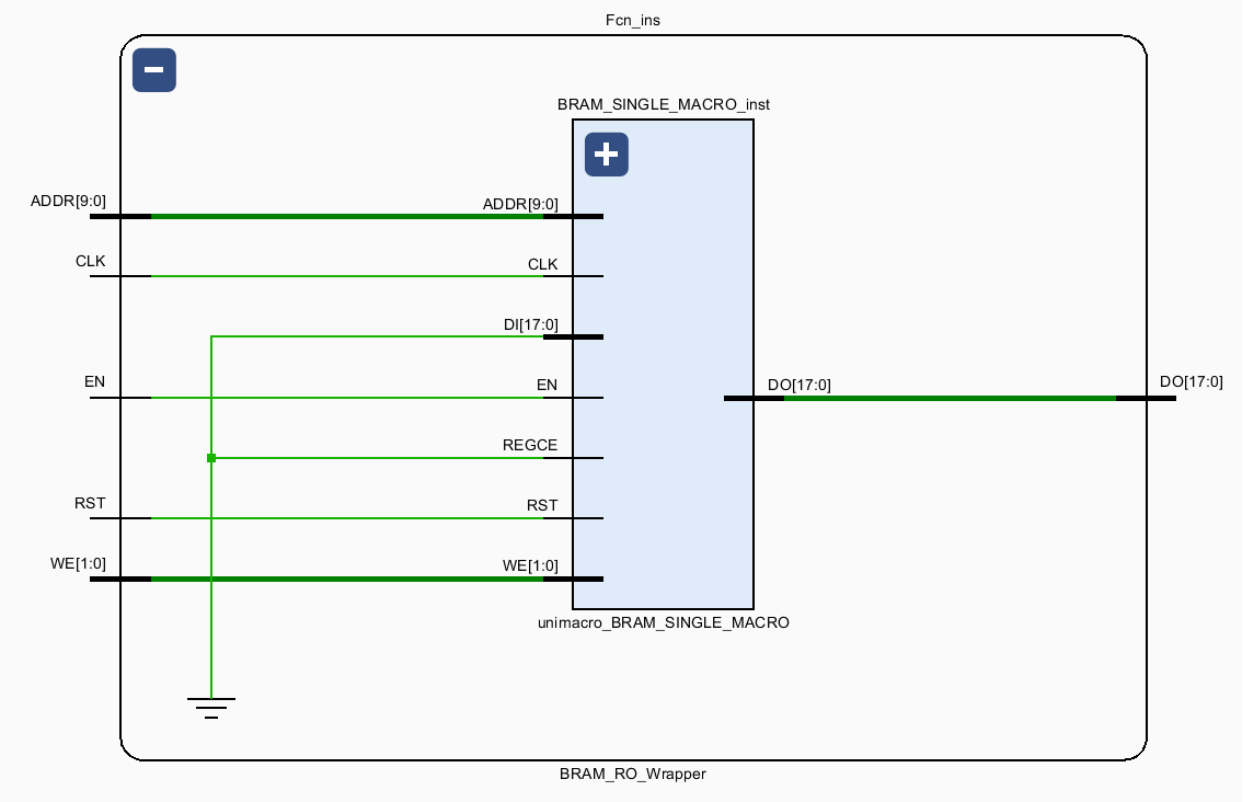

Figura 21. Bloque BRAM Read-Only.

3.1.7. Máquina de estados

Una vez se han definido los bloques principales de la neurona (multiplicador/acumulador y función de activación), hay que definir una máquina de estados que controle los bloques definidos y establezca la temporización de la neurona. En primer lugar, se muestra en la Figura 22 el diagrama de la máquina de estados y posteriormente se va a explicar brevemente el funcionamiento de ésta.

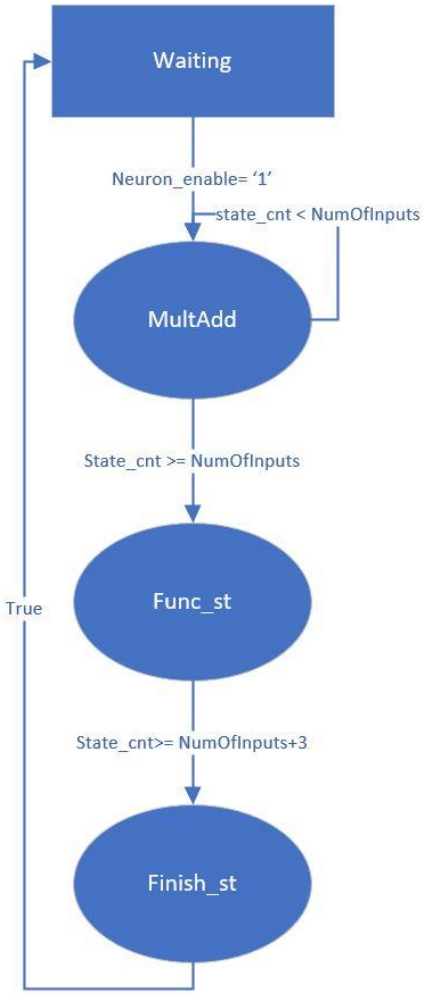

Figura 22. Máquina de estados de la neurona.

En el primer estado, llamado "Waiting", la neurona está esperando a ser activada; en este estado no se hace nada excepto esperar. La salida de este estado se produce cuando se activa la señal de habilitación de la neurona, con lo que se pasaría al estado "MultAdd". En esta transición se captura el valor de la entrada de la neurona (Input) y del peso (Weight) y se introducen al bloque acumulador, el cual es habilitado.

En el segundo estado llamado "MultAdd", con cada ciclo de reloj se introduce una nueva entrada y un nuevo peso a las entradas correspondientes del multiplicador/acumulador; la condición de salida de este estado es que el contador de entradas y pesos introducidos llegue al valor del genérico que indica el número de entradas de la neurona, se pasaría al estado "Func_st". En esta transición se deshabilita el multiplicador para que no siga multiplicando.

En el tercer estado llamado "Func_st", en primer lugar, se comprueba si el valor que sale del multiplicador/acumulador se sale de los rangos definidos para la función de activación (en el ejemplo actual, este rango es de -8 a 8). Si es así, satura a los límites, de todas maneras, recorta el valor de salida a 10 bits y lo introduce al bus de direcciones de la BRAM y la habilita. Pasados dos ciclos de reloj, la salida está disponible, por lo que en la transición de salida se coloca en la salida de la neurona el valor obtenido, se activa una salida de "Out_Valid" para indicar a la capa superior del diseño que la

salida ya es válida y se aprovecha para reiniciar el módulo de multiplicador/acumulador, se pasa a continuación al estado de "Finish_st".

En el cuarto y último estado, llamado "Finish_st", se baja la señal de "Out_Valid". Las únicas funciones de este estado es terminar de reiniciar el bloque multiplicador/acumulador y dar tiempo a la capa superior a bajar la señal de habilitación de la neurona. En el siguiente ciclo de reloj se vuelve a pasar al estado de "Waiting" y se puede volver a empezar el proceso. Como se puede observar el funcionamiento es muy simple y permite con un solo multiplicador realizar todas las multiplicaciones de las neuronas sin gastar muchos recursos.

3.2. Capa

La capa de una red neuronal es el siguiente paso en la arquitectura, una capa es un conjunto de neuronas que comparten las mismas entradas, una red neuronal se compone de varias capas, tal y como se muestra en la Figura 23.

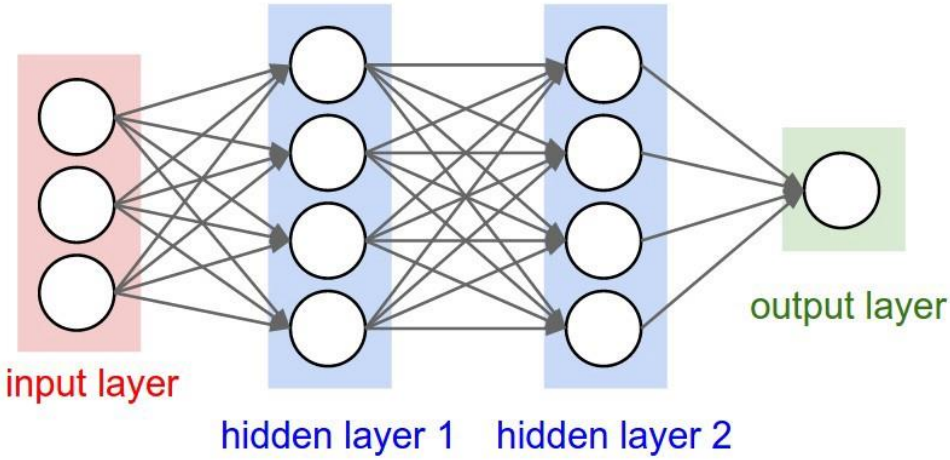

Figura 23. Estructura de una red genérica. Fuente: https://towardsdatascience.com/

Una vez se tiene definida cómo va a ser la arquitectura interna de la neurona, realizar una capa debería ser sencillo, sería introducir tantas neuronas como se necesiten, pero esto supondría por cada capa introducir *N* neuronas con *N* multiplicadores y *N* BRAMs para su función de transferencia, lo cual no es muy eficiente, ya que limitaría muchísimo el número de capas y el número de neuronas por capa.

Para solventar este problema, se plantea la posibilidad de instanciar una única neurona y que ésta pueda ser multiplexada, es decir, se plantea una doble multiplexación: por un lado, se está multiplexando las entradas y los pesos para usar un único multiplicador por cada neurona; y por otro lado, se plantea multiplexar las neuronas para poder usar una única neurona por capa, lo que abriría la posibilidad de tener más capas con más neuronas. Como en el caso anterior, se va a desgranar qué partes y bloques debe tener esta capa para poder hacer correctamente su función.

3.2.1. Entidad y parámetros genéricos

En este apartado se va a explicar las entradas/salidas y parámetros de la entidad, la cual se puede ver en la Figura 24.

```
entity Layer is
    generic
    (
        NumOfInputs     : integer :=32;
        NumOfNeurons    : integer :=32;
        Datalength      : integer :=18;
        DecimalLength   : integer :=14
    );
    Port
    (
        clk                 : in    STD_LOGIC;
        rst                 : in    STD_LOGIC;
        Layer_Enable        : in    STD_LOGIC;
        Weight_In           : in    STD_LOGIC_VECTOR(Datalength-1 downto 0);
        Load_Weight         : in    STD_LOGIC;
        Next_Input          : in    STD_LOGIC_VECTOR(Datalength-1 downto 0);
        Next_Input_Valid    : in    STD_LOGIC;
        Next_Output         : out   STD_LOGIC_VECTOR(Datalength-1 downto 0);
        Next_Output_Valid   : out   STD_LOGIC;
        Calc_End            : out   STD_LOGIC
    );
end Layer;
```

Figura 24. Entidad de la capa.

Como en el caso anterior, este módulo tiene varios genéricos:

- *NumOfInputs*: define el número de entradas de cada neurona de la capa.
- *NumOfNeurons*: define el número de neuronas de la capa.
- *Datalength*: ancho en bits de todas las representaciones numéricas (entradas, salidas y pesos), máximo 18.
- *DecimalLength*: número de bits dedicados a la parte decimal.

En cuanto a las entradas y salidas:

- **Clk**: señal de reloj del diseño.
- **Rst**: señal de reset activa a nivel alto.
- **Layer_enable**: señal de habilitación de la capa.
- **Weight_In**: bus por donde se cargan los pesos correspondientes a las neuronas de esa capa.
- **Load_Weight**: señal que indica cuando hay un peso válido en el bus de pesos.
- **Next_Input**: bus por donde se reciben las entradas del siguiente ciclo de ejecución.
- **Next_Input_Valid**: señal que indica que el dato en Next_Input es válido.
- **Next_Output**: bus de salida en el que se van extrayendo los resultados de las neuronas (se conecta con el Next_Input de la siguiente capa).
- **Next_Output_Valid**: señal que indica que el dato en Next_Output es válido (se conecta con el Next_Input_Valid de la siguiente capa).
- **Calc_End**: señal que indica que la capa se ha terminado de ejecutar.

3.2.2. Neurona

Por supuesto, para poder funcionar como capa, debe tener un bloque de neurona, solo necesita uno, por la razón que se ha mencionado anteriormente. Por ello, no se volverá a explicar nuevamente la neurona, se remite al lector al apartado anterior donde se da una explicación detallada de su funcionamiento interno.

3.2.3. Sistema de almacenamiento de pesos

Como se explicó anteriormente, la neurona no tiene memoria de sus pesos, este hecho viene motivado por la idea de que esa arquitectura no va a ser una única neurona, sino que va a ser multiplexada, por lo que no puede almacenar unos pesos que van a ir cambiando en función de la neurona que tenga que ser en cada momento, por lo que se hace necesario un sistema que permita almacenar esos pesos y poder usarlos cuando sea necesario.

Este almacenamiento, como en el caso de la función de activación de la neurona, es perfecto para utilizar una BRAM, pero en este caso no puede ser solo de escritura, por lo que habrá que utilizar otra macro. En este caso, además se va a utilizar una BRAM dual port, ya que estos pesos deben poder ser escritos desde un proceso y leídos desde otro, como se verá posteriormente. Por lo tanto, se busca una macro que permita que la memoria sea Dual Port, que la BRAM se configura para permitir su escritura y, además, se mantiene el tamaño de 18 kbit y el bus de datos de 18 bits. La macro utilizada es BRAM_TDP_MACRO, su esquema puede verse en la Figura 25.

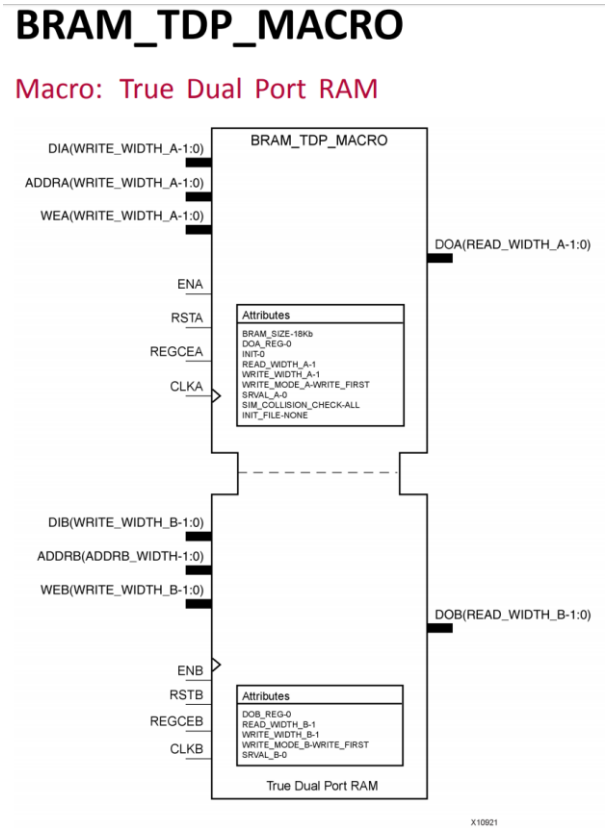

Figura 25. BRAM dual port lectura/escritura. Fuente: UG953-vivado-7series-libraries

Como se puede observar, esta macro tiene más atributos que la anterior, por lo que se proporciona una lista de los parámetros y sus valores elegidos:

- BRAM_SIZE: este atributo permite configurar una BRAM de 36kb o de 18 kb, en este caso, 18 kb.
- DOA_REG y DOB_REG: habilita un modo de escritura más rápido a costa de la latencia.
- READ_WIDTH_A y READ_WIDTH_B: ancho del bus de lectura, en este caso 18 bits.
- WRITE_WIDTH_A y WRITE_WIDTH_B: ancho del bus de escritura, en este caso 18 bits.
- WRITE_MODE_A: modo de escritura, en este caso, "WRITE_FIRST", arbitrariamente se ha elegido que el interfaz A es el de escritura.
- WRITE_MODE_B: modo de escritura, en este caso, "NO_CHANGE", ya que no se desea escribir desde el puerto B.
- INIT_xx: a continuación, en bloques de 256 bits hay que especificarle la inicialización.

En este caso, no es necesario especificar ninguna inicialización, ya que esta memoria tiene que ser escrita antes de la entrada en funcionamiento de la red neuronal. El mecanismo para su escritura se especifica en el siguiente apartado. Hay que observar que esta BRAM limita el número de entradas y o neuronas que puede tener la capa, ya que se tienen 1024 posiciones, por lo que el producto de nº de entradas x nº de neuronas no puede exceder 1024. El esquemático de esta arquitectura puede visualizarse en la Figura 26.

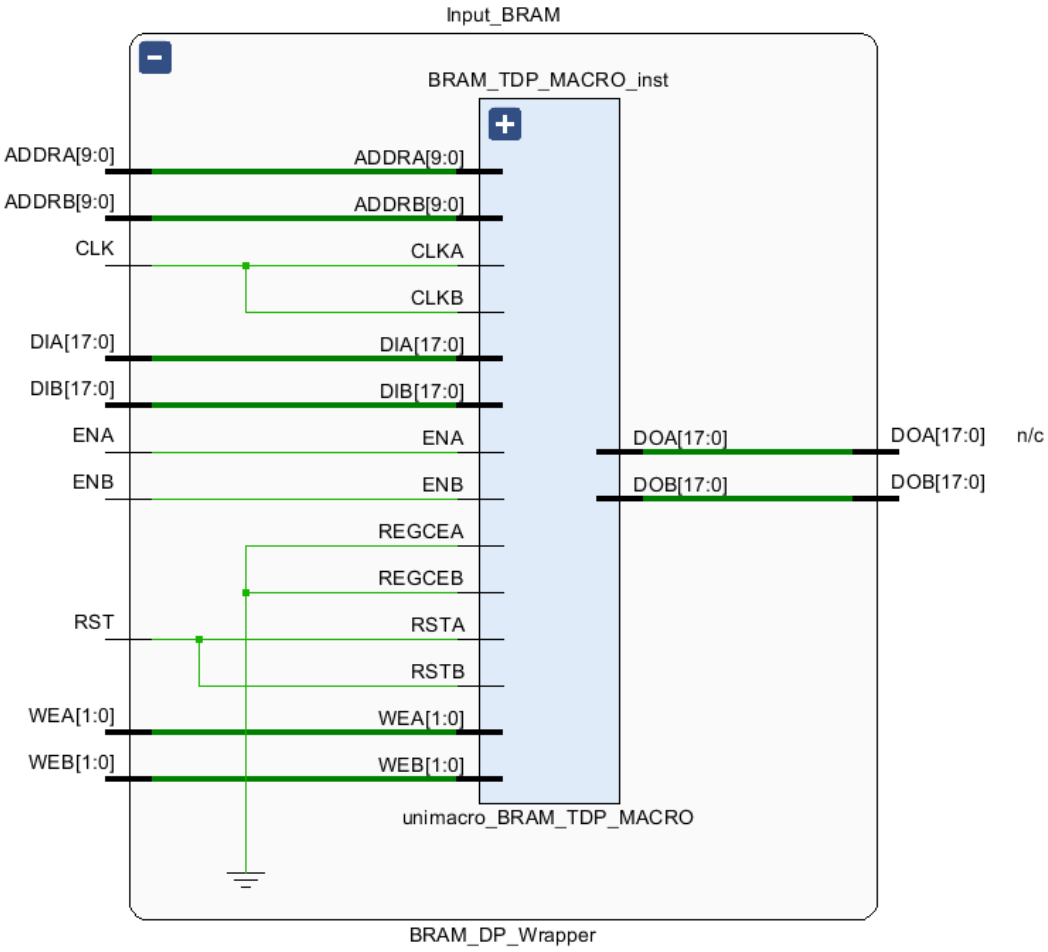

Figura 26. Esquemático de la arquitectura BRAM Dual Port.

3.2.4. Sistema de carga de pesos

Como es lógico, de alguna manera hay que cargar los pesos a la capa; esto se hace mediante la entrada "Weight_In" que tiene la capa y su señal de habilitación de carga de peso llamada "Load_Weight". El funcionamiento de este sistema de carga es muy parecido al de un puerto paralelo, donde el bus de datos sería "Load_Weight" y el reloj "Load_Weight". Por cada flanco de Load_Weight se carga un peso, el orden de carga sería los pesos de la neurona 1, a continuación, los pesos de la neurona 2, etc. Estos pesos se irían almacenando en la BRAM definida en el apartado anterior hasta llegar al número de pesos necesarios (nº de neuronas x nº de entradas). Otra opción de cargar los pesos es mantener "Load_Weight" a nivel alto y por cada flanco de reloj se cargaría un peso distinto (carga más rápida).

3.2.5. Sistema de almacenamiento de entradas

Este sistema es necesario porque normalmente, exceptuando la capa de entrada, las entradas a una capa son las salidas de las neuronas de la capa anterior, por lo que, si se están ejecutando todas las capas a la vez, las salidas de la capa anterior deben ser almacenadas en algún tipo de memoria, por lo que se vuelve a hacer necesario el uso de una BRAM para esta funcionalidad.

En este caso sí es estrictamente necesario que la BRAM sea dual port, porque, mientras se está leyendo la memoria para acceder a una entrada para ejecutar el ciclo actual, la capa anterior puede estar escribiendo una entrada para el ciclo siguiente. Esto plantea otro problema muy interesante: si la capa anterior se ejecuta más deprisa que la capa actual, puede llegar a pasar que se esté escribiendo la entrada del ciclo siguiente antes de que haya utilizado la entrada del ciclo actual, provocando errores en la ejecución.

Este problema se ha solucionado mediante un sistema de memoria denominado "Ping-Pong", que consiste en que el sistema de escritura usa unas posiciones de memoria distintas que el sistema de lectura, y en el ciclo siguiente, se intercambian las zonas de lectura/escritura. Con este sistema es imposible que se sobrescriba algo que hace falta en el ciclo actual. El esquema de este tipo de memorias puede verse en la Figura 27.

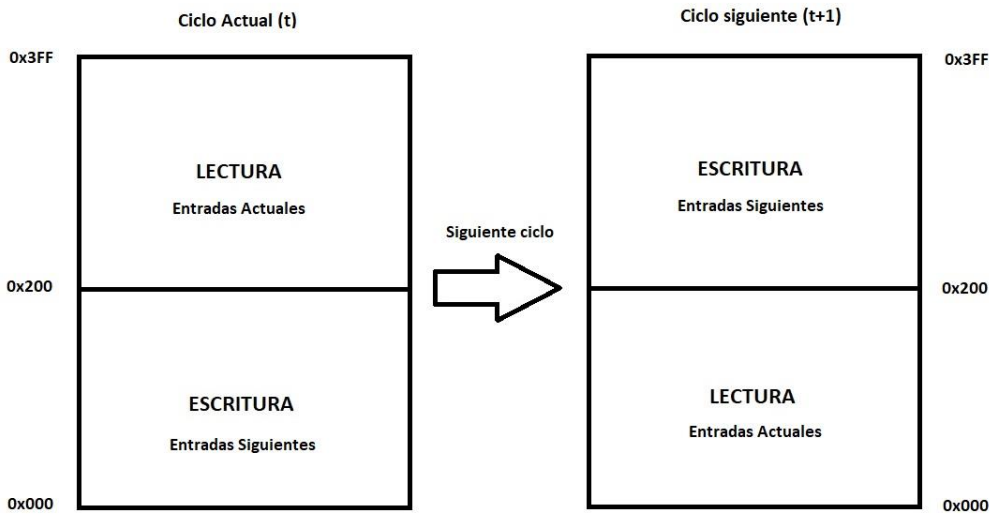

Figura 27. Memoria ping-pong.

Con este sistema, el número máximo de entradas que se puede llegar a tener es de 512, que es un número bastante elevado (no es limitante a no ser que solo tenga 2 neuronas de 512 entradas por capa).

3.2.6. Sistema de carga de entradas

Como en el caso de los pesos, es necesario un sistema de carga de las entradas, de hecho, es el mismo sistema. Con un proceso se atiende la señal de "Next_Input_Valid" y si ésta se activa se carga como siguiente entrada el valor de "Next_Input". En el caso de la capa de entrada estas señales deberán ser controladas por el sistema que alimente a la red neuronal y en el caso de las capas intermedias, estas señales estarán controladas por la capa anterior.

3.2.7. Máquina de estados

Una vez se han definido todos los bloques que integran la capa, es el momento de definir la máquina de estados que debe gobernar el funcionamiento de la capa. Igual que en el caso anterior, se presenta en primer lugar el esquema de la máquina de estados para posteriormente, explicar uno a uno los estados y cuáles son sus acciones. El esquema de la máquina de estados se puede ver en la Figura 28.

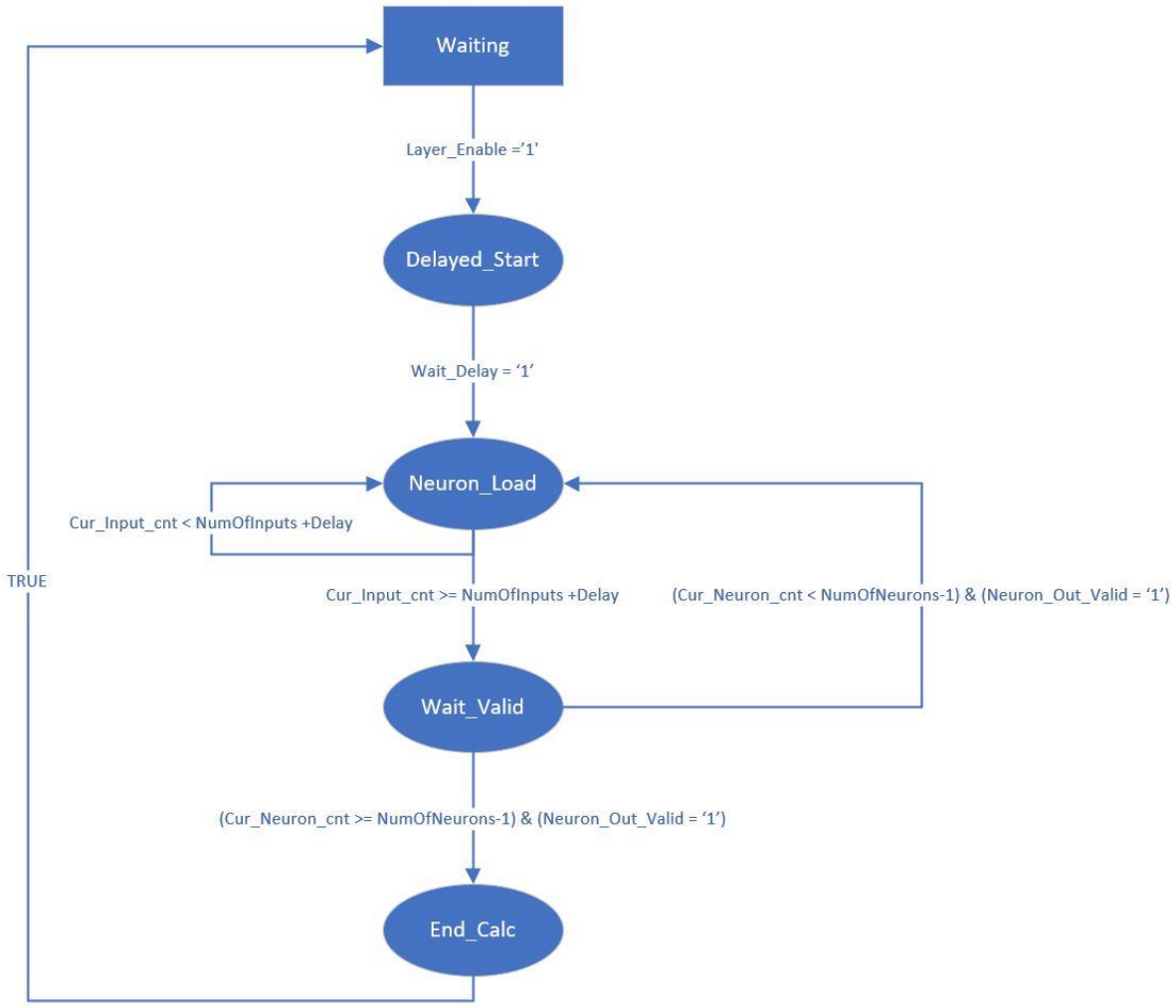

Figura 28. Máquina de estados de la capa.

Como se puede observar, esta máquina de estados es más complicada que la máquina de estados de la neurona, explicando brevemente su funcionamiento a continuación. El estado inicial de la máquina de estados es “Waiting”; en este estado, mientras no se habilite la capa, se encuentra revisando en qué estado se encuentra el sistema Ping Pong, ya que, en cada iteración de la capa, este estado debe cambiar. La condición de salida del estado es que se habilite la capa (Layer_Enable =’1’): cuando esto sucede, se direccionan las BRAM de pesos y entradas a la primera posición y se pasa al estado de “Delayed Start”.

El estado de “Delayed Start” es necesario para introducir dos ciclos de espera para que las memorias saquen los datos antes de poder metérselos a la neurona. Una vez ha pasado ese ciclo de espera, se pasa al estado “Neuron_Load”.

En el estado “Neuron_Load” se van cargando los pesos y las entradas en cada ciclo de reloj hasta que se cargan todos los pesos y las entradas correspondientes a una neurona; cuando esto sucede, se pasa al estado de “Wait_Valid”. Es importante observar el contador de entradas, que es el que se usa para saber si se han cargado todas las entradas de la neurona. Hay que tener en cuenta que hay que contar hasta 2 ciclos más, que son los que se han introducido de retardo en “Delayed Start”.

En el estado de “Wait_Valid” se espera a que la neurona termine de realizar los cálculos (la función de activación). Cuando esto sucede, se comprueba si la neurona actual es la última neurona, si no es así, se reinicia el contador de entradas, se suma 1 al contador de neuronas y se vuelve al estado de “Delayed Start”. Si era la última neurona (el contador de neuronas es igual al número de neuronas menos 1), se reinician los contadores de pesos, entradas y neuronas, se activa la señal de “Calc_End” para indicar a la capa superior que la capa se ha terminado de ejecutar y se pasa al estado de “End_Calc”.

En el estado de “End_Calc”, se desactiva la señal de “Calc_End” y se vuelve al estado de “Waiting”. Simplemente sirve para tener activa un ciclo de reloj la señal de “Calc_End” y dar tiempo a la capa superior a desactivar la señal de habilitación de la capa. Es importante observar que toda la máquina de estados es síncrona, es decir, todas las señales dependen de flancos de reloj y solo se cambian en estos flancos.

3.2.8. Bloque final

En este apartado, se presenta el bloque final que se utiliza para implementar una red neuronal, es decir, el bloque correspondiente a una capa. Este bloque se puede ver en la Figura 29.

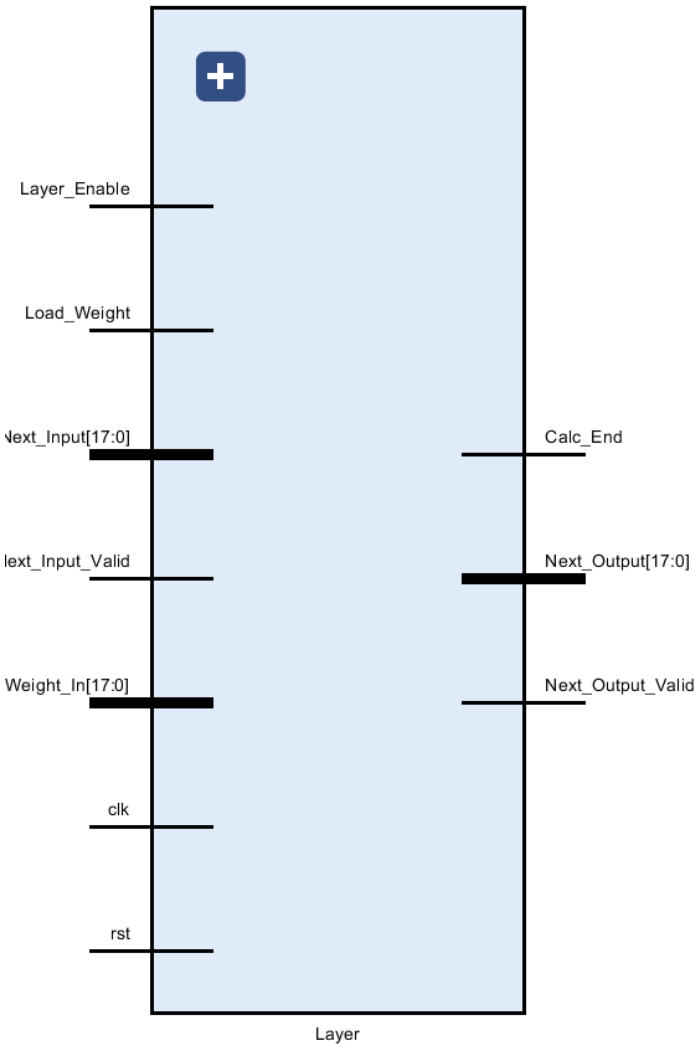

Figura 29. Bloque de Capa.

Con bloques de tipo capa, se pueden montar redes neuronales de tipo perceptrón multicapa, concatenando bloques del mismo tipo y ajustando sus genéricos a las necesidades específicas de cada red. En los se puede observar un ejemplo de cómo se puede hacer.

4. Resultados experimentales

En este apartado se van a exponer las simulaciones realizadas para cada uno de los bloques propuestos en la arquitectura, además de algunos montajes posibles que se pueden hacer con la arquitectura propuesta.

Cabe destacar que la herramienta utilizada para las simulaciones es el simulador embebido en Vivado 2017.4.

Se va a seguir la misma estructura que en el apartado anterior, se va a comenzar desde las unidades más simples hasta las unidades más complejas.

4.1. Multiplicador/Acumulador (MACADDR)

Como no podía ser de otra manera, se comienza con los elementos que componen la neurona, en primer lugar, se valida el funcionamiento del multiplicador/Acumulador. Para ello, se ha preparado un TestBench en el que se introducen varios valores en sus entradas y se comprueba que su salida coincide con el resultado que devuelve una multiplicadora. Para el funcionamiento de este TestBench se ha generado un reloj de 100 MHz, que previsiblemente será la frecuencia de funcionamiento final del diseño, como se verá más adelante.

En la primera captura, mostrada en la Figura 30, se puede observar el comportamiento del bloque cuando está en estado de reset, la salida se mantiene a cero.

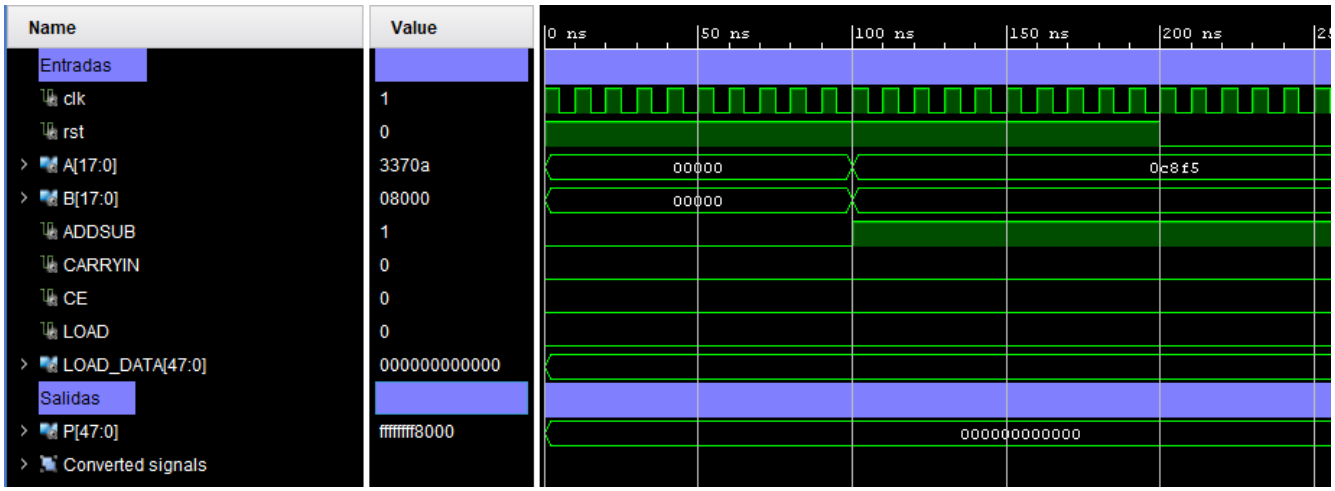

Figura 30. Estado de reset MACADDR.

En la siguiente captura de la Figura 31 se muestra cómo se introducen dos valores y posteriormente otros dos, comprobando que en la salida se devuelve el resultado correctamente.

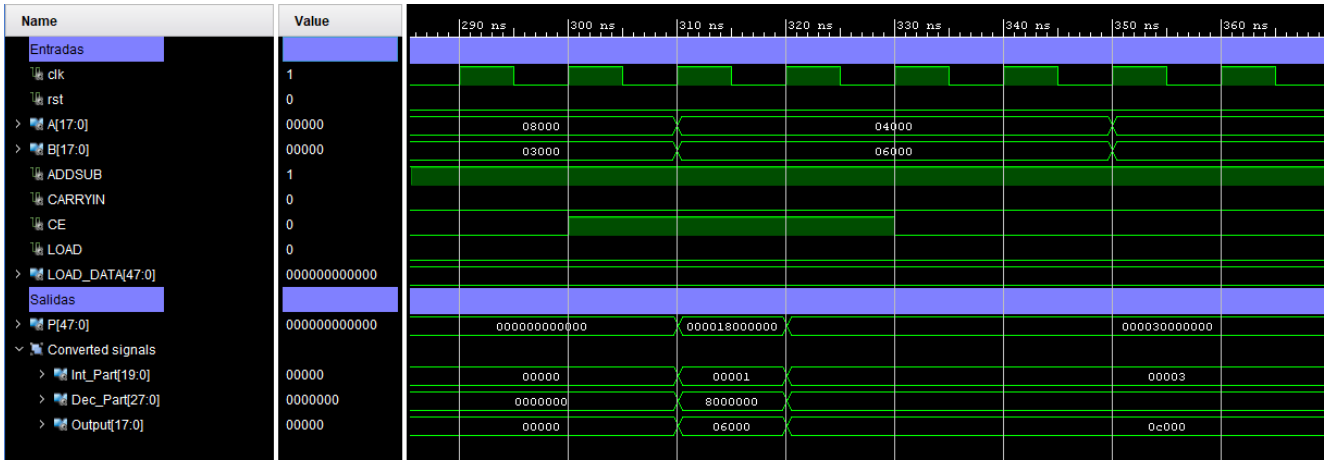

Figura 31. Comprobación 38ultiplicación+Acum MACADDR.

Los datos introducidos son:

- 0x8000 x 0x3000 = 0x18000000
- 0x18000000+(0x4000 x 0x6000) = 0x30000000

Comprobando con una calculadora hexadecimal, este valor es correcto. De esta simulación se pueden extraer más conclusiones interesantes: por ejemplo, el módulo es capaz de sacar una salida por cada ciclo de reloj, pero siempre va desfasado un ciclo. Si se observa el ejemplo de la red neuronal, se va a suponer que A y B son peso y entrada, estos pesos y entradas tienen 18 bits de ancho; además, la cuantificación se ha realizado en el rango de -8 a 8 y se ha escogido el mínimo número de bits necesario para poder representar -8, que son 4 bits de parte entera y 14 de parte decimal. Con esta configuración se tiene una precisión de $6.1035 \cdot 10^{-5}$.

Se puede transformar los valores A y B a valores decimales para ver si la cuenta se ha realizado correctamente.

- A1 = 0x8000 = 2.
- B1 = 0x3000 = 0.75.
- A2 = 0x4000 = 1.
- B2 = 0x6000 = 1.5.

Si se realizan los cálculos, el resultado sería 3. Para poder llegar a ese resultado, hay que tener en cuenta que esto es aritmética en coma fija, por lo que hay que realizar una serie de descomposiciones. Puesto que hemos elegido 14 bits de parte decimal y 4 de parte entera, al multiplicar se duplica el número de bits de cada tipo, por lo que la estructura de la salida del multiplicador acumulador sería como sigue:

- Bits del 27 a 0: parte decimal.
- Bits del 47 al 28: parte entera.

Si se quiere obtener un resultado en 18 bits (recuérdese que debe almacenarse en una BRAM de 18 bits de bus de datos), se necesita eliminar los 14 bits de menor peso y de la parte entera, quedando con los 4 bits de menor peso. Este procedimiento es precisamente el que se ha seguido en la parte de "Converted signals" de la simulación. Se tiene por un lado los bits de la parte entera en la señal que se llama "Int_Part" y en "Dec_Part" la parte decimal. Finalmente se monta la señal de 18 bits en "Output", que es 0xC000, que pasando el cuantificador al revés devuelve el valor de 3. Es importante

que a la salida el bloque que está por encima (la neurona), haga una comprobación de que la parte entera de este número no se salga de ±8.

Para completar este set de simulaciones simples, se comprueba si la precarga funciona. Para ello, como solo se va a utilizar la precarga a 0 (cada vez que se quiera reiniciar la cuenta), en la Figura 32 se puede ver este procedimiento.

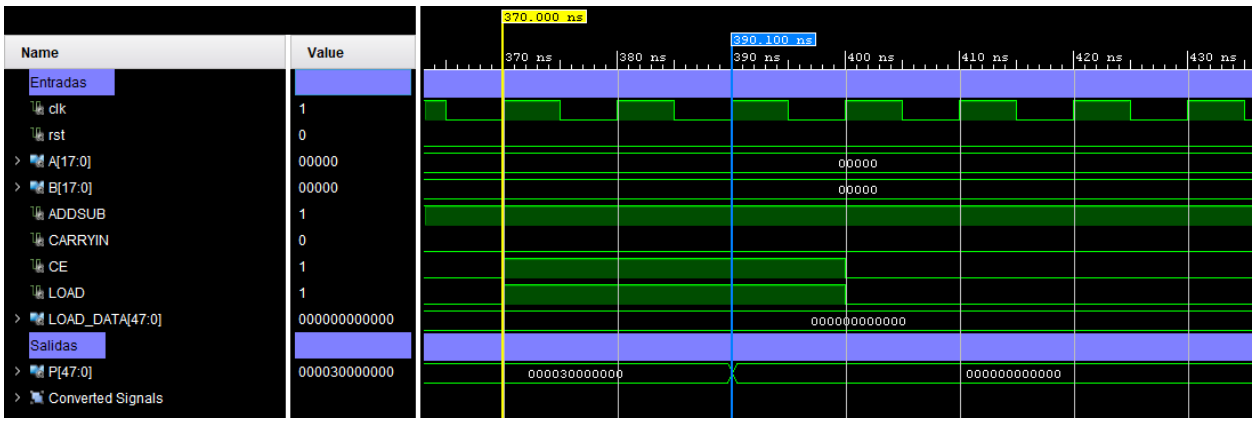

Figura 32. Carga de MACADDR.

Como se puede observar, este procedimiento tarda dos ciclos de reloj en hacerse.

4.2. Memoria BRAM de solo lectura (función de activación)

En este apartado se revisa que la memoria está funcionando correctamente, que está cargada con los valores elegidos y los tiempos que maneja desde que se solicita una lectura hasta que la memoria los dispone. Como en el caso anterior, lo primero es comprobar el comportamiento tras un reset, esto se puede ver en la Figura 33.

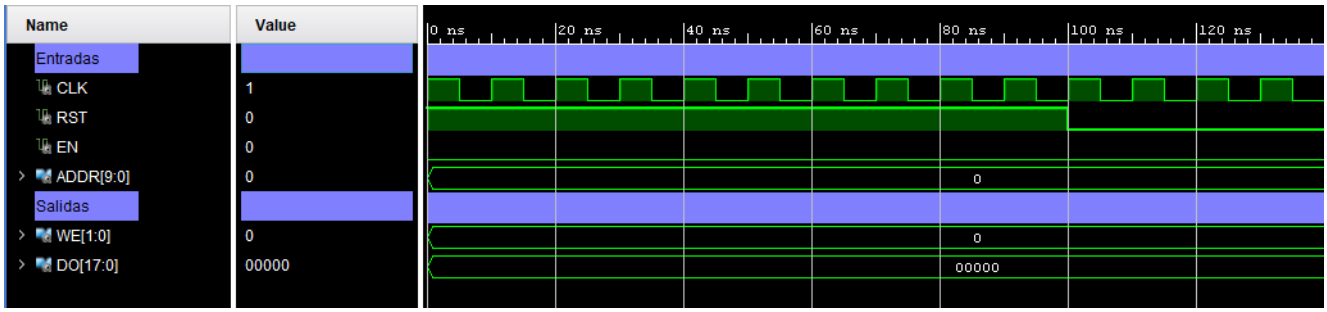

Figura 33.Estado de Reset BRAM solo lectura.

Como se puede observar, la salida de la memoria cuando se aplica el reset es de 0, por lo que funciona correctamente y se observa que no hay nada que esté en “Undefined” (sin definir).

Para la realización de esta prueba completa, se ha creado un proceso en el testbench mediante un contador ascendente de 0 a 1023 para recorrer todas las posiciones de memoria (recuerde que el bus de direcciones es de 10 bits→ 1024 posiciones). Además, se ha preparado un sistema de depuración que va escribiendo en un archivo de texto la salida, para posteriormente compararlo con la salida directa de Matlab.

La simulación es larga, por lo que se expone un fragmento de la misma, de hecho, de la parte central, que es donde el cambio es más apreciable, ya que la función sigmoidal tiene una pendiente mayor en esta zona. Esta simulación se puede ver en la Figura 34.

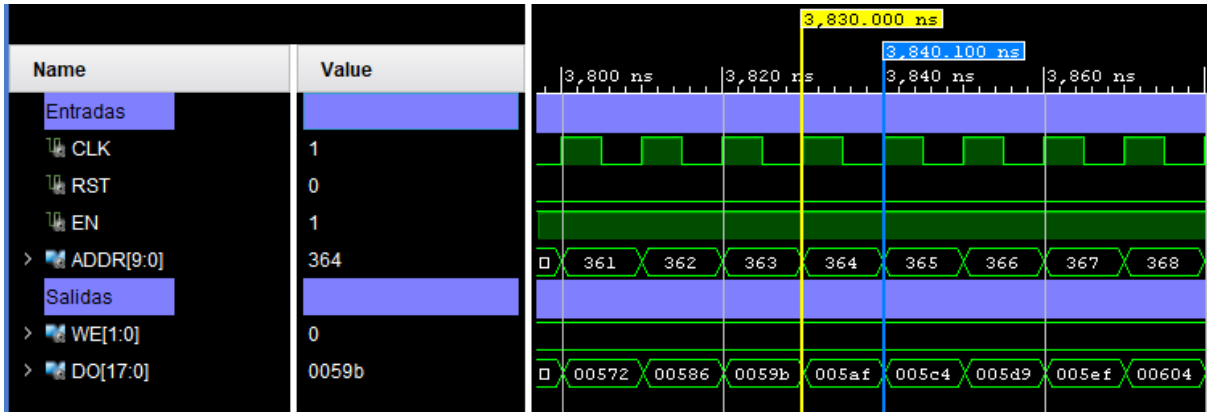

Figura 34. Lectura de BRAM de función de activación.

Es importante observar que la memoria tiene una latencia de 1 ciclo de reloj, por lo que habrá que tenerlo en cuenta cuando se utilice. Finalmente, en la Figura 35, se puede comprobar una comparación entre lo sacado por la simulación y lo que Matlab saca para que se introduzca como inicialización de la BRAM.

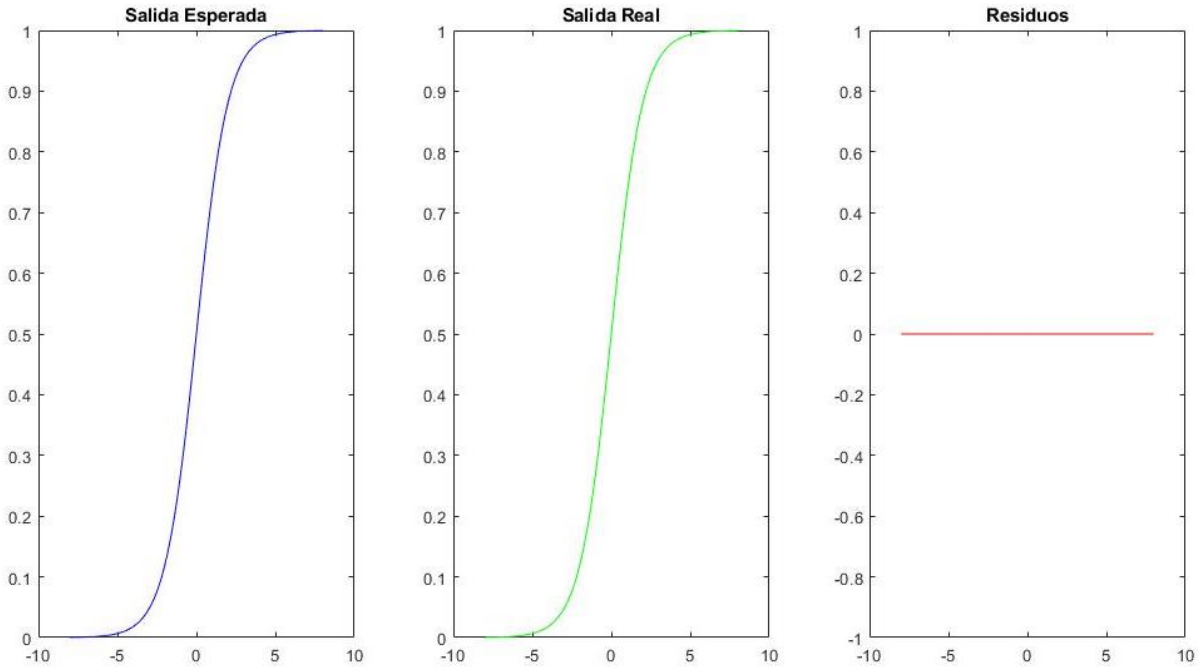

Figura 35. Comprobación de la salida BRAM solo lectura

Como se puede observar, los residuos son nulos, por lo que se considera validada la memoria de solo lectura.

4.3. Neurona

En este apartado se proporcionan todas las pruebas de validación que se le han hecho a la neurona, ya que en anteriores apartados se han realizado las validaciones a los distintos bloques que la componen. Como en los anteriores apartados, la primera comprobación es que el estado de reset funciona correctamente; para ello, se proporciona la simulación en la Figura 36.

Figura 36. Estado de reset del bloque Neurona.

En la simulación se observan tres grandes grupos de señales: entradas, salidas y señales internas. Las entradas y las salidas son las entradas y las salidas del bloque, es decir, las que se ven desde las capas superiores; las señales internas son las que modelan el funcionamiento interno de la neurona. En cuanto al estado de reset, simplemente se puede decir que la salida se mantiene a 0 y la señal de Out_Valid no está activa, todas las variables internas están a 0.

En el llamado “Funcionamiento Normal”, la neurona va a recibir una serie de pesos y entradas, los cuales tiene que multiplicar y acumular; cuando haya multiplicado y acumulado todos los pesos, debe aplicarles la función de activación. Para una prueba sencilla, mostrada en la Figura 37, se le va a introducir a la neurona un total de tres entradas con sus respectivos pesos y comprobar que la salida coincide con lo esperado.

Figura 37. Funcionamiento normal del bloque Neurona.

En primer lugar, los pesos y entradas que se están introduciendo son respectivamente los valores que toman “Weight” e “Input”, que son los siguientes:

- Entrada1= 0x08000 (2) Peso1 = 0x0C8F5 (3.14)
- Entrada2= 0x06000 (1.5) Peso2 = 0x0151F (0.33)
- Entrada3= 0x02000 (0.5) Peso3 = 0x04000 (1)

La señal State muestra el estado interno de la Neurona, que sigue el comportamiento definido en el correspondiente apartado de la arquitectura propuesta. Las señales dentro del grupo “Mul/Acum” permiten observar el comportamiento del multiplicador acumulador y, finalmente, las señales de la función de activación permiten ver la dirección de la memoria a la que se está apuntando.

Si se observa la simulación, la salida de la neurona está disponible 6 ciclos después de su activación, por lo que el tiempo de proceso siempre es nº de entradas+3. La salida de esta simulación es 0x3FF5, que pasándolo por el cuantificador de Matlab devuelve un valor de 0.9993, que es exactamente el resultado de logsig(2·3.14 + 1.5·0.33 + 0.5·1).

Uno de los problemas que puede aparecer usando la arquitectura actual es que si el resultado de multiplicar y acumular excede el rango elegido, podría suceder que el resultado sea erróneo. Éste era el motivo por el que el bloque multiplicador/acumulador tiene 48 bits de ancho cuando en principio con 36 (ancho de bits del dato x 2) debería valer. Este ancho extra de bits permite comprobar si se ha salido de rango y saturar a los valores máximo y mínimo. En la Figura 38 se puede ver el ejemplo para una saturación positiva, es decir, por encima de 8 (en el caso de la función sigmoidal).

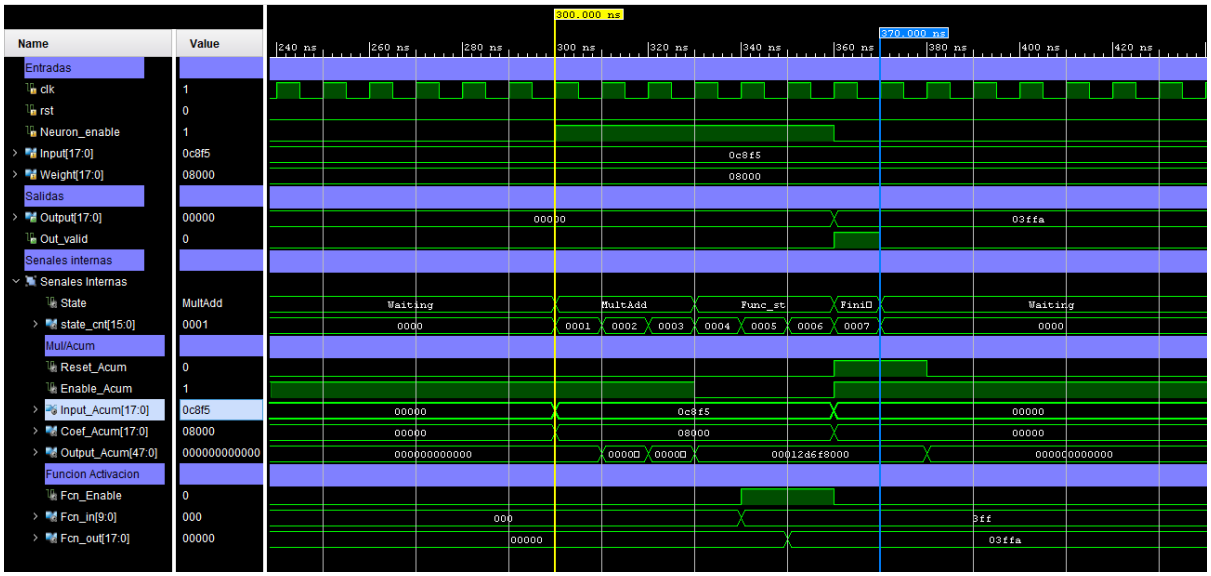

Figura 38. Saturación positiva del bloque Neurona.

En este caso, aparece la misma neurona que antes (tres entradas), pero en esta simulación las tres entradas y los tres pesos valen lo mismo:

- Entrada = 0x0C8F5 (3.14) Peso = 0x08000 (2)

Haciendo los cálculos, a la salida del acumulador se tiene:

$$Output_{Acum} = 3.14 \cdot 2 \cdot 3 = 18.84$$

Como se puede ver, está lejos del límite superior de la función de activación; esto es detectado por la lógica interna e introduce la máxima dirección de la memoria, para que devuelva el valor correspondiente a 8, que en nuestro caso es 0x03FFA (0.9996).

Otro de los problemas es que el resultado de multiplicar y acumular exceda el rango elegido negativamente, podría suceder que el resultado sea erróneo al igual que antes. En la Figura 39 se puede ver el resultado si el resultado excede -8.

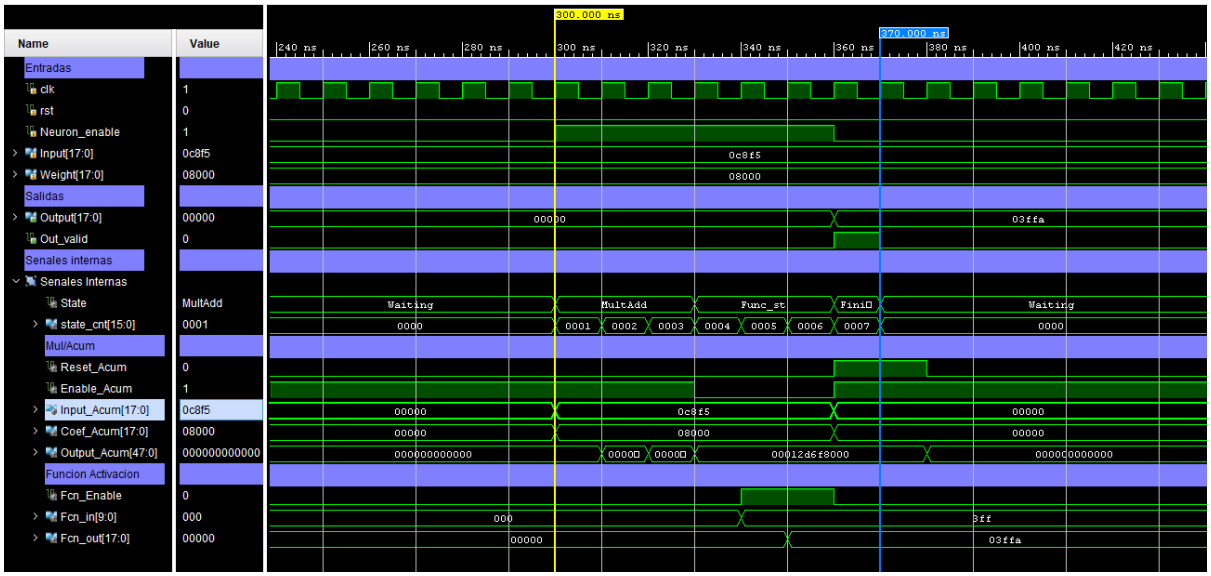

Figura 39. Saturación negativa del bloque Neurona.

En este caso, es la misma neurona que antes (tres entradas), pero en esta simulación las tres entradas y los tres pesos valen lo mismo:

- Entrada = 0x3370A (-3.14) Peso = 0x08000 (2)

Y a la salida del acumulador se tiene:

$$Output_{Acum} = -3.14 \cdot 2 \cdot 3 = -18.84$$

Como se puede ver, está lejos del límite inferior de la función de activación, esto es detectado por la lógica interna e introduce la primera dirección de la memoria, para que devuelva el valor correspondiente a -8, que en este caso es 0x00005 ($3.0518 \cdot 10^{-4}$).

4.4. Memoria BRAM de lectura y escritura Dual Port (entradas y pesos)

En este apartado se va a mostrar la validación de un bloque BRAM de lectura y escritura, validando ambos puertos con una prueba de funcionamiento parecida a la que se va a utilizar posteriormente. Este bloque es el que se utiliza para dotar de memoria de las entradas y los pesos a los bloques de tipo capa. Al igual que en todos los bloques anteriores, en la Figura 40 se muestra el estado de todas las señales cuando se activa la señal de reset.

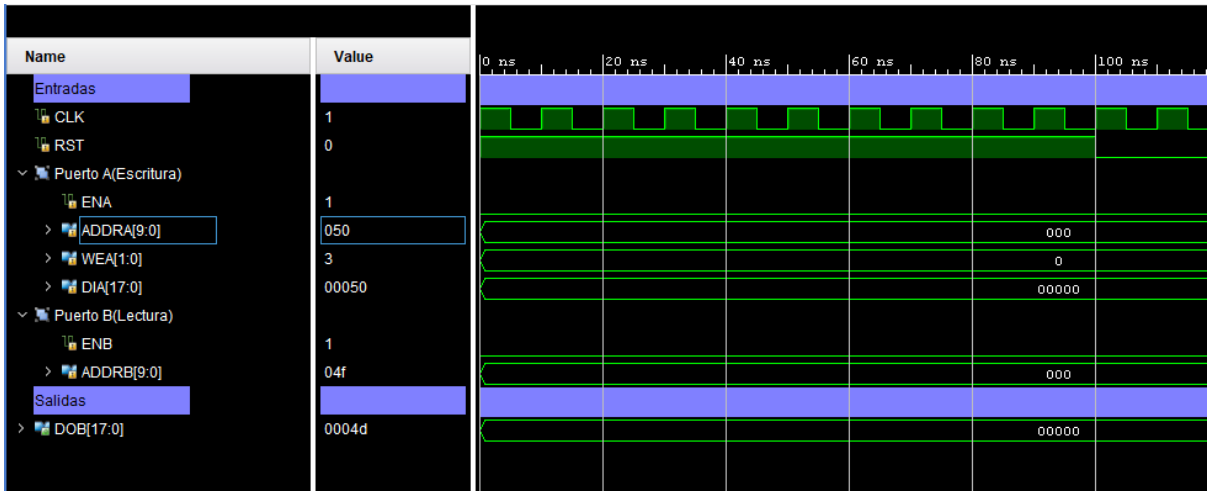

Figura 40. Estado de reset BRAM Dual Port.

Como se puede observar el estado de todas las señales de salida es cero. Se han omitido las señales irrelevantes como la señal de entrada del puerto B (ya que no se utiliza) y la señal de salida del puerto A (ya que tampoco se utiliza).

Para esta prueba, se ha ideado un sistema en el que se direcciona con un contador ascendente desde el puerto A y se escribe el valor de ese contador desde ese mismo puerto. Al mismo tiempo, desde el puerto B se lee el valor que haya en la posición anterior de la memoria (es decir, el contador menos 1). El resultado de esta simulación se puede ver en la Figura 41.

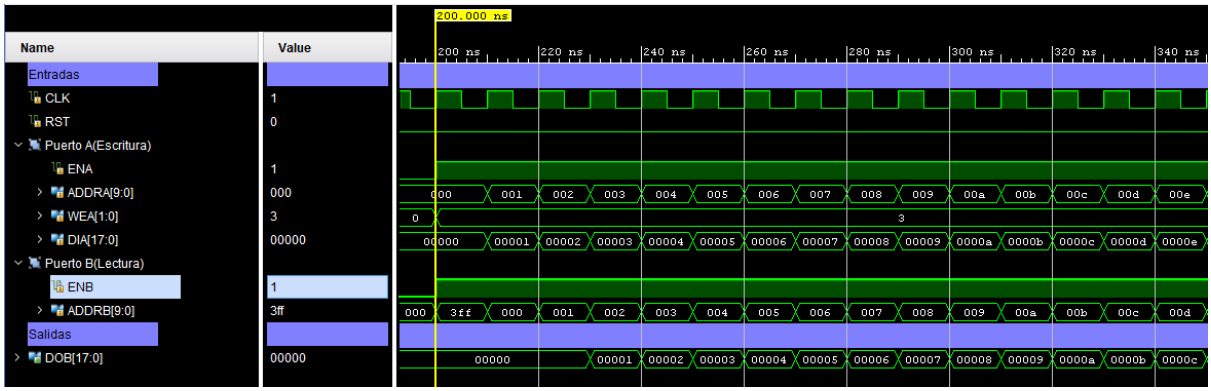

Figura 41. Prueba de lectura/escritura de la BRAM Dual Port.

Como se puede observar, el tiempo necesario tanto como para leer como para escribir en la BRAM es de un ciclo de reloj (esto ya se adelantó al hablar de esta BRAM en el apartado de la Arquitectura). También es importante apreciar que la salida del puerto B va desfasada un ciclo, la escritura se ha hecho correctamente, pero tal y como se ha comentado, el tiempo de lectura es de 1 ciclo.

4.5. Capa (Layer)

Una vez validados todos los bloques necesarios para poder conformar una capa, ha llegado el momento de validar la misma; para ello, se proponen una serie de simulaciones que van a permitir comprobar todas las funcionalidades descritas en el apartado de Arquitectura propuesta.

Como viene siendo habitual en estos apartados, la primera prueba es el comportamiento en reset. En la Figura 42 se puede comprobar cuál es el estado de las salidas cuando se provoca un reset.

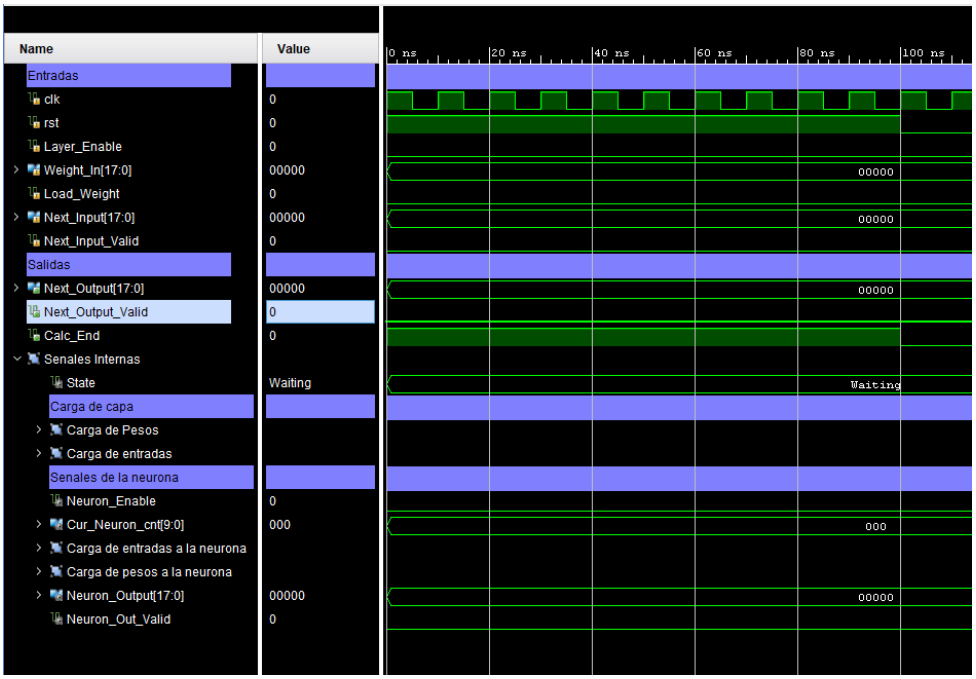

Figura 42. Estado de reset del bloque Capa.

En este nivel de jerarquía se tiene muchas más señales, por lo que se ha decidido crear varios grupos de señales y las no relevantes se van a ocultar por ahora. En lo que nos ocupa, que es el reset, solo es reseñable que la señal Calc_End se mantiene a uno, para indicar a la capa superior que esta capa ha

finalizado sus cálculos y está lista para volver a comenzar unos nuevos, es decir, se ha reiniciado todo su estado interno. La capa comienza en el estado de Waiting (como es lógico).

En este apartado se va a realizar una simulación de cómo se cargarían los pesos a la capa; esta carga de pesos se puede realizar en cualquier momento, pero es aconsejable no realizarlo durante la ejecución de la capa. Por lo demás, se puede realizar cuantas veces se quiera. La simulación se puede ver en la Figura 43.

Figura 43. Carga de pesos del bloque Capa.

Como se puede observar en la parte de las entradas, la carga es muy simple, se coloca el peso a cargar en la capa y se activa la señal Load_Weight; al activarse esta señal, el mecanismo de carga interno va contando ascendentemente con un contador hasta que se carga completamente la capa.

Se puede cargar un peso por cada ciclo de reloj: se puede ver que se han cargado seis pesos en la prueba actual, se ha configurado la capa para que acoja dos neuronas de tres entradas cada una y para comprobar que todo está funcionando correctamente, se van a cargar los mismos pesos y las mismas entradas a las dos neuronas para comprobar que ambas devuelven el mismo resultado.

En este apartado se va a aportar una simulación del sistema de carga de las entradas. Este sistema se utiliza tanto en el caso de que la capa sea la de entrada, en cuyo caso, la capa superior se debe ocupar de cargar estas entradas, como si es una capa intermedia o de salida, en cuyo caso, esta alimentación se hace de manera automática desde la capa anterior. En la Figura 44 se puede ver un ejemplo de la carga.

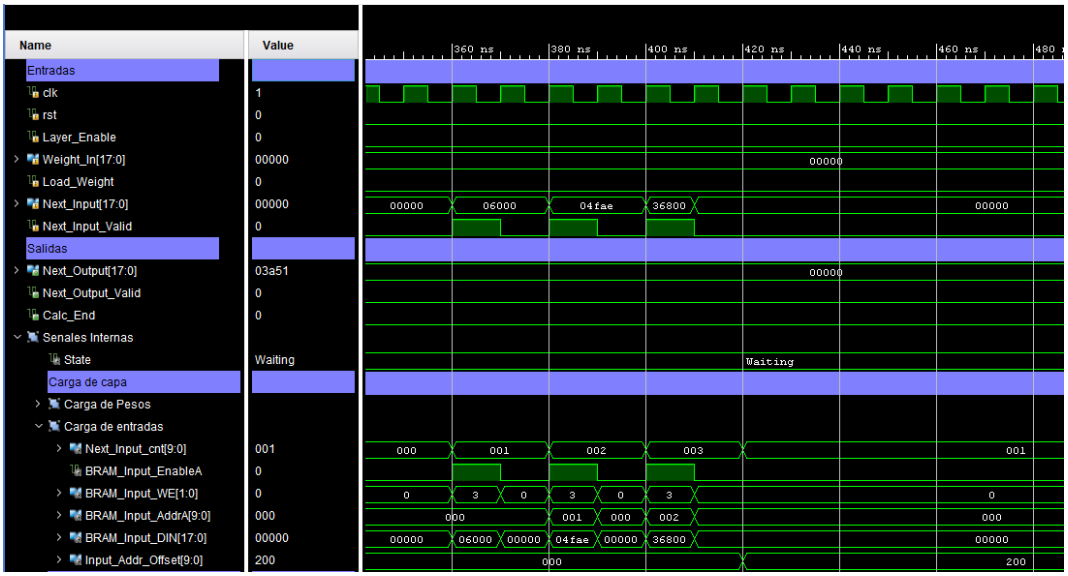

Figura 44. Carga de entradas del bloque Capa.

En este caso se ha elegido otro sistema de carga basado en pulsos (mucho más parecido a lo que va a suceder entre capas), aunque también podría haberse cargado de forma continuada como los pesos, pero así se tiene un ejemplo de los dos procedimientos. El funcionamiento es el mismo, por cada ciclo de reloj que se tenga a nivel alto Next_Input_Valid, se cargará una entrada, existe un contador que permite saber en qué posición va la siguiente entrada. Lo más remarcable de esta parte es la señal Input_Addr_Offset, que permite gestionar correctamente el buffer Ping-Pong, ya que cuando se terminan de cargar las entradas se cambia el offset para empezar en la otra mitad de la memoria.

A continuación se proporciona una simulación de cómo sería la ejecución de la capa. Para ello, se va a presentar todos los parámetros de la misma:

- 2 neuronas
- 3 entradas

En cuanto a los pesos, las dos neuronas tienen los mismos pesos para facilitar el seguimiento de la simulación:

- Peso 1: 0x0151F (0.33)
- Peso 2: 0x04666 (1.1)
- Peso 3: 0x3F333 (-0.2)

En cuanto a las entradas:

- Entrada 1: 0x06000 (1.5)
- Entrada 2: 0x04FAE (1.245)
- Entrada 3: 0x36800 (-2.375)

El resto de los parámetros se mantienen como en el resto de las simulaciones, 18 bits de ancho de datos con 4 bits para la parte entera y la función sigmoidal. La simulación realizada puede verse en la Figura 45.

Figura 45. Ejecución del bloque Capa

Se han colocado marcadores para facilitar encontrar los puntos clave. En el primer marcador (amarillo) comienza la ejecución de la capa, se preparan todos los datos que se habían almacenado previamente y se carga la neurona con ellos. En el segundo marcador (azul), la neurona ha marcado que ha terminado el cálculo y éste se saca hacia el exterior a través de Next_Output, en este caso vale 0x03A51. Sin detenerse, se carga la neurona con las siguientes entradas y pesos y en el tercer marcador (azul) la neurona marca que tiene la salida lista y se saca al exterior; en este caso, igual que en el anterior, la salida es 0x03A51.

Introduciendo a Matlab la fórmula de la neurona:

$$Salida_{esperada} = logsig\left((0.33 \cdot 1.5) + (1.1 \cdot 1.245) + \left(-0.2 \cdot (-2.375)\right)\right)$$
$$= logsig(5.191215625) = 0.9121$$

Y deshaciendo el cuantificador de la salida de la neurona sale:

$$Salida_{real} = hex2num(Qout, 3ª51) = 0.9112$$

Como se puede ver, hay error en esta salida, esto es debido en su mayor parte a que los pesos y las entradas se han escogido de manera aleatoria y al cuantificar se pierde parte de la precisión. En el siguiente apartado se va a realizar una prueba con entradas y pesos distintos para las neuronas de la misma capa, además se introducirá más de una capa.

4.6. Red Multicapa (Layer)

En este apartado se va a realizar una pequeña demostración de que la arquitectura funciona, para ello se proponen una serie de simulaciones. En esta simulación se busca la simpleza, se han colocado dos capas de dos neuronas cada una, las dos capas tienen el mismo número de entradas, el número de entradas de la segunda capa viene impuesto por el número de neuronas de la primera, por lo que ambas capas tendrán 2 entradas.

A continuación, se muestran los pesos de cada neurona:

- Capa 1
  - Neurona 1:
    - Peso 1: 0x0151F (0.33)
    - Peso 2: 0x04666 (1.1)
  - Neurona 2:
    - Peso 1: 0x3F333 (-0.2)
    - Peso 2: 0x3F99A (-0.1)
- Capa 2
  - Neurona 1:
    - Peso 1: 0x04000 (1)
    - Peso 2: 0x3ECCD (-0.3)
  - Neurona 2:
    - Peso 1: 0x03333 (0.33)
    - Peso 2: 0x0C000 (3)

En la Figura 46 se puede observar cómo se han cargado estos pesos en cada una de las capas.

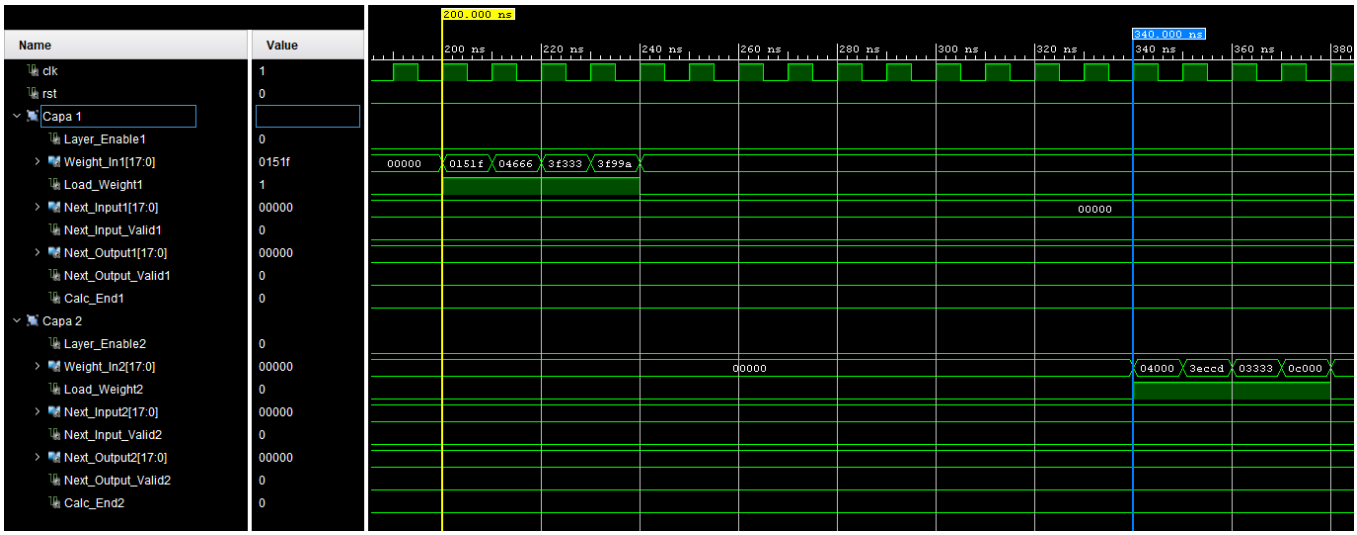

Figura 46. Carga de pesos en la red neuronal bicapa.

Se han dejado 100 ns entre una carga y otra, pero no son necesarios, se ha hecho para mayor claridad de la simulación. A continuación, solo se van a cargar las entradas de la primera capa, en la segunda capa, las entradas iniciales son cero, ya que no ha habido inicialización ninguna ni la capa anterior se ha ejecutado aún.

Las entradas a introducir son:

- Capa 1
  - Entrada 1: 0x04FAE (1.245)
  - Entrada 2: 0x36800 (-2.375)
- Capa 2
  - Entrada 1: 0x00000 (0)
  - Entrada 2: 0x00000 (0)

La carga de estas entradas se puede ver en la Figura 47.

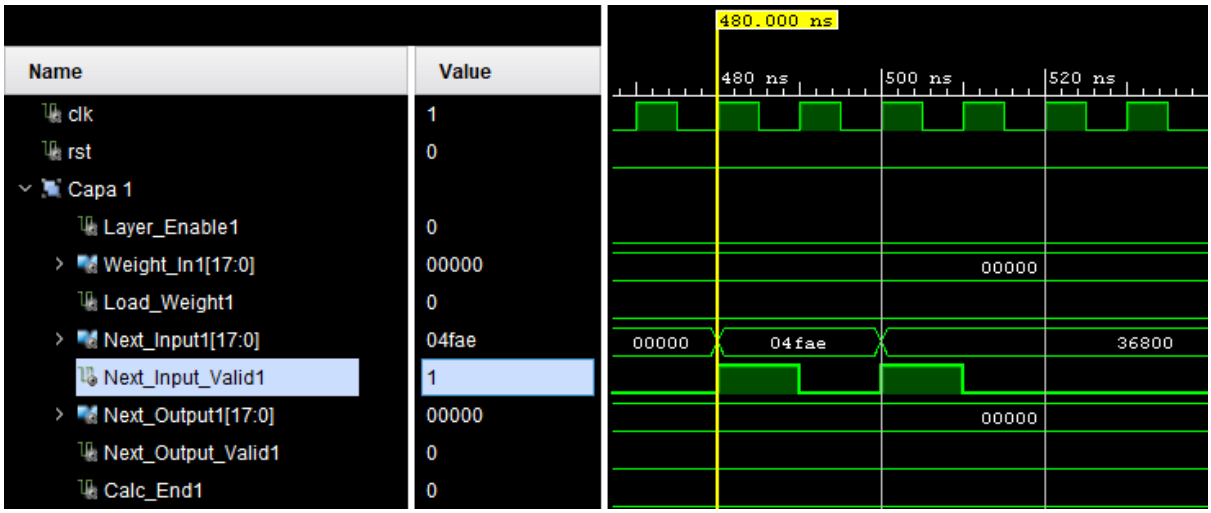

Figura 47. Carga de entradas de la red neuronal bicapa

A continuación, teniendo ya cargadas las capas, se procede a que se ejecuten, ambas a la vez, pero previamente, se predice lo que tendría que sacar la red; para ello, se prepara un script de Matlab con todos los datos que se han mencionado y se hace la simulación, obteniendo los siguientes resultados:

L1N1_hex = '0065E' (0.0995)

L1N2_hex = '01FC0' (0.0303)

L2N1_hex = '02000' (0.5)

L2N2_hex = '02000' (0.5)

La nomenclatura es L =Layer(Capa), N= Neurona. Esas son las salidas que deberían salir en la primera ejecución de las capas, en la Figura 48 se muestran las salidas obtenidas tras ejecutar las capas.

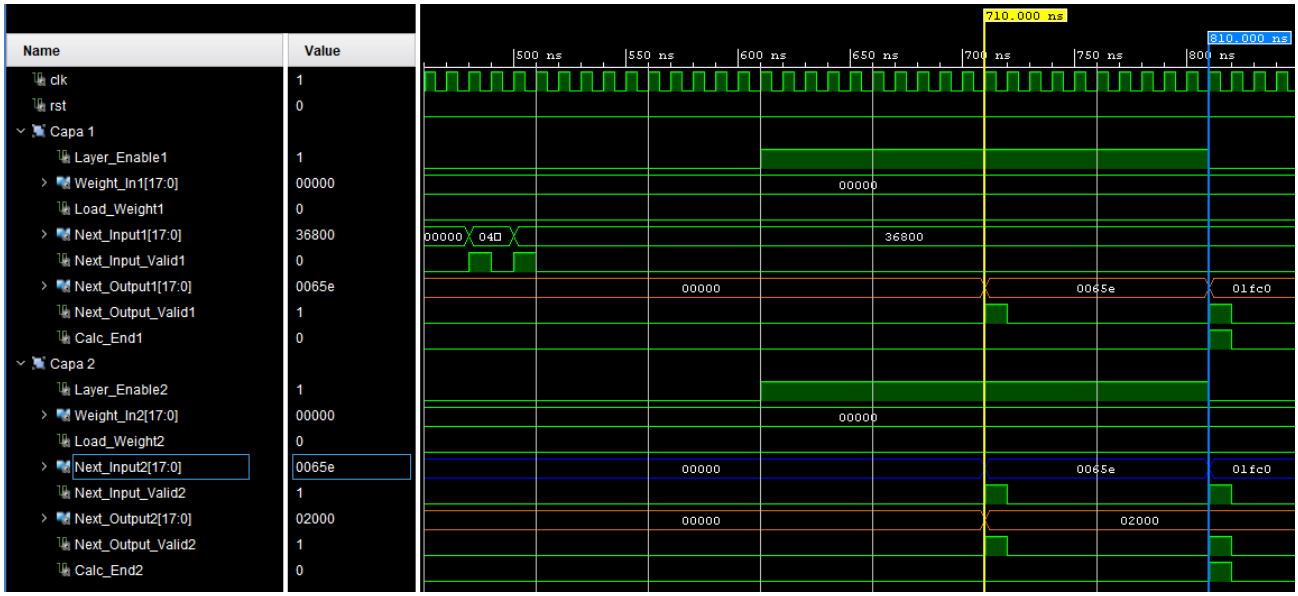

Figura 48. Primera Ejecución de la red neuronal bicapa.

Para facilitar la interpretación de las simulaciones, se ha decidido usar colores. En marrón se han marcado las salidas de las dos capas, la superior es la capa número 1, que devuelve dos salidas: 0x0065E y 0x01FC0, que coinciden con lo predicho por Matlab. La segunda capa ha devuelto dos 0x02000, que también coincide con lo predicho.

Por otro lado, en azul, se puede ver que en la entrada Next_Input de la segunda capa se ha conectado la salida de la primera, la segunda capa, al no haber cargado nuevos datos, se ejecuta con los datos anteriores, por lo que se ejecuta una segunda vez las capas, la nueva situación es la siguiente:

- Capa 1
  - Entrada 1: 0x0065E (0.0995)
  - Entrada 2: 0x01FC0 (0.0303)
- Capa 2
  - Entrada 1: 0x0065E (0.0995)
  - Entrada 2: 0x01FC0 (0.0303)

Al igual que en el caso anterior, se pasa primero por el script de Matlab para después poder compararlo con lo que se extraiga de la FPGA.

L1N1_hex = ‘0065E’ (0.0995)

L1N2_hex = ‘01FC0’ (0.0303)

L2N1_hex = ‘01F00’ (0.4844)

L2N2_hex = ‘034E9’ (0.8267)

Ahora, con estos resultados teóricos, se observa la simulación en la Figura 49.

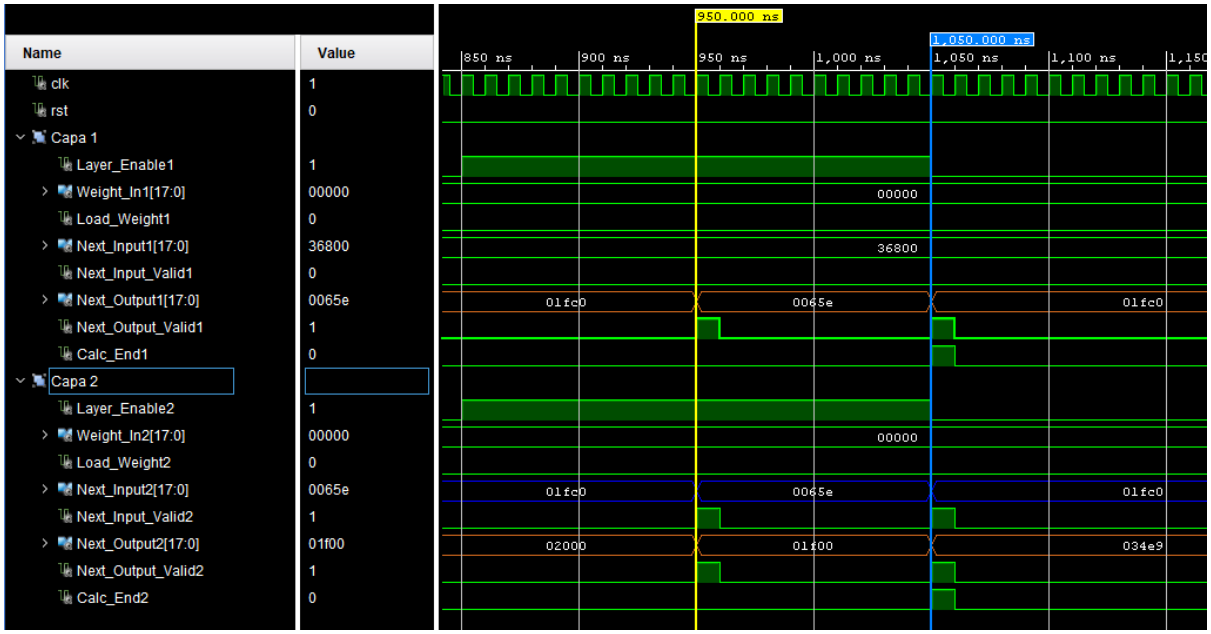

Figura 49. Segunda ejecución de la red neuronal bicapa.

El código de colores es el mismo que en el caso anterior, se tiene que en la primera capa, las salidas son: 0x0065E y 0x01FC0, las mismas que en el caso anterior, por lo que están bien. La segunda capa devuelve: 0x01F00 y 0x034E9, lo mismo que se había predicho, por lo que se concluye que los resultados son correctos y se valida la posibilidad de hacer perceptrones multicapa con la arquitectura propuesta.

5. Consumo de recursos de la arquitectura

En este apartado se va a realizar un pequeño desglose de los consumos que tiene la arquitectura y de los recursos que se poseen actualmente con el chip con el que se ha desarrollado el proyecto. Este proyecto está orientado para su uso bajo una plataforma Zynq de Xilinx, aunque es directamente compatible con cualquier FPGA de la serie 7 de Xilinx. La plataforma Zynq 7000 posee una estructura basada en 2 núcleos ARM cortex A9 y una capa FPGA que puede ser interconectada con los cores, un resumen de la arquitectura puede verse en la Figura 50.

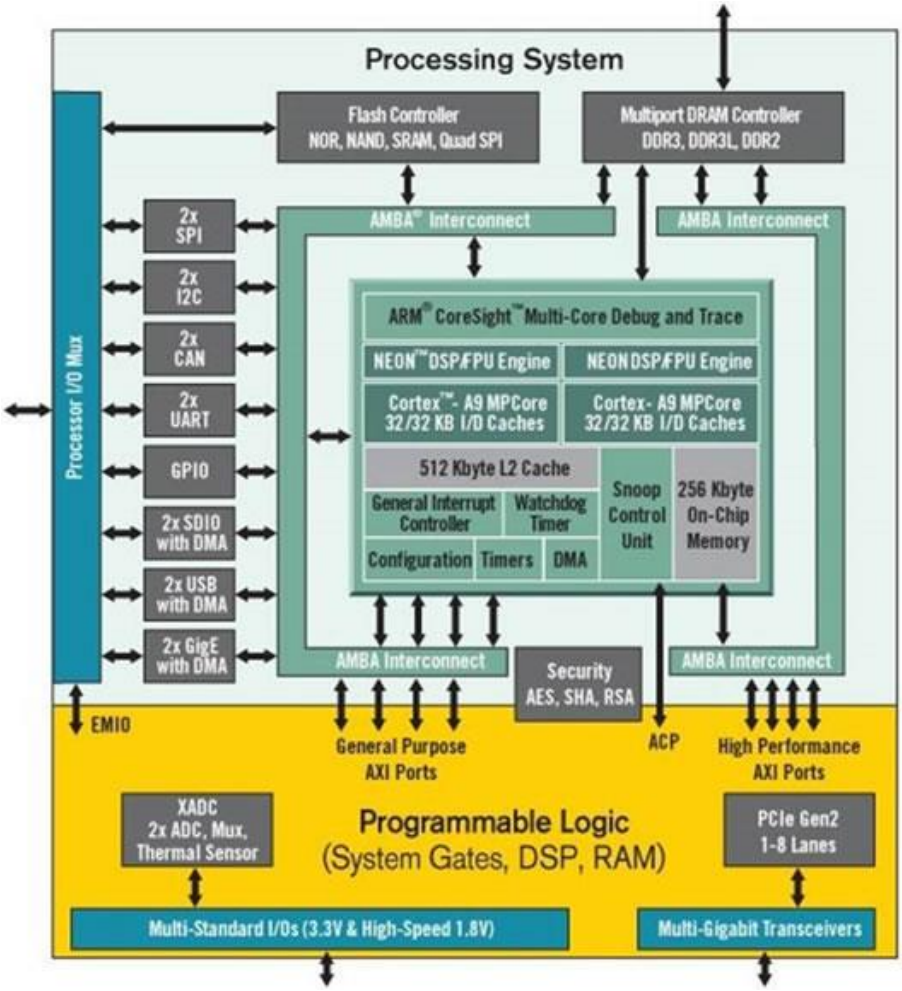

Figura 50. Arquitectura de la familia Zynq 7000. Fuente: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

Aunque la parte de los núcleos ARM pueden ser interesantes para entrenar la red, no es un tema que centre este trabajo. La parte de lógica programable depende de la versión de Zynq que se utilice. En este caso particular, está orientando el proyecto para una tarjeta de evaluación ZedBoard, que se puede ver en la Figura 51.

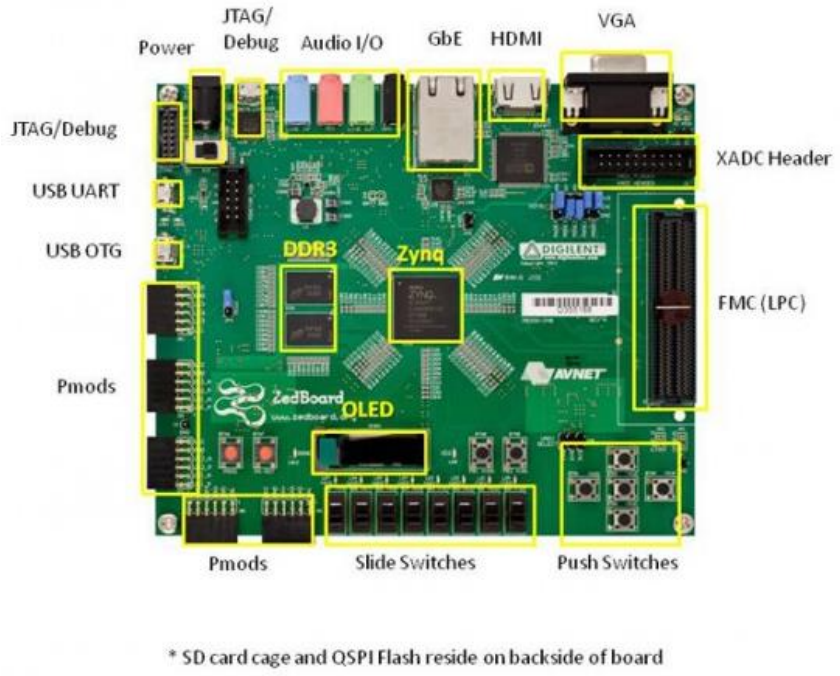

Figura 51. Imagen de la plataforma de desarrollo ZedBoard. Fuente: http://www.zedboard.org/product/zedboard

Esta tarjeta posee un Zynq XC7Z020-1CLG484C, que tiene una FPGA de gama media con las características que se muestran en la Figura 52.

Zynq®-7000 SoC Family

		Cost-Optimized Devices						Mid-Range Devices			
	Device Name	Z-7007S	Z-7012S	Z-7014S	Z-7010	Z-7015	Z-7020	Z-7030	Z-7035	Z-7045	Z-7100
	Part Number	XC7Z007S	XC7Z012S	XC7Z014S	XC7Z010	XC7Z015	XC7Z020	XC7Z030	XC7Z035	XC7Z045	XC7Z100
Processing System (PS)	Processor Core	Single-Core ARM® Cortex™-A9 MPCore™ Up to 766MHz			Dual-Core ARM Cortex-A9 MPCore Up to 866MHz			Dual-Core ARM Cortex-A9 MPCore Up to 1GHz[1]			
	Processor Extensions	NEON™ SIMD Engine and Single/Double Precision Floating Point Unit per processor									
	L1 Cache	32KB Instruction, 32KB Data per processor									
	L2 Cache	512KB									
	On-Chip Memory	256KB									
	External Memory Support[2]	DDR3, DDR3L, DDR2, LPDDR2									
	External Static Memory Support[2]	2x Quad-SPI, NAND, NOR									
	DMA Channels	8 (4 dedicated to PL)									
	Peripherals	2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x 32b GPIO									
	Peripherals w/ built-in DMA[2]	2x USB 2.0 (OTG), 2x Tri-mode Gigabit Ethernet, 2x SD/SDIO									
	Security[3]	RSA Authentication of First Stage Boot Loader, AES and SHA 256b Decryption and Authentication for Secure Boot									
	Processing System to Programmable Logic Interface Ports (Primary Interfaces & Interrupts Only)	2x AXI 32b Master, 2x AXI 32b Slave 4x AXI 64b/32b Memory AXI 64b ACP 16 Interrupts									
Programmable Logic (PL)	7 Series PL Equivalent	Artix®-7	Artix-7	Artix-7	Artix-7	Artix-7	Artix-7	Kintex®-7	Kintex-7	Kintex-7	Kintex-7
	Logic Cells	23K	55K	65K	28K	74K	85K	125K	275K	350K	444K
	Look-Up Tables (LUTs)	14,400	34,400	40,600	17,600	46,200	53,200	78,600	171,900	218,600	277,400
	Flip-Flops	28,800	68,800	81,200	35,200	92,400	106,400	157,200	343,800	437,200	554,800
	Total Block RAM (# 36Kb Blocks)	1.8Mb (50)	2.5Mb (72)	3.8Mb (107)	2.1Mb (60)	3.3Mb (95)	4.9Mb (140)	9.3Mb (265)	17.6Mb (500)	19.2Mb (545)	26.5Mb (755)
	DSP Slices	66	120	170	80	160	220	400	900	900	2,020
	PCI Express®	—	Gen2 x4	—	—	Gen2 x4	—	Gen2 x4	Gen2 x8	Gen2 x8	Gen2 x8
	Analog Mixed Signal (AMS) / XADC[2]	2x 12 bit, MSPS ADCs with up to 17 Differential Inputs									
	Security[3]	AES & SHA 256b Decryption & Authentication for Secure Programmable Logic Config									
	Speed Grades Commercial	-1			-1			-1			-1
	Speed Grades Extended	-2			-2,-3			-2,-3			-2
	Speed Grades Industrial	-1, -2			-1, -2, -1L			-1, -2, -2L			-1, -2, -2L

Figura 52. Características generales de la familia Zynq. Fuente: https://www.xilinx.com/support/documentation/selection-guides/zynq-7000-product-selection-guide.pdf

Como se puede observar, hay una gran variedad de tamaños, por lo que se podrá ajustar a las necesidades de cada caso, incluso se podría ir con una FPGA sin procesadores, que sería aún más barato.

En particular, en el chip que nos ocupa tenemos:

85K celdas En particular, en el chip se tiene disponible:

- 85K celdas lógicas.
- 53200 Look-Up Tables.
- 106400 Flip Flops.
- 140 BRAMs de 36 Kb.
- 220 celdas DSP.

Para comprobar el consumo de cada uno de los bloques, es necesario sintetizarlo y optimizarlo, para ello se usa Vivado con los ajustes por defecto, para la neurona se obtiene la siguiente tabla resumen, mostrada en la Figura 53.

Resource	Utilization	Available	Utilization %
LUT	89	53200	0.17
FF	86	106400	0.08
BRAM	0.50	140	0.36
DSP	1	220	0.45
IO	58	200	29.00
BUFG	1	32	3.13

Figura 53. Consumo de recursos de una neurona.

Como se puede observar, los porcentajes son bastante pequeños, lo que invita a pensar que pueden caber muchas estructuras como la actual. Por cómo se ha pensado la arquitectura, da igual el número de entradas que tenga la neurona, va a consumir la misma cantidad de recursos. Un vistazo rápido es suficiente para darse cuenta de que en el caso de querer instanciar solo neuronas, el recurso limitante serían las celdas DSP, que son las encargadas de realizar la multiplicación y la acumulación. En este caso, el número máximo de neuronas que seríamos capaces de instanciar antes de quedarnos sin DSPs sería de 220.

En el caso de la capa, se va a realizar el mismo análisis, se va a sintetizar la capa (da igual el número de neuronas y entradas con las que se defina porque están multiplexadas), los resultados pueden verse en la Figura 54.

Resource	Utilization	Available	Utilization %
LUT	254	53200	0.48
FF	272	106400	0.26
BRAM	1.50	140	1.07
DSP	1	220	0.45
IO	61	200	30.50
BUFG	1	32	3.13

Figura 54. Consumo de recursos de una capa.

En este caso, tal y como se puede ver, el máximo consumo de recursos es de BRAMs, nótese que cada capa tiene un consumo de 3 BRAMs, por lo que el número máximo de capas que se podrá instanciar es de 93.

6. Límites de la arquitectura

En este apartado se pretende explorar los límites de la arquitectura propuesta, ya que pueden ser posibles puntos de mejora para trabajos futuros. El primer límite que aparece siempre que se hable de un diseño en FPGA es la frecuencia máxima de funcionamiento, para ello, hay que hacer un análisis del timing de la solución propuesta. Para poder probar esto, se está sintetizando una capa de 32 neuronas con 32 entradas (peor caso). En Vivado, que es la herramienta que se está utilizando actualmente para el diseño, esto se encuentra tras sintetizar el proyecto, hay que ejecutar un análisis del timing y realizar una serie de suposiciones de a qué velocidad se desea ir.

Como primera aproximación, se va a fijar una frecuencia de reloj de 100 MHz, que es una velocidad aceptable para este tipo de diseños, tras fijar esta velocidad y ejecutar el análisis, los resultados obtenidos se muestran en la Figura 55.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	4,933 ns	Worst Hold Slack (WHS):	0,101 ns	Worst Pulse Width Slack (WPWS):	4,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	552	Total Number of Endpoints:	552	Total Number of Endpoints:	279

All user specified timing constraints are met.

Figura 55. Resumen de temporización para una frecuencia de reloj de 100 MHz.

Se tiene un slack negativo de 4.93 ns, por lo que se podría ajustar el reloj aún más, casi hasta los 200 MHz (actualmente a 100 MHz el reloj es de 10 ns de periodo). En la Figura 56 se muestra el resultado si se declara el reloj como de 200 MHz.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,203 ns	Worst Hold Slack (WHS):	0,099 ns	Worst Pulse Width Slack (WPWS):	2,000 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	552	Total Number of Endpoints:	552	Total Number of Endpoints:	279

All user specified timing constraints are met.

Figura 56. Resumen de temporización para una frecuencia de reloj de 200 MHz.

Tal y como se puede apreciar en el reporte, el sistema de capaz de funcionar a 200 MHz.

6.1. Número máximo de Neuronas/Entradas por capa

Como se comentó anteriormente, las BRAM internas a las capas tienen una limitación de espacio. La más crítica es la BRAM de los pesos, ya que como máximo se pueden cargar 1024 pesos, por lo que el producto entre neuronas y entradas no puede exceder 1024. Los pares de límites serían entonces los siguientes:

- 1 entrada y 1024 neuronas.
- 1024 entradas y 1 neurona.
- 32 entradas y 32 neuronas.

Los dos primeros no tienen sentido, ya que la siguiente capa solo podría ser análoga, por lo que saldría una red sin sentido. Por otro lado, el número máximo de entradas es 512, ya que, recuérdese, las entradas pasan por una memoria configurada en modo Ping-Pong, por lo que solo se puede utilizar la mitad de la memoria en cada ejecución, por lo que actualizando los límites quedarían así:

- 1 entrada y 1024 neuronas.
- 512 entradas y 2 neuronas.
- 32 entradas y 32 neuronas.

6.2. Número máximo de capas

Este límite es independiente de las neuronas que haya por capa, tal y como está pensada la arquitectura, consumen los mismos recursos una capa de 32 entradas y 32 neuronas que una capa de 1 entrada y 1 neurona. Por tanto, el consumo de recursos de 1 capa es lo mostrado en la Figura 57.

Resource	Utilization	Available	Utilization %
LUT	254	53200	0.48
FF	272	106400	0.26
BRAM	1.50	140	1.07
DSP	1	220	0.45
IO	61	200	30.50
BUFG	1	32	3.13

Figura 57. Consumo de recursos de una capa.

Como se puede observar, quitando los IO (que en este caso no son relevantes), el mayor consumo porcentual es de BRAM, consumiendo cada capa 1.5 BRAM de 36 Kb (la BRAM de la función de activación, la BRAM de almacenamiento de pesos y la BRAM de almacenamiento de entradas). Por tanto, haciendo las cuentas el número máximo de capas que se podrían generar con el chip actual es de:

$$Nº\ capas = \frac{140}{1.5} = 93.33 = 93\ capas$$

Se podrían generar 93 capas, suponiendo capas iguales de 32 entradas y 32 neuronas, hacen un total de:

$$Nº\ neuronas = 32 * 93 = 2976\ neuronas$$

6.3. Frecuencia de actualización de la red y latencia

La estrategia de multiplexación tiene sus ventajas y sus desventajas, la ventaja es que se consigue un gran ahorro de recursos, consiguiendo así con un chip más barato y más pequeño los mismos

resultados que con un chip más potente. Pero también tiene la desventaja de que vas a tardar más tiempo en conseguir los mismos.

Para calcular la frecuencia mínima a la que se va a poder extraer los datos, se considera el peor caso, asumiendo que en algún punto de la red una capa en la que hay 512 entradas y dos neuronas, como es la capa más lenta que se puede construir, va a ser la que marque la frecuencia máxima a la que se puede ejecutar toda la red. Se procede a continuación a realizar un par de simulaciones para estimar esa frecuencia; en la Figura 58 se puede ver el tiempo que se invierte en cargar los pesos de una capa.

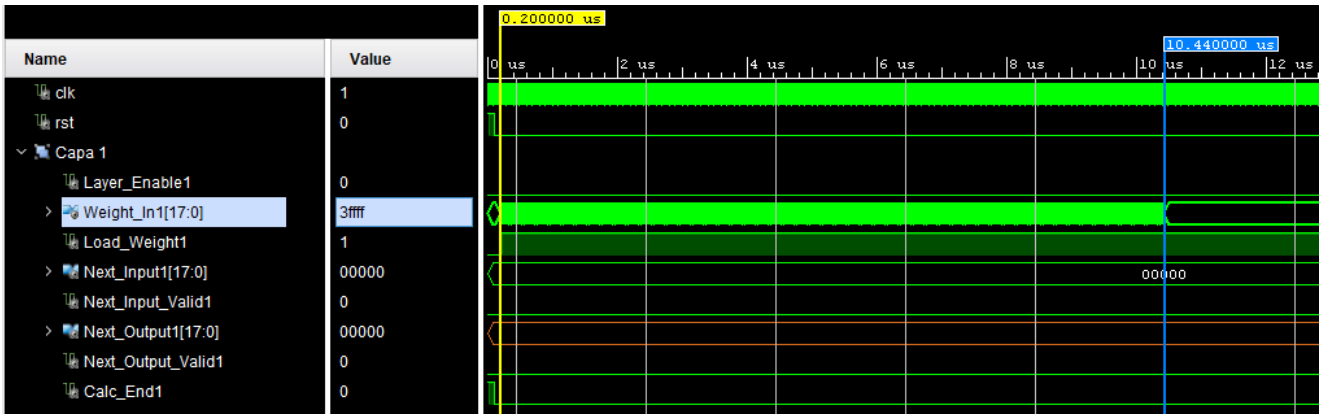

Figura 58. Tiempo invertido en cargar 1024 pesos.

Como se puede observar, se tarda 10.24us, a 10 ns por ciclo de reloj son 1024 ciclos de reloj. En la Figura 59 se puede observar el tiempo que se tarda entre que se le da a la capa la orden de ejecutarse y saca sus resultados.

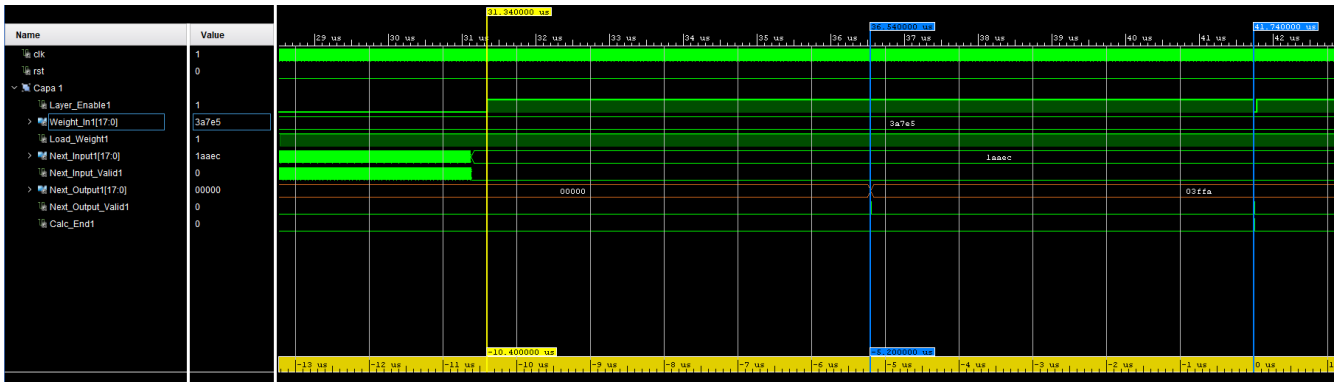

Figura 59. Tiempo invertido en procesar una capa de 512 entradas y dos neuronas.

Como se puede ver, es un ejemplo complicado, pero es el caso límite. Se puede apreciar que tarda en ejecutar cada neurona 5.2us y la capa entera en 10.4us. Por tanto, la frecuencia máxima a la que podrían ejecutarse las capas y la frecuencia máxima a la que se podría alimentar esta red sería de:

$$f_{max} = \frac{1}{10.4\ us} = 96.154\ kHz$$

Es decir, esta red podría sacar datos 96154 veces por segundo. Por otro lado, la latencia del sistema, suponiendo el peor caso en el que tengamos 93 capas iguales a la anterior, sería de:

$$Latencia = 93 * 10.4\ us = 0.9672\ ms$$

Es decir, desde que se introduce la información a la red hasta que lo que aparece a la salida pasarían como máximo en el peor caso 1 ms.

7. Conclusiones y trabajos futuros

En este apartado se van a obtener las conclusiones que se pueden extraer del trabajo realizado y las pautas a seguir en los posibles trabajos futuros que surgen de los aspectos a mejorar de la arquitectura.

7.1. Conclusiones

En este subapartado vamos a hablar de las conclusiones que se han podido extraer.

Se ha conseguido el objetivo principal de implementar en una FPGA una arquitectura que permite modelar y ejecutar redes neuronales. El diseño posee la flexibilidad necesaria para modelar distintas topologías de red y distintas precisiones para los datos. Además, se ha conseguido una frecuencia de funcionamiento del sistema razonablemente alta. A continuación, se confecciona una lista con los objetivos logrados.

1. **Es posible generar una red neuronal dentro de una FPGA.** Aunque el consumo de recursos a priori parecía alto, si se realiza la doble multiplexación es posible albergar un gran número de neuronas con un consumo acotado.
2. **La frecuencia de actualización de la red es suficiente para una aplicación orientada a la robótica**. Este punto es importante, se consigue en el peor de los casos una tasa de actualización de casi 100 kHz, que es una tasa más que suficiente, además, el tiempo de respuesta en el peor de los casos es de 1 ms, que para un robot, es un tiempo bastante pequeño (los ciclos de control suelen ser de 50-100 ms).
3. **Se ha conseguido la flexibilidad proyectada**. Se ha conseguido generar una arquitectura flexible en el formato de los datos, ya que mediante los genéricos se pueden ajustar los bits dedicados a la parte entera y los bits dedicados a la parte decimal. También se ha conseguido una arquitectura flexible que permite formar infinidad de redes distintas, ya que se pueden enlazar unas capas a otras y formar grandes redes.
4. **Se ha logrado minimizar el consumo de recursos.** Este apartado era muy importante en la lista de objetivos, ya que, si este objetivo no se conseguía, no era posible utilizar una FPGA para este propósito, y esto era necesario para poder minimizar los costes de poner en marcha un sistema con red neuronal incorporada. En este ámbito siempre hay posibilidad de mejoras, como se verá en el apartado de trabajos futuros.
5. **Se ha conseguido una frecuencia de reloj alta.** Esto era un objetivo importante y que depende directamente de cómo se genere el código VHDL, si se siguen buenas prácticas o no. Al igual que en el caso anterior es un área en el que se puede mejorar.

7.2. Trabajos Futuros

En este apartado vamos a exponer algunos trabajos futuros que pueden continuar la línea abierta por este proyecto.

- **Comparación de eficiencia y consumo de recursos con un diseño HLS.** Sería muy interesante poder comparar el diseño de la misma arquitectura con lo extraído de la herramienta de codificación en C de Xilinx (Vivado HLS), para ver si es más eficiente o se consigue mejores resultados que con la codificación directa en VHDL.
- **Incorporación de la lógica necesaria para el entrenamiento de la red**. Sería interesante también hacer un proyecto para añadir a la arquitectura actual toda la lógica necesaria para poder entrenar la red in situ. Obviamente, incorporar esta lógica limitaría el tamaño de la red, pero sería muy interesante.
- **Aumento de la eficiencia en el consumo de recursos.** Sería también interesante mejorar aún más la eficiencia, por ejemplo, compartiendo las memorias de la neurona de cada capa entre dos capas, con esto se reduciría en un 30% el consumo de BRAM, o centralizando el almacenamiento de los pesos.
- **Implementación de interacción con los núcleos ARM.** Otro proyecto que se podría plantear es una vez que se tiene la arquitectura definida, integrarlo en un periférico y darle conectividad con la parte de los procesadores, para poder alimentar la red desde ellos o entrenarla desde los procesadores.

Como se puede observar, se abren varios caminos a partir del trabajo actual, siempre hay un paso posterior a los proyectos de Ingeniería.

Pliego de condiciones

En este apartado se ofrece una lista del material necesario para la realización de este proyecto, en el siguiente capítulo se tabularán los costes de los materiales.

- Equipo portátil MSI GE72 con i7 6700HQ, 16 Gb de RAM, 500 GB SSD.
- Monitor Acer QHD de 32”.
- Tarjeta de evaluación ZedBoard.
- Licencia de Matlab 2018b.
- Licencia de Office 365.
- Vivado 2017.4 HLS Webpack Edition.

Se excluye de este pliego de condiciones los fungibles necesarios como papel, bolígrafos, clips, grapas y demás equipo que se haya utilizado en la fase de planificación y diseño a mano. Tampoco se incluyen en este aparatado los elementos de mobiliario ya existentes ni el gasto energético de los equipos.

Presupuesto

En este apartado se van a definir los costes teóricos que han tenido este proyecto, esto incluirá tanto los costes materiales como los costes por trabajo personal.

Costes materiales

Objeto	Precio Unitario(€)	Unidades	Precio Total (€)
MSI GE72 (i7 4 GHz)	1100	1	1200
Monitor Acer QHD	600	1	250
ZedBoard	450	1	450
Matlab	120	1	120
Licencia de Office	0	1	0
Vivado WebPack	0	1	0
Total (IVA incluido)			2020

Costes profesionales

Actividad	Precio (€)	Tiempo(Meses)	Precio Total (€)
Ingeniería	1800	3	5400
Escritor	1000	1	1000
Total (IVA incluido)			6400

Costes totales

Costes Materiales	**2020 €**
Mano de obra	6400 €
Total	8420 €

Bibliografía

[1] GEINTRA, «Página Principal Geintra,» 2015. [En línea]. Available: http://www.geintra-uah.org/. [Último acceso: 1 Julio 2019].

[2] I. S. Janardan Misra, «Artificial neural networks in hardware: A survey of two decades of progress,» *Neurocomputing,* vol. 74, nº 1-3, pp. 239-255, 2010.

[3] J. Qiu, J. Wang y S. Yao, «Going Deeper with Embedded FPGA Platform for Convolutional Neural Network,» de *2016 ACM/SIGDA International Symposium*, Monterey, CA, USA, 2016.

[4] S. Himavathi, D. Anitha y A. Muthuramalingam, «Feedforward Neural Network Implementation in FPGA Using Layer Multiplexing for Effective Resource Utilization,» *IEEE Transactions on Neural Networks,* vol. 18, nº 3, pp. 880 - 888, 2007.

[5] J. T. d. S. Peter Y. K. Cheung George A. Constantinides, Field-Programmable Logic And Applications, Lisboa: Springer, 2003.

[6] Xilinx, «Vivado Design Suite 7 Series FPGA,» [En línea]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug953-vivado-7series-libraries.pdf.

[7] E. Fiesler, «Neural Network Topologies,» p. 17, 1996.

[8] S. Hochreiter, «Long Short-Term Memory,» *Neural Computation,* vol. 9, nº 8, pp. 1735-80, 1997.

[9] H. B. D. Martin T. Hagan, Neural Network Design, Oklahoma: Martin Hagan, 2014.

[10] A. Muthuramalingam, S. Himavathi y E. Srinivasan, «Neural Network Implementation Using FPGA: Issues and Application,» *International Journal of Information Technology,* vol. 4, nº 2, 2007.

Universidad de Alcalá

Escuela Politécnica Superior