UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES CUAUTITLÁN.

Implementación de una red neuronal multicapa en una tarjeta de desarrollo ZYBO Z7-10 con un procesador ARM y FPGA Artix-7

T E S I S

PARA OBTENER EL TÍTULO DE

INGENIERO EN TELECOMUNICACIONES, SISTEMAS Y ELECTRÓNICA

PRESENTAN:

MATA GAMA EDUARDO TONATIHU

RAMÍREZ SÁNCHEZ MARCO ALEXANDER

ASESOR: ING. JOSÉ LUIS BARBOSA PACHECO

CUAUTITLÁN IZCALLI, ESTADO DE MÉXICO, 2019

VNIVERSIDAD NACIONAL AVPNOMA DE MEXICO

FACULTAD DE ESTUDIOS SUPERIORES CUAUTITLÁN
SECRETARÍA GENERAL
DEPARTAMENTO DE EXÁMENES PROFESIONALES

U. N. A. M.
FACULTAD DE ESTUDIOS SUPERIORES CUAUTITLAN

ASUNTO: VOTO APROBATORIO

M. en C. JORGE ALFREDO CUÉLLAR ORDAZ
DIRECTOR DE LA FES CUAUTITLAN
PRESENTE

ATN: I.A. LAURA MARGARITA CORTAZAR FIGUEROA
Jefa del Departamento de Exámenes Profesionales
de la FES Cuautitlán.

Con base en el Reglamento General de Exámenes, y la Dirección de la Facultad, nos permitimos comunicar a usted que revisamos el: **Trabajo de Tesis**

Implementación de una red neuronal multicapa en una tarjeta de desarrollo ZYBO Z7-10 con un procesador ARM y FPGA Artix-7

Que presenta el pasante: EDUARDO TONATIHU MATA GAMA
Con número de cuenta: 31125264-8 para obtener el Título de la carrera: Ingeniería en Telecomunicaciones, Sistemas y Electrónica

Considerando que dicho trabajo reúne los requisitos necesarios para ser discutido en el **EXAMEN PROFESIONAL** correspondiente, otorgamos nuestro **VOTO APROBATORIO**.

ATENTAMENTE
"POR MI RAZA HABLARÁ EL ESPÍRITU"
Cuautitlán Izcalli, Méx. a 07 de mayo de 2019.

PROFESORES QUE INTEGRAN EL JURADO

	NOMBRE	FIRMA
PRESIDENTE	Mtro. Jorge Buendía Gómez	
VOCAL	Ing. José Luis Barbosa Pacheco	
SECRETARIO	Ing. Noemi Hernández Domínguez	
1er. SUPLENTE	Mtro. Leopoldo Martín del Campo Ramírez	
2do. SUPLENTE	Dr. David Tinoco Varela	

NOTA: los sinodales suplentes están obligados a presentarse el día y hora del Examen Profesional (art. 127).

LMCF/javg

VNIVERSIDAD NACIONAL AVFNOMA DE MEXICO

FACULTAD DE ESTUDIOS SUPERIORES CUAUTITLÁN
SECRETARÍA GENERAL
DEPARTAMENTO DE EXÁMENES PROFESIONALES

U. N. A. M.
FACULTAD DE ESTUDIOS SUPERIORES CUAUTITLAN

ASUNTO: VOTO APROBATORIO

M. en C. JORGE ALFREDO CUÉLLAR ORDAZ
DIRECTOR DE LA FES CUAUTITLAN
PRESENTE

ATN: I.A. LAURA MARGARITA CORTAZAR FIGUEROA
Jefa del Departamento de Exámenes Profesionales
de la FES Cuautitlán.

Con base en el Reglamento General de Exámenes, y la Dirección de la Facultad, nos permitimos comunicar a usted que revisamos el: **Trabajo de Tesis**

Implementación de una red neuronal multicapa en una tarjeta de desarrollo ZYBO Z7-10 con un procesador ARM y FPGA Artix-7

Que presenta el pasante: MARCO ALEXANDER RAMÍREZ SÁNCHEZ
Con número de cuenta: 31022593-9 para obtener el Título de la carrera: Ingeniería en Telecomunicaciones, Sistemas y Electrónica

Considerando que dicho trabajo reúne los requisitos necesarios para ser discutido en el EXAMEN PROFESIONAL correspondiente, otorgamos nuestro **VOTO APROBATORIO**.

ATENTAMENTE
"POR MI RAZA HABLARÁ EL ESPÍRITU"
Cuautitlán Izcalli, Méx. a 07 de mayo de 2019.

PROFESORES QUE INTEGRAN EL JURADO

	NOMBRE	FIRMA
PRESIDENTE	Mtro. Jorge Buendía Gómez	
VOCAL	Ing. José Luis Barbosa Pacheco	
SECRETARIO	Ing. Noemi Hernández Dominguez	
1er. SUPLENTE	Mtro. Leopoldo Martín del Campo Ramírez	
2do. SUPLENTE	Dr. David Tinoco Varela	

NOTA: los sinodales suplentes están obligados a presentarse el día y hora del Examen Profesional (art. 127).

LMCF/javg

Agradecimientos

A nuestro profesor José Luis Barbosa Pacheco.

Gracias por todo el apoyo brindado, por que desde que nos conocimos nos llevo a dar nuestro máximo esfuerzo, por motivarnos a siempre seguir estudiando, por que con su exigencia nos hizo mejores estudiantes, por orientarnos, apoyarnos y enseñarnos durante la carrera y hasta este trabajo, gracias por ser un profesor en toda la extensión de la palabra, sin duda alguna tuvo un impacto muy positivo en nuestra carrera ya que al impartir las clases con el gran conocimiento que posee y la gran pasión con la que lo comparte nos contagio a ser mejores en todos los aspectos, siempre sera un ejemplo a seguir, se le tiene una profunda admiración, respeto y cariño. Gracias totales.

A la Universidad

Por ser nuestro segundo hogar durante todo este tiempo, por abrirnos las puertas de sus aulas y laboratorios, para forjarnos como ingenieros, porque ahí vivimos experiencias y momentos inolvidables y nada es comparable con el orgullo que significa formar parte de esta institución. Por todo eso y mas, gracias. “Por mi raza hablara el espíritu”

Índice

Introducción

Desde siempre, la tecnología es una herramienta para la solución de distintos tipos de problemas, la vida como la percibimos hoy en día, sería muy diferente sin los avances de la misma, sin embargo, gran parte de estos avances no serían posibles bajo el cómputo convencional, ya que tardaría demasiado en procesar la información para arrojar un resultado o simplemente no lo lograría, algunos de estos casos pueden ser la identificación de patrones, análisis de señales, predicciones, entre otros más. [1]

Para estos casos el desarrollo de la Inteligencia Artificial (IA) se torna protagonista ya que logra dar solución a dichos problemas.

En la IA existen 4 pilares básicos:

- Algoritmos Genéticos: Análogos al proceso de adaptación en la evolución natural de las cadenas de ADN.
- Redes Neuronales: Emulan el funcionamiento del cerebro humano en cuanto aprendizaje se refiere.
- Sistemas Difusos: Imitan el razonamiento humano, ya no son solo 0ś y 1's para la toma de decisiones.
- Sistemas Inteligentes: Buscan la mejor solución posible para maximizar sus posibilidades de éxito en alguna tarea.

La inteligencia artificial se encarga de estudiar modelos de cómputo capaces de realizar actividades propias de los seres humanos, como son, la capacidad de adaptación, aprendizaje, razonamiento, y la toma de decisiones.

Dado que el interés de este trabajo es implementar y evaluar un sistema que aprenda y a partir de este aprendizaje ejecute ciertas funciones, se enfocara en las Redes Neuronales.

Para implementar dicho sistema se utilizará la tarjeta de desarrollo ZYBO Z7-10 la cual pertenece a los dispositivos de la serie Zynq-7000, estos dispositivos tienen una arquitectura que ayuda a la implementación y desarrollo del la IA, dicha arquitectura está dividida en dos

partes, el sistema de proceso y la lógica programable. El primero cuenta con una unidad de procesamiento de aplicaciones, la cual contiene dos núcleos de procesadores ARM[1], cada ARM tiene ciertos recursos, como son una unidad NEON™ MPE (Motor de Procesamiento de Medios)[2], la extensión para punto flotante, el MMU (Unidad de Administración de Memoria), entre otros, que se adaptan a las operaciones y velocidades requeridas en algoritmos de IA. En la segunda parte se tiene la lógica programable basada en un FPGA (Arreglo Genérico Programable en Campo), los FPGAs cuentan con un procesamiento en paralelo, adaptándose perfectamente a las redes neuronales artificiales pues una de sus características es dicho procesamiento. En geneal los algoritmos de IA necesitan de procesamiento en paralelo y no de tipo secuencial como se haría en el cómputo convencional.

Dicha tarjeta se torna ideal ya que permite el desarrollo de software y hardware, obteniendo flexibilidad a la hora del diseño, de esta manera se pueden dividir las tareas que involucra una red neuronal artificial, como el entrenamiento de la ejecución.

En la primera parte se da un panorama general de la teoría que conforma el marco teórico, se hablará brevemente de la neurona biológica, para así poder pasar al modelo matemático de las redes neuronales artificiales, las bases que lo sustentan y sus algoritmos más importantes. Después, se adentrará a los sistemas embebidos dando un panorama general sobre procesadores, arquitectura ARM, dispositivos FPGA y sistemas en un solo chip, para finalizar con la descripción de la tarjeta que se utilizará para implementar la red neuronal artificial. Posteriormente se abordará el diseño de la red neuronal, como fue estructurada y programada, los recursos, algoritmos y estándares utilizados, para de esta manera implementarla y realizar las pruebas que puedan permitir analizar su desempeño y con esto poder evaluar los resultados de una red neuronal embebida en la tarjeta ya mencionada.

1 ARM es una arquitectura de procesadores tipo RISC desarrollada por la empresa ARM Holdings.

2 NEON™ MPE es la unidad encargada de realizar y acelerar operaciones aritméticas.

Objetivos

Diseñar e implementar una red neuronal multicapa en un sistema de un sólo chip (SoC), aprovechando los recursos de hardware de una tarjeta de desarrollo. De tal manera que las tareas de aprendizaje se harán en el microcontrolador y la ejecución en el FPGA.

1 Conceptos básicos de las redes neuronales artificiales

Las redes neuronales, como ya se mencionó, emulan el funcionamiento del cerebro humano en cuanto aprendizaje se refiere. Entonces, es necesario conocer cómo aprende un humano, cual es el proceso por el que obtiene conocimiento y cómo percibe la realidad que lo rodea.

"En el campo interdisciplinario de la ciencia cognitiva convergen modelos computacionales de IA y técnicas experimentales de psicología intentando elaborar teorías precisas y verificables sobre el funcionamiento de la mente humana." [2]

Dentro de este campo se encuentra la hipótesis conexionista, que dice que "el aprendizaje o la conducta inteligente se logra mediante sistemas formados por muchos elementos sencillos, pero muy bien interconectados." Bajo este concepto trabajan las redes neuronales artificiales (RNA), donde los elementos sencillos serán las neuronas, luego, con la adecuada conexión de muchas de ellas y un algoritmo adecuado se obtiene como resultado aprendizaje.

En este capítulo estudiaremos brevemente a la neurona como ente biológico, para pasar al modelo matemático y terminar describiendo uno de sus algoritmos.

1.1 Neurona biológica

El desarrollo de las redes neuronales artificiales ha sido inspirado, en parte, por la observación de cómo los sistemas de aprendizaje biológico están construidos con redes muy complejas de neuronas interconectadas. [3]

El primero en estudiar el tejido nervioso fue Santiago Ramón y Cajal en 1911, en España, quien introduce la idea de las neuronas como estructura constituyente del cerebro.
El cerebro es la parte central del sistema nervioso humano, representado por una red neuronal que continuamente recibe información, la percibe y toma decisiones apropiadas, sin embargo, aun hay mucho que se desconoce sobre el sistema nervioso, no así con la neurona como elemento básico del tejido.

Se estima que el ser humano tiene 10^{11} de neuronas, cada una conectada en promedio con otras 10^{4}. [4]

La neurona, como se ilustra en la figura 1.1, esta conformada por el cuerpo de la célula o soma en donde se aloja el núcleo de ésta. Del cuerpo de la célula salen ramificaciones de diversas fibras conocidas como dendritas y a su vez una fibra más larga, (como si se tratase de un canal) denominada axón, este último al final también se ramifica.

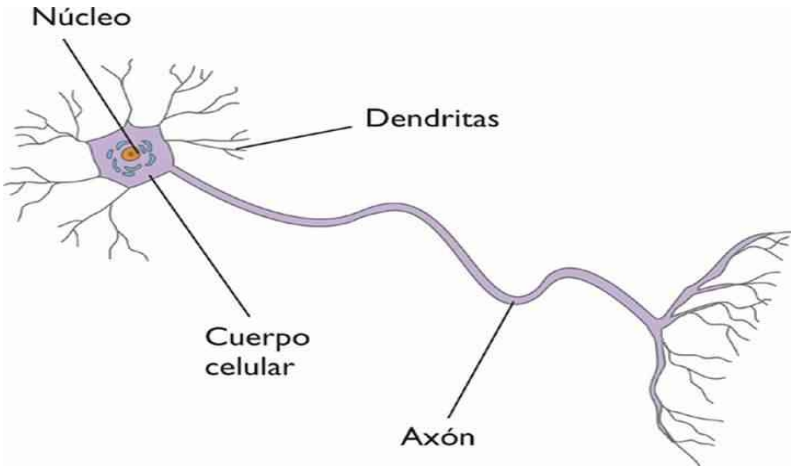


Figura 1.1. Neurona Biológica. [3]

Las dendritas funcionan como receptores captando la información del cuerpo humano o del ambiente externo para convertir esta información en pulsos eléctricos. Estos pulsos eléctricos llegan al soma, el cual procesa la información y "decide" qué hacer, ya sea permanecer en reposo o emitir un pulso eléctrico. A este pulso eléctrico se le conoce como potencial de acción, este potencial de acción viaja hacia otras neuronas por el axón, a esta conexión entre neuronas se le conoce como sinapsis.

La sinapsis es un proceso donde se liberan sustancias químicas (neurotransmisores), existen dos tipos de sinapsis: excitatorias e inhibitorias. Las primeras elevan el potencial de acción, mientras que las segundas lo disminuyen.

Es importante mencionar que los axones no responden a los estímulos inferiores al valor requerido para iniciar un impulso (un valor de umbral). Una vez alcanzado este impulso, la neurona dispara un impulso de salida, que constituye la respuesta de la neurona. [5]

3 "Partes de la neurona". [En línea]. Disponible en: https://www.partesdel.com/partes_de_la_neurona.html. [Consultado: 29-mar-2019].

1.2 Neurona artificial y modelo matemático

La neurona es la unidad fundamental para la operación de las redes neuronales artificiales. A continuación se muestra en la figura 1.2 el modelado de la neurona formal desarrollado por McCulloch y Pits. En el proceso de esta descripción se puede observar la analogía con la neurona biológica.

La neurona artificial, consta de cuatro elementos:

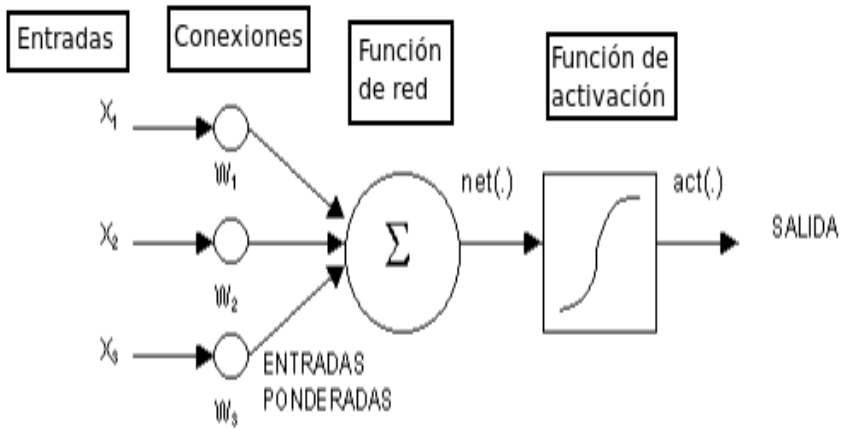


Figura 1.2. Neurona Formal. [4]

- Entradas (X_i): Valor de la neurona, donde llegan las señales de entrada, cada una de ellas con un numero real que indica la intensidad y el signo de la conexión denominado peso (W_i).
- Sumador: efectúa la sumatoria de las entradas con los pesos. A esta operación también se le conoce como net o dinámica.

$$net = \sum_{i=1}^{n} W_i X_i = W_1 X_1 + W_2 X_2 + W_3 X_3 + ... + W_n X_n \ldots (1)$$

4 "Analogía entre las Redes Neuronales Biológicas y Artificiales", rneuronales, 07-nov-2012. .

El net tendrá un “bias” en función del offset, manteniendo así siempre una neurona activa y facilitando el sistema de entrenamiento. Esto se verá en profundidad más adelante.

$$net = \sum_{i=0}^{n} W_i X_i - \theta \quad \ldots (2)$$

- Función de activación: es una función que sirve como umbral, determinando si la neurona dispara o no. Existen varios tipos de funciones, a continuación se muestran dos de las más importantes.

  1) Función Escalón: se utiliza cuando la salida de la red es binaria, la neurona se activará solo cuando el estado de activación es mayor a cierto valor de umbral.

$$S \begin{cases} 1 \text{ si } net > 0 \\ 0 \text{ si } net \leq 0 \end{cases}$$

  2) Función Sigmoide: se utiliza cuando a la salida se requieren valores analógicos o difusos, la importancia de esta función es que su derivada siempre sea positiva y cercana a cero. Para valores grandes positivos o negativos, toma su valor máximo cuando x es igual a cero, esto favorece al entrenamiento de las redes neuronales.

$$f(x) = \frac{1}{1+e^{-x}} \ldots (3)$$

- Salida: la señal de la salida de la neurona

A manera de analogía se puede decir que las entradas X_i son las dendritas de la neurona, los pesos ponderados W_i conforman la sinapsis, la sumatoria junto con la función de activación realizan el proceso de disparo de una neurona, esto es el potencial de acción, para así producir una salida que viene siendo la respuesta de la neurona.

Este modelado es utilizado en la mayoría de las redes neuronales variando unicamente el tipo de función activadora, aunque la función sigmoide es por mucho la mas utilizada. [6]

1.2.1 Hipótesis de Hebb

La hipótesis de Hebb es de gran importancia en el desarrollo de las redes neuronales artificiales ya que fue el fundamento de los algoritmos de dichas redes. Es por eso que antes de estudiar algunos algoritmos se tiene que entender esta hipótesis.

Donald Olding Hebb fue una figura en la psicología, su interés por saber qué ocurre entre el estímulo y la respuesta (percepción, aprendizaje, pensamiento) lo llevaron a publicar su libro "The Organization of Behavior" en 1949.

En dicho libro Hebb propuso su ley de aprendizaje: "Cuando un axón de una célula A está lo suficientemente cercano a una célula B como para excitarla y participa repetida o persistentemente en su disparo, ocurre algún proceso de crecimiento o cambio metabólico en una o en ambas células de modo tal que aumenten tanto la eficiencia de A como la de una de las distintas células que disparan a B". Dicho de otra forma, el aprendizaje significa cambiar la intensidad de conexiones, en consecuencia lo que se busca en los algoritmos de aprendizaje en las redes neuronales artificiales es encontrar los pesos que dan la respuesta correcta para cada entrada. [7]

1.2.2 Perceptrón

En 1958 Frank Rosenblant desarrolló un modelo simple de neurona basado en el modelo de McCulloch y Pits y la hipótesis de Hebb, a este modelo se le llamó perceptrón. Con esto Rosenblant crea el primer algoritmo de aprendizaje el cual consta básicamente de adaptar los pesos de las entradas y el bias para obtener la salida deseada. En el esquema de la figura 1.3 se puede observar de forma general como funciona dicho algoritmo.

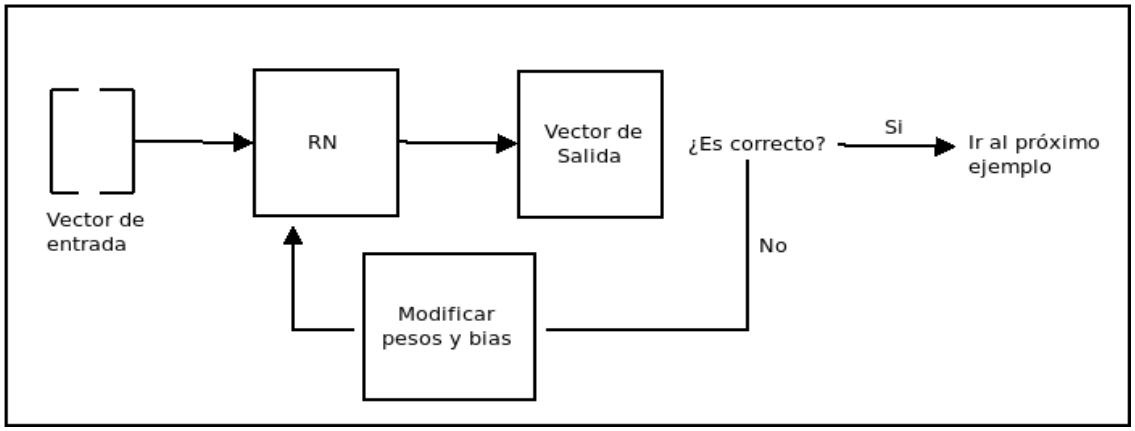


Figura 1.3. Esquema básico del algoritmo de las redes neuronales.

Antes de describir dicho algoritmo es importante recordar ciertos conceptos del modelo McCulloch y Pits, mostrado en la figura 1.4.

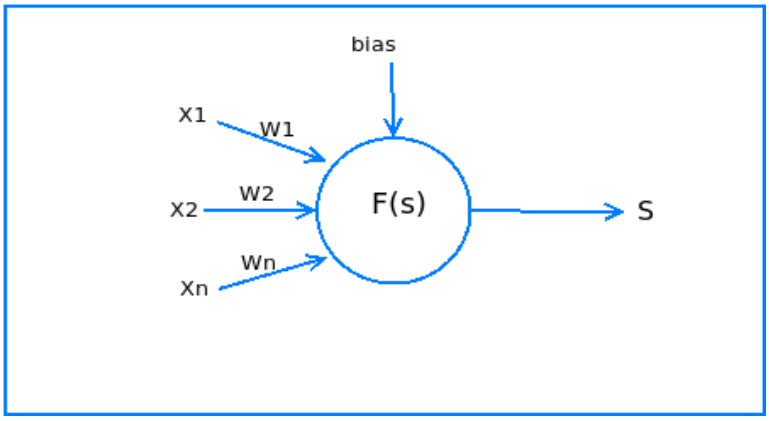


Figura 1.4. Modelo de McCulloch y Pits.

La salida estará dada por la ecuación numero 2: $net = \sum_{i=0}^{n} W_i X_i - \theta$

La cual será activada por la función escalón.

$$f(s) \begin{cases} 1 \text{ si } net > 0 \\ 0 \text{ si } net \leq 0 \end{cases}$$

Se puede notar que la neurona tiene una entrada extra (bias) ya que puede suceder que al re-alimentar, el resultado sea cero. Entonces "se revive a la neurona" (el bias formalmente es igual a la unidad).

A esta información se le debe agregar el concepto de factor de aprendizaje (λ), variable que definirá la velocidad con la que la neurona aprende, esta variable deberá fijarse entre 0 y 1.

Algoritmo del perceptrón:

1. Colocar los Wś de forma aleatoria
2. Colocar el bias (Θ) de forma aleatoria
3. Calcular S
4. Si S es distinto a la salida deseada (Y) $S \neq Y$
    4.1. Calcular el error $e = (Y - S)$
    4.2. Calcular incrementos en $\Delta\theta = -\lambda e$
    4.3. Calcular incrementos en $\Delta W_i = (\lambda)(e)(X_i)$
    4.4. Calcular nuevo bias $\theta_1 = \theta + \Delta\theta$
    4.5. Calcular nuevos pesos $W_{i1} = Wi + \Delta Wi$
5. Volver a 3.

De esta manera es como la neurona aprende, a cada ciclo que repita en busca de los pesos adecuados, se le conoce como época.

Con el perceptrón se pueden representar funciones booleanas tales como las AND, OR, NOR, NAND, pero desafortunadamente algunas funciones no pueden ser representadas con un simple perceptrón como es el caso de la función XOR, debido a que ésta es linealmente no separable.

Se considera linealmente separable cuando con una línea recta puede dividir a los ceros de los unos, este problema es el principal limitante del perceptrón.

Problemas linealmente separables

Tomando en cuenta la ecuación numero 4 y como se menciono anteriormente, la salida del perceptrón es activada por la función escalón, observar figura 1.5.

$$net = \sum_{i=0}^{n} W_i X_i = W_0 X_0 + W_1 X_1 + W_2 X_2 \quad \ldots \quad (4)$$

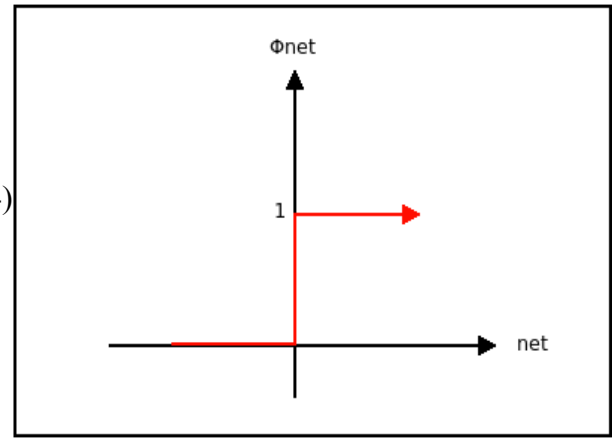


Figura 1.5. Función Escalón.

A modo de poder dividir a los ceros de los unos como se muestra en la figura 1.6 igualamos el net a cero (ecuación numero 5) y despejamos a X2 (ecuación numero 6).

$$0 = W_0 X_0 + W_1 X_1 + W_2 X_2 \ldots (5)$$

$$X_2 = \frac{-W_0}{W_2} - \frac{W_1}{W_2} X_1 \ldots (6)$$

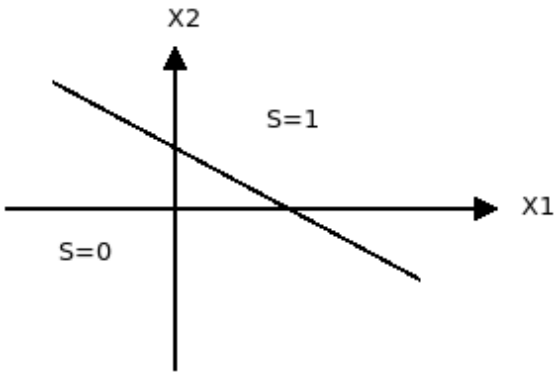
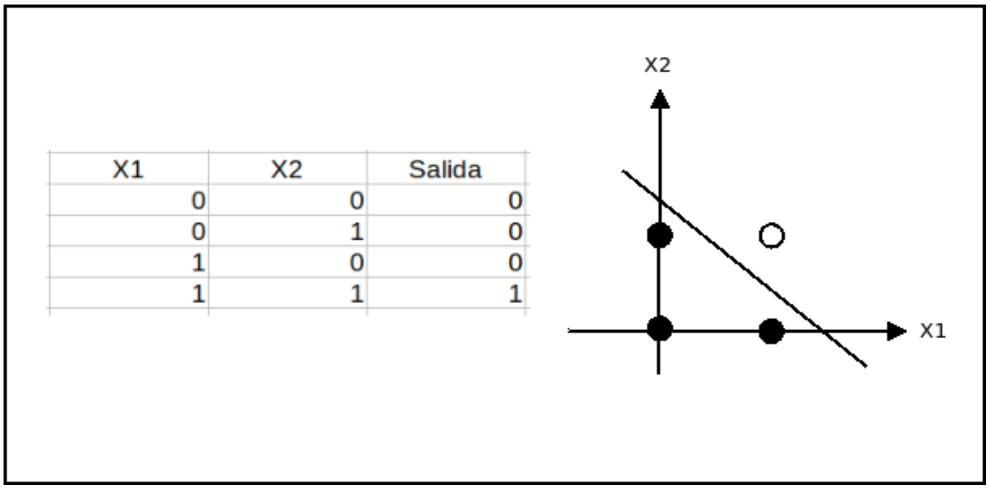


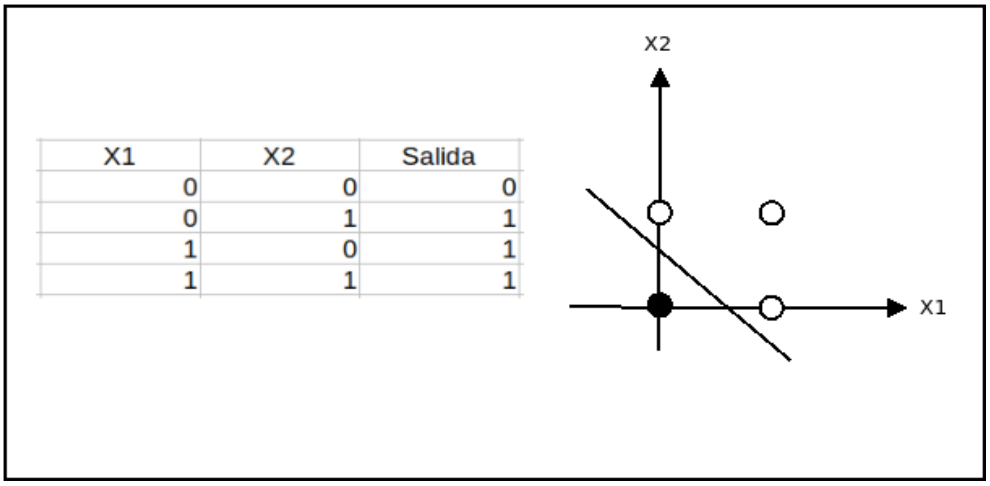
Figura 1.6. Función linealmente separable.

En las figuras 1.7 y 1.8 se muestran las funciones AND y OR respectivamente, se puede observar que estas son separables linealmente, en consecuencia, pueden ser resueltas por el perceptrón.



X1	X2	Salida
0	0	0
0	1	0
1	0	0
1	1	1

Figura 1.7. Función AND.



X1	X2	Salida
0	0	0
0	1	1
1	0	1
1	1	1

Figura 1.8. Función OR.

Por otra parte como se puede ver en la figura 1.9 la función XOR no es linealmente separable y por lo tanto el perceptrón no la puede resolver.

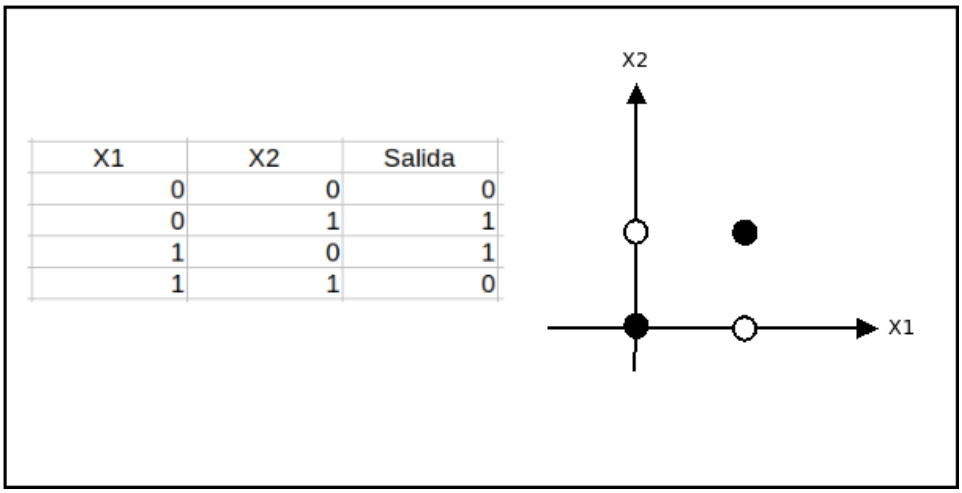


Figura 1.9. Función XOR.

Fue Marvin Lee Minsky y Seymur Papert (1969) quienes rompen con el Perceptrón de Rosenblant al demostrar matemáticamente que no resolvía problemas relativamente fáciles, tales como el aprendizaje de una función no-lineal. Esto demuestra que el perceptrón era débil y las Redes Neuronales pierden fuerza, retomándose hasta 1982. [8]

1.2.3 Redes multicapa y algoritmo backpropagation

Después de la publicación de Minsky y Papert, donde exhiben las limitaciones del perceptrón, hubo un desinterés en el desarrollo de las RNA. Fue hasta 1986 cuando David Rumelhart, Georfrey Hinton y Ronald Williams en el libro Paralell Distributed Processing introdujeron el algoritmo de backpropagation en una red neuronal multicapa.

Las redes neuronales multicapa (RNM) están conformadas por un conjunto de neuronas artificiales interconectadas.

Las neuronas de la red se encuentran distribuidas en diferentes capas de neuronas, de tal manera que las neuronas de una capa están conectadas con las de la capa siguiente a las que pueden enviar información.

Se pueden distinguir tres tipos de capas:

1. De entrada: recibe la información externa a la red.
2. Ocultas: encargadas de realizar el trabajo de la red.
3. De salida: transfieren información de la red hacia el exterior.

Aunque el diseño de una RNM depende de la aplicación, hay ciertos puntos importantes a considerar.

a) Con respecto a las capas:

- Entre más capas ocultas el entrenamiento es más lento.
- Entre más capas ocultas el número de mínimos locales se incrementa.[5]
- Cada capa adicional, hace el gradiente de error más inestable.

b) Con respecto a las neuronas:

- Entre más neuronas, más tiempo de entrenamiento.
- Muchas neuronas en la capa oculta puede causar overfitting (sobre entrenamiento).
- Pocas neuronas puede causar underfitting (no resuelve el problema).

5 Errores que de inicio la red no contempla, en la curva de aprendizaje se encuentran "atrás" del error inicial.

En la figura 1.10 se muestra el esquema básico de una RNM.

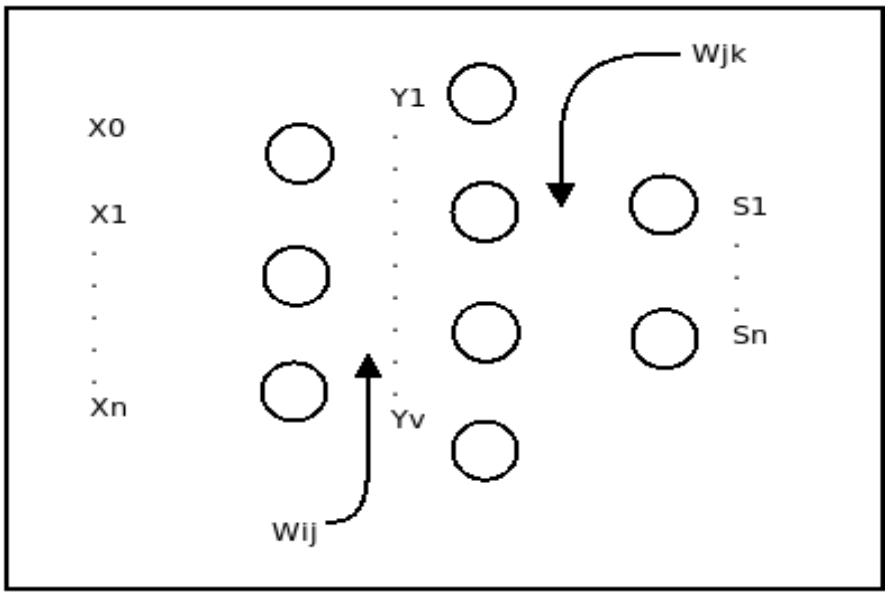


Figura 1.10. Esquema básico de las Redes Neuronales Multicapa.

Algoritmo Backpropagation

Este algoritmo consta de dos fases la primera de ellas es el feedforward y la segunda el backpropagation.

En la primera fase se le presenta a la red un vector de entrada a través de sus neuronas de entrada y dirigida a cada una de las neuronas $_j$ de la capa intermedia, cada una de ellas al recibir esas entradas calcula su suma, pondera el net y pasa por la función de activación (ecuaciones 7 y 8), generando la salida de esa capa, que a su vez sera la entrada para cada neurona de la capa de salida (ecuaciones 9 y 10).

$$netj = \sum_{i=0}^{n} W_{ij} X_i \ldots (7)$$

$$Yj = \varphi(netj) \ldots (8)$$

Entrada

$$netk = \sum_{j=0}^{v} W_{jk} Y_j \ldots (9)$$

$$SK = \varphi(netK) \ldots (10)$$

Salida

φ(x) : función respuesta.

$$\varphi(x) = \frac{1}{1+e^{-\beta x}} \ldots (11)$$

β = Parámetro que indica que tan abrupta es la subida.

Para este caso la función de activación es la función sigmoide, figura 1.11.

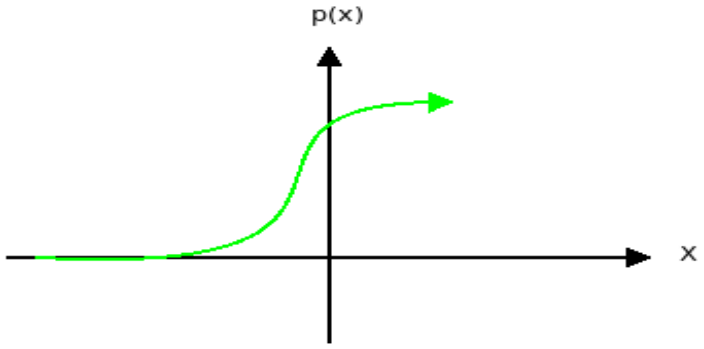


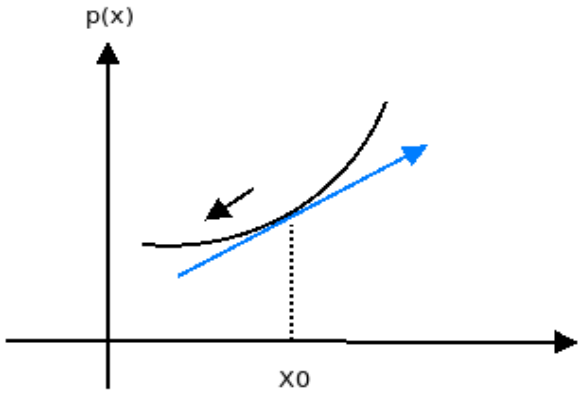
Figura 1.11. Función Sigmoide.

La segunda fase comienza con la comparación de la salida real de la red con la salida deseada calculando el error para cada neurona $_j$ de salida.

Función error:

$$E = \frac{1}{2} \sum_{entrada} \sum_{salida} (Sk(l) - \eta k(l))^2 \ldots (12)$$

Bajo este algoritmo, la consigna para lograr un aprendizaje es hacer mínima la función error. Es por eso que hay que dar un paso en dirección opuesta a la derivada. Esto nos asegurara "bajar" en la curva acercándonos al mínimo más cercano como se muestra en la figura 1.12.



$$x^{new} = X_0 - C\left(\frac{df}{dx}\right)X_0 \ldots (13)$$

Figura 1.12. Gradiente descendente.

Para minimizar la función E (error) se aplica el método del gradiente descendente.

$$W_{jk}^{new} = W_{jk}^{old} + C\sum_{ejmplo} S_k^l Y_j^l \ldots (14)$$

$$S_k^l = \left(n_k^l - S_k^l\right) * \varphi'(netk) \ldots (14')$$

$$W_{ij}^{new} = W_{ij}^{old} + C\sum_{ejemplo} S_j^l X_i^l \ldots (15)$$

$$S_j^l = \left(\sum_k W_{jk} S_k\right) * \varphi'(netj) \ldots (15')$$

La función activadora se deriva, ya que de esta manera se fuerza al error a una corrección más grande, cuando el net queda cerca de la zona de pendiente máxima de la curva sigmoidal.

Metodología para la programación e implementación del backpropagation

1. Inicializar los W's, se les asigna un valor al azar entre -1 y 1
2. Bucle sobre el vector de entrada

3. Aplicar el vector de entrada en la capa de entrada

4. Calcular el valor de las neuronas ocultas

$$Y_j(l) = \varphi\left(\sum_{i=0}^{n} W_{ij} X_i(l)\right)$$

5. Calcular el valor de las neuronas de salida

$$S_k(l) = \varphi\left(\sum_{j=0}^{n} W_{jk} Y_k(l)\right)$$

6. Calcular los errores en la capa de salida

$$S_k{}^l = \left(n_k^l - S_k^l\right) * \varphi'(netk)$$

7. Calcular los errores en la capa oculta

$$S_j{}^l = \left(\sum_k W_{jk} S_k\right) * \varphi'(netj)$$

8. Calcular los cambios en los W's pero, no se aplican sino que se guardan en un acumulador

$$\Delta W_{jk} = \Delta W_{jk} + c\, S_k$$

$$\Delta W_{ij} = \Delta W_{ij} + c\, S_j$$

9. Cerrar el bucle sobre el vector de entrada

10. Aplicar los cambios en los acoplamientos

$$W_{jk}{}^{new} = W_{jk}{}^{old} + \Delta W_{jk}$$

$$W_{ij}{}^{new} = W_{ij} + \Delta W_{ij}$$

Este algoritmo marcó un parte aguas en el desarrollo de las RNA, se retomaron investigaciones de manera mucho mas formal, hasta llegar a todo lo que conocemos hoy en día.

2 Sistema embebido en un solo circuito

Dado que el objetivo principal es diseñar e implementar una red neuronal en un sistema embebido (SE), es necesario adentrarse en los conceptos que esto engloba.

Dicho SE es una tarjeta de desarrollo llamada ZYBO Z7-10 la cual cuenta con un procesador ARM y una FPGA, todo esto será descrito en este capitulo.

Un SE es un sistema de computación en el cual la mayoría de sus componentes están en una placa, en la parte central se encuentra el "cerebro" del sistema, un chip que comúnmente es un procesador, aunque no necesariamente, el cual aporta la capacidad de cómputo del sistema. Dicho chip cuenta con interfaces de entrada/salida que permiten la conexión con el mundo exterior, los SE también se caracterizan por ser empleados en sistemas en tiempo real, donde deberán reaccionar a estímulos de su entorno a partir de sensores, recogiendo la información de estos sensores para procesarlos y realizar una tarea específica.

Se dice que un sistema trabaja en tiempo real si las respuestas de este son correctas y se dan en un intervalo de tiempo determinado.

Los SE están diseñados para cumplir funciones muy especificas, al contrario de lo que sucede con las computadoras de propósito general que están diseñadas para cumplir un amplio rango de necesidades, como una computadora personal.

Todo SE debe contar con cierta cantidad de memoria, pues el código debe alojarse en algún lado, dicha memoria puede ser RAM o ROM y puede encontrarse dentro del mismo chip.

Un SE consiste en un software y un hardware que están específicamente diseñados y optimizados para resolver un problema concreto eficientemente. [9]

2.1 Procesadores ARM

Como se mencionó en el apartado anterior, los sistemas embebidos, en su mayoría, constan en su parte central de un procesador.

Para el correcto funcionamiento del procesador, este necesita conectarse con los siguientes elementos: memoria RAM para contener los datos que se usan mientras el programa se

ejecuta, memoria flash para mantener el programa, lineas de entrada/salida para la comunicación con el mundo exterior y diversos módulos para el control de periféricos, como convertidores analógico-digital, digital-analógico, temporizadores, entre otros.

Un procesador puede ser usado para múltiples aplicaciones, es por eso que en el mercado existe una gran variedad de opciones, pues de lo contrario todos sus recursos tendrían que ser potenciados (mayor capacidad de almacenamiento, un numero muy alto tanto en las lineas de entrada/salida como en los módulos para el control de periféricos), esto generaría precios inaccesibles.

El fabricante oferta múltiples opciones, es posible seleccionar entre la capacidad de memoria, el número de líneas de entrada/salida, cantidad y potencia de los elementos auxiliares, pero quizá la clasificación más importante sea entre procesadores de 8 bits, 16 bits, 32 bits, etc.

Entre los procesadores de 32 bits predomina los ARM. ARM es una arquitectura que fue diseñada por ARM Holdings para permitir implementaciones de tamaño muy reducido y de alto rendimiento. Estas arquitecturas tan simples permiten tener dispositivos con muy bajo consumo de energía, se caracterizan por ser de tipo RISC (Computadoras con un Conjunto de Instrucciones Reducido).

Arquitectura RISC

- Tamaño fijo de instrucciones reducidas, con pocos modos de direccionamiento.
- Modelo de conjunto de instrucciones cargar-almacenar, donde solo dichas instrucciones acceden a memoria y el resto de operaciones se procesan a través de registros, separadas de las instrucciones que acceden a memoria.
- Ausencia de microcódigo.

Estas características simplifican enormemente el diseño del procesador y permiten organizar la arquitectura de tal manera que aumenta el rendimiento de los dispositivos. [10]

Por las características ya mencionadas, los fabricantes de procesadores así como las empresas de telefonía móvil, electrodomésticos, automóviles, entre otras más, han usado esta arquitectura para el desarrollo de sus dispositivos.

Como resultado de ésto se han convertido en los dominantes dentro del mercado de la electrónica móvil e integrada.

Dado que esta arquitectura empezó a cubrir una amplia gama de tecnologías, se dividen en tres familias:

- Cortex A: se caracterizan por poder soportar un sistema operativo dentro de sí. Es el cortex que utilizan los teléfonos celulares.
- Cortex B: su prioridad son aplicaciones en tiempo real, dominan en el mercado automotriz.
- Cortex M: son para el uso de microcontroladores.

2.2 Dispositivos FPGA

La lógica programable tiene su fundamento en la ley de Shannon la cual dice que cualquier función booleana puede expresarse mediante la suma de productos, bajo este concepto se crearon los primeros arreglos de matrices AND y OR que se pueden programar para conseguir funciones lógicas especificas.

Los primeros dispositivos, dedicados a implementar funciones lógicas mediante la programación, fueron los siguientes:

PLA (Programmable Logic Array): matriz AND y OR programables.

PAL (Programmable Array Logic): matriz AND programable y OR fija.

Son conocidos como PLDs (Dispositivos Lógicos Programables) aunque tienen limitaciones, entonces se le añadió algunos flip-flops a la salida del plano OR para formar una macro celda.

En forma breve se puede decir que varias macro celdas forman un SPLD (Dispositivo Lógico Programable Simple), varios SPLD forman un CPLD (Dispositivo Lógico Programable Complejo) y varios CPLD forman lo que hoy conocemos como FPGA.

FPGA (Arreglo Genérico Programable en Campo) es un dispositivo lógico programable que soporta la implementación de circuitos lógicos complejos, son diferentes a los SPLD y CPLD ya que no cuentan con arreglos de matrices AND y OR. En lugar de esto las FPGAs proporcionan bloques de lógica para la implementación de las funciones que requiera. La estructura general de una FPGA consta de tres principales tipos de recursos: bloques de lógica, bloques de Entrada/Salida y la interconexión de cables y switches. Los bloques de lógica están hechos de una matriz de dos dimensiones y las interconexiones de cables están organizadas como un canal de enrutamiento tanto vertical como horizontal entre las filas y las columnas de los bloques de lógica. El canal de enrutamiento contiene cables y switches programables, para conectar con los bloques de lógica de igual manera se conectan los bloques de entrada y salida. En la figura 2.1 se muestra la estructura genera de un FPGA.

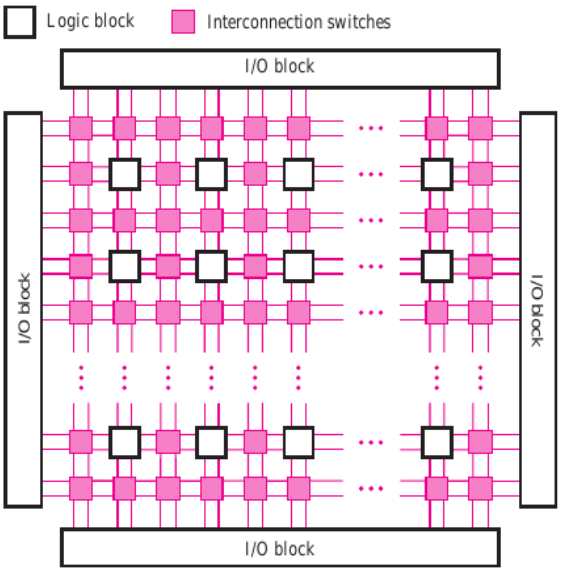
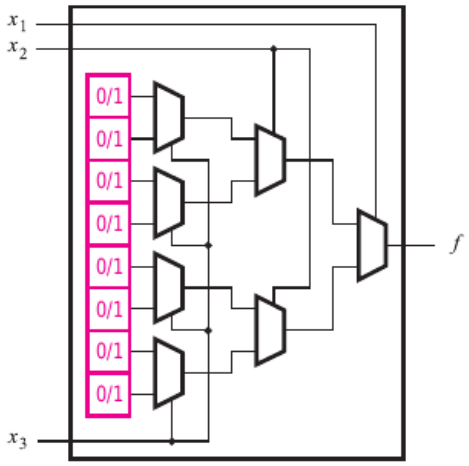


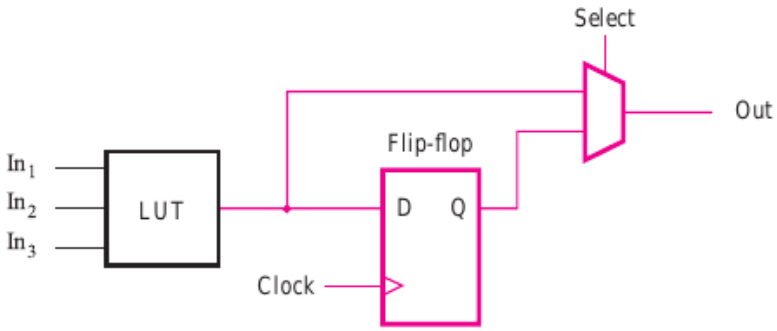
Figura 2.1. Estructura general FPGA.[6]

6 S. D. Brown y Z. G. Vranesic, Fundamentals of digital logic with VHDL design, 3rd ed. New York, NY: McGraw-Hill, 2009.

A los bloques de lógica se les conoce como LUT (lookup table), estos contienen pequeñas memorias que son usadas para implementar las funciones lógicas, cada memoria es capaz de mantener un valor lógico (0 o 1), el valor almacenado será la salida de celda. Se pueden crear LUTs de varios tamaños, el tamaño será definido por el número de entradas. Cada LUT a su salida tiene un circuito extra, el cual es igual a las macro celdas colocadas en los PAL. En la figura 2.2 se muestra la estructura de un pequeño LUT.



a) Celda de un LUT.



b) Circuito extra a la salida del LUT.

Figura 2.2. Arquitectura básica LUT .[7]

7 S. D. Brown y Z. G. Vranesic, Fundamentals of digital logic with VHDL design, 3rd ed. New York, NY: McGraw-Hill, 2009.

Cuando un circuito es implementado en un FPGA, los bloques de lógica son programados para realizar las funciones necesarias, de igual manera los canales de enrutamiento para hacer las conexiones requeridas. Los LUTs en FPGA son volátiles, esto significa que al apagar el sistema se borraran todas las conexiones y funciones implementadas, la solución a esto fue agregar una pequeña memoria (normalmente una EEPROM).

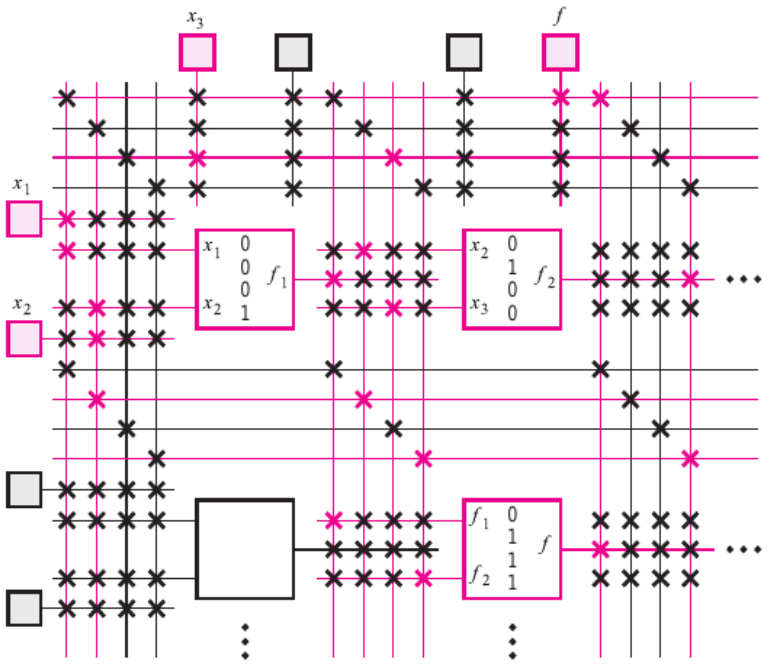


Figura 2.3. FPGA programado. [8]

Hoy en día los FPGAs son usadas en diversas aplicaciones, desde productos básicos como reproductores de DVDs, televisiones, equipo de cómputo, hasta aplicaciones en bioinformática, criptografía, imágenes médicas, visión por computadora, sistemas de audio y video, entre otras. [11]

Uno de los fabricantes más importantes de estos dispositivos es Xilinx[9], el cual cuenta con una amplia gama de FPGAs, siendo la serie ZYNQ una de las más importantes, pues integra la

8 S. D. Brown y Z. G. Vranesic, Fundamentals of digital logic with VHDL design, 3rd ed. New York, NY: McGraw-Hill, 2009.

9 Empresa estadounidense, su principal función es desarrollar y proveer dispositivos lógicos programables. Inventor del FPGA y sistemas embebidos en un solo circuito.

programación de software con la programación en hardware (ARM + FPGA), con lo que se conoce como sistema en un circuito (SoC), permitiendo una gran flexibilidad de diseño.

2.3 Zynq-7000 Sistema en un circuito todo programable

Los dispositivos Zynq son la nueva generación de los sistemas en un solo chip (SoC). Están diseñados para brindar una gran flexibilidad a la hora de trabajar con ellos y así poder abarcar una amplia variedad de aplicaciones.

Estos dispositivos combinan un procesador de dos núcleos ARM Cortex A-9 con una FPGA. En Zynq el ARM Cortex A-9 es un procesador, capaz de correr un sistema operativo completo, como puede ser Linux, mientras que la lógica programable está basada en la arquitectura FPGA de la serie 7 de Xilinx. La arquitectura es complementada con la interfaz AXI, la cual permite la conexión entre ambas partes.

2.3.1 Sistema en un solo circuito

De una manera general se puede decir que un SoC es un dispositivo usado para implementar un sistema completo. En el pasado, el término SoC servía para referirse a los ASIC (Circuito Integrado de Aplicación Específica).

Los SoC pueden combinar todos los aspectos de un sistema digital: procesamiento, lógica de alta velocidad, interfaces, memorias, etc. Todas estas funciones también pueden cubrirse mediante dispositivos por separado y combinarlos todos en una tarjeta de circuito impreso (PCB), sin embargo, la solución de un SoC es más barata, dispone de más velocidad, es mucho más segura en cuanto a transmisión de datos, bajo consumo de energía y físicamente ocupa menor espacio. Por estas razones, es preferible el SoC sobre el PCB.

Por otro lado los ASIC basados en SoC también tiene desventajas ya que estos sistemas tienen ausencia de flexibilidad y son solo sustentables cuando se producen en altos volúmenes, que no requieran de futuras actualizaciones, estos dispositivos se pueden ver en los teléfonos inteligentes, PCs y tabletas electrónicas.

Estas limitaciones hacen que los ASIC basados en SoC sean incompatibles con un número muy grande de aplicaciones, la solución a esto durante mucho tiempo fueron las FPGAs. Ahora, Zynq provee una plataforma ideal de implementar SoCs flexibles, Xilinx crea los dispositivos de la serie APSoC (Zynq All Programmable SoC).

Dichos dispositivos se componen de dos partes principales: un sistema de proceso (PS) formado alrededor de un procesador de dos núcleos ARM Cortex A-9 y una lógica programable (PL), la cual es equivalente a una FPGA.

La sección PL es ideal para implementar lógica de alta velocidad, aritmética y subsitemas de flujo de datos, mientras que el PS soporta rutinas de software y/o sistemas operativos, lo que significa que la funcionalidad general de cualquier sistema diseñado puede particionarse adecuadamente entre hardware y software. La unión entre ambos sistemas es gracias a la interfaz AXI (Interfaz de Extensión Avanzada), la cual será descrita mas adelante.

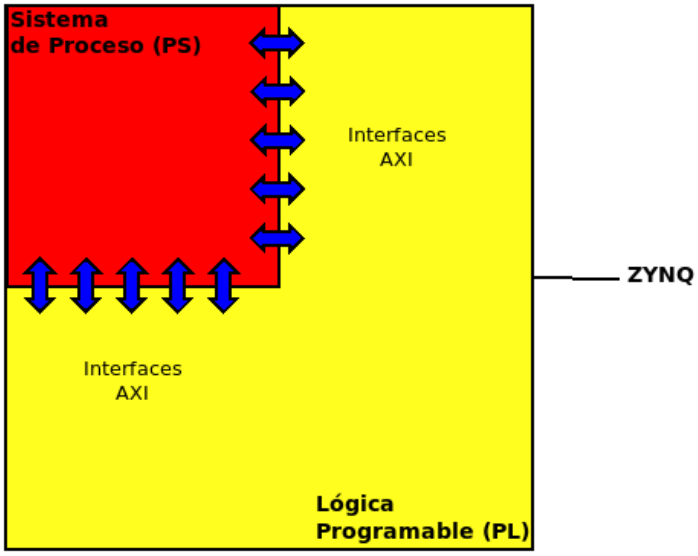


Figura 2.4.Arquitectura básica Zynq. [10]

10 L. H. Crockett, R. a Elliot, y M. a Enderwitz, The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc. Glasgow: Strathclyde Academic Media, 2014.

2.3.2 Arquitectura básica

Es importante conocer la arquitectura básica de un SoC y cómo esta arquitectura es implementada en los APSoC de Xilinx .

De manera general un SoC incorpora un procesador, memorias, y periféricos, junto con los buses que conectan a los elementos entre sí, representando el sistema de hardware.

Este modelo se puede apreciar en la figura 2.5.

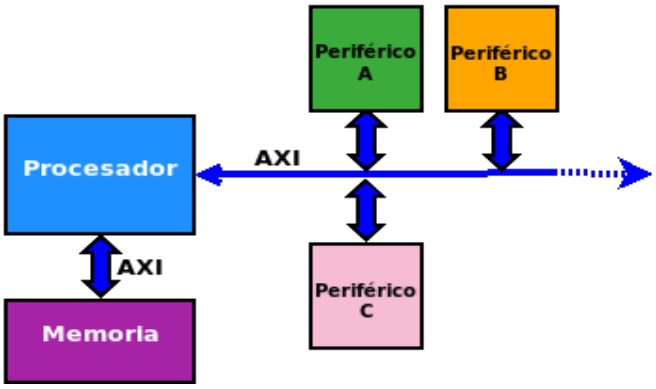


Figura 2.5. Arquitectura simplificada de un SoC.[11]

El procesador es la parte central del sistema de hardware y ahí se alojará el sistema de software. La comunicación entre los elementos del sistema se realizará vía interconexiones, esta interconexión puede ser de distintas maneras.

Los periféricos son componentes que trabajan afuera del procesador, existen diferentes tipos de periféricos, de los cuales se destacan los siguientes:

1. Coprocesadores, son elementos que sustituyen en ciertas tareas al procesador principal.
2. Núcleos, para interactuar con las interfaces externas.
3. Elementos de memoria adicionales.

11 L. H. Crockett, R. a Elliot, y M. a Enderwitz, The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc. Glasgow: Strathclyde Academic Media, 2014.

La arquitectura de los dispositivos Zynq como se deja ver en la figura 2.6 esta basada en esta estructura. El PS tiene una arquitectura fija, alojando el procesador y el sistema de memoria, mientras que el PL es completamente flexible, dando libertad para crear periféricos propios. Como ya se mencionó, la interconexión es implementada gracias a la interfaz AXI, uniendo el PS con el PL .

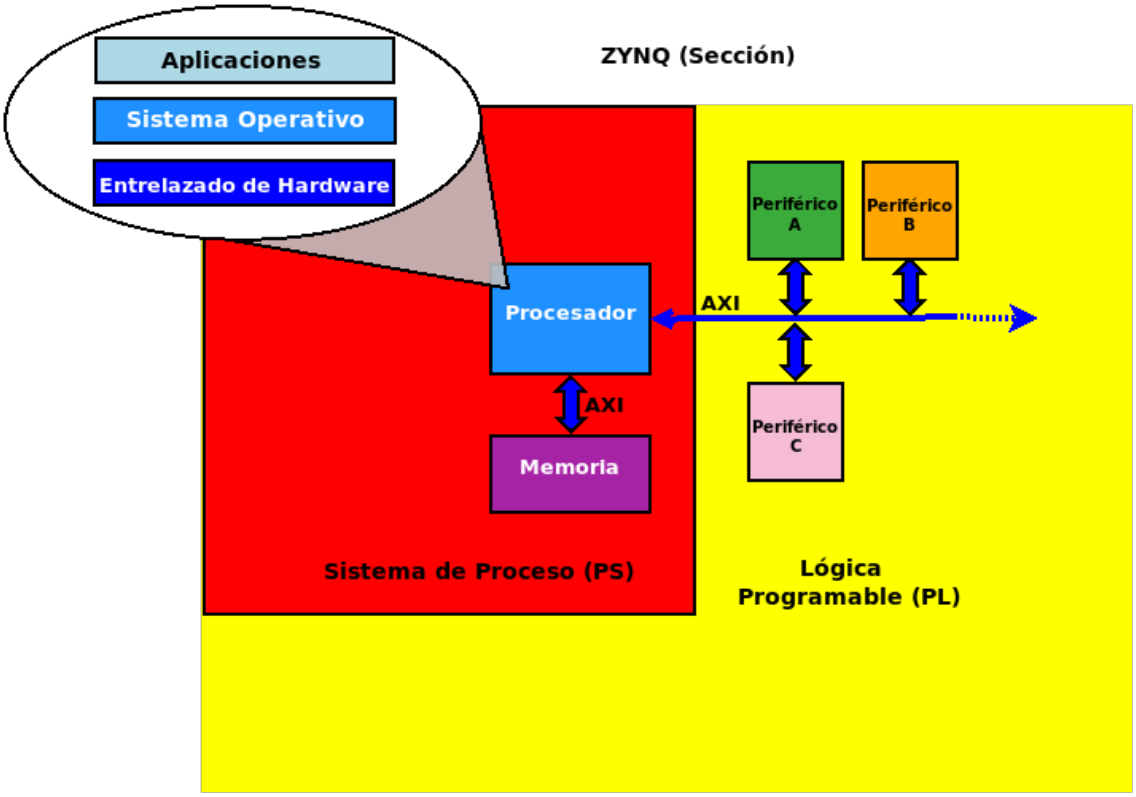


Figura 2.6. Arquitectura básica de un APSoC .[12]

12 L. H. Crockett, R. a Elliot, y M. a Enderwitz, The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc. Glasgow: Strathclyde Academic Media, 2014.

2.3.3 Sistema de proceso

Todos los dispositivos Zynq tienen la misma arquitectura y todos ellos contienen como base del sistema de procesamiento, un procesador de dos núcleos ARM Cortex A-9.

Es importante señalar que el sistema de proceso de Zynq no solamente consta de un procesador ARM, sino también cuenta con un conjunto de recursos de procesamiento asociados, como son la APU (Unidad de Procesamiento de Aplicación), periféricos de interfaz, memoria caché, interfaz de memoria, entre otros recursos más.

En la figura 2.7 se muestra la arquitectura de un PS, donde la APU está resaltada.

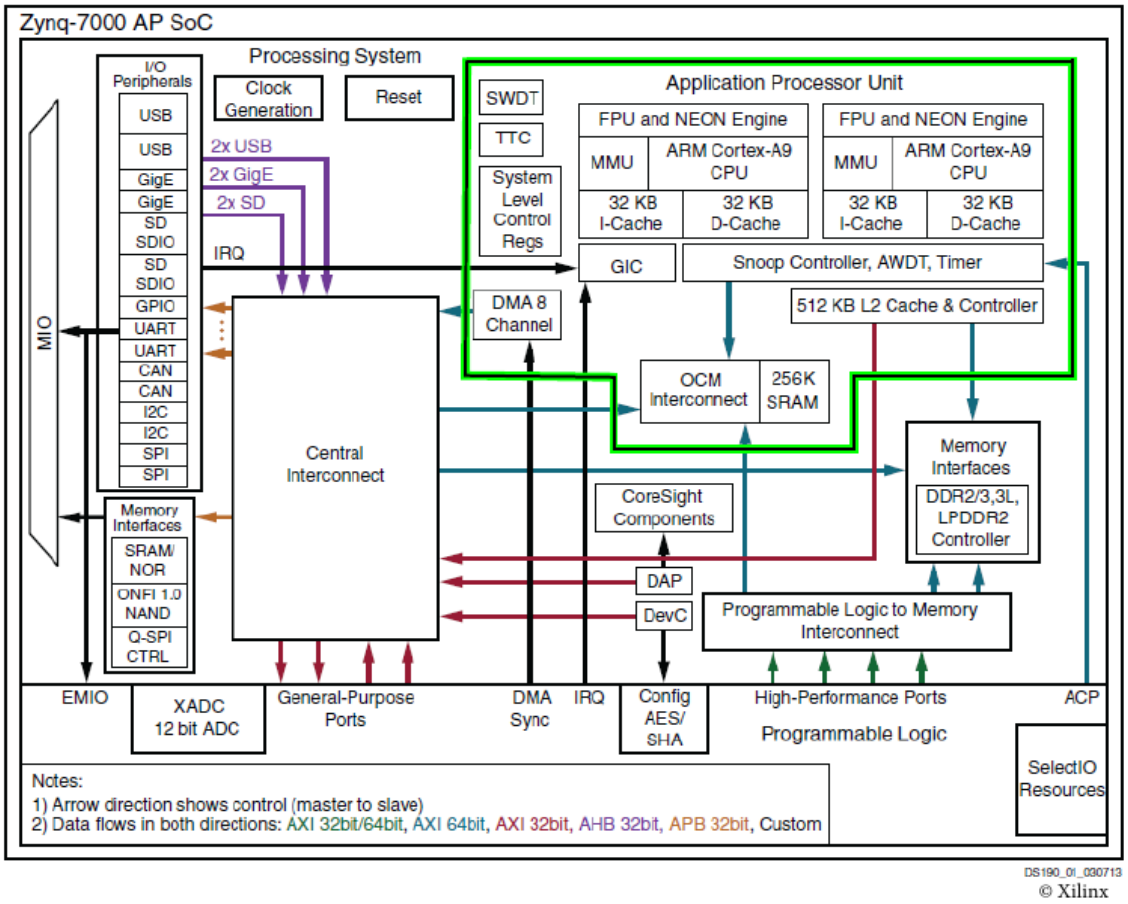


Figura 2.7. Sistema de proceso de Zynq.[13]

13 L. H. Crockett, R. a Elliot, y M. a Enderwitz, The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc. Glasgow: Strathclyde Academic Media, 2014.

La APU como se puede observar en la figura 2.8 se compone principalmente por dos núcleos de procesadores ARM, cada procesador cuenta con: un NEON™ MPE (Motor de Procesamiento de Medios), FPU (Unidad de Punto Flotante), MMU (Unidad de Gestión de Memoria) y un nivel 1 de memoria caché (dividida en dos para instrucciones y datos). La APU también contiene un segundo nivel de memoria caché, un OCM (Memoria En Chip) y finalmente una SCU (Unidad de Control de Busqueda), esta última funciona como un puente entre los dos niveles de memoria caché y contribuye en la comunicación con el PL.

Cada uno de los dos núcleos, tiene dividido el nivel uno de memoria caché, para datos e instrucciones, ambas son de 32KB, esta estructura genera tiempos de acceso más rápidos, y optimización del comportamiento del procesador.

Adicionalmente los dos procesadores comparten el nivel 2 de caché. Éste es de 512KB, y también cuenta con 256KB para OCM.

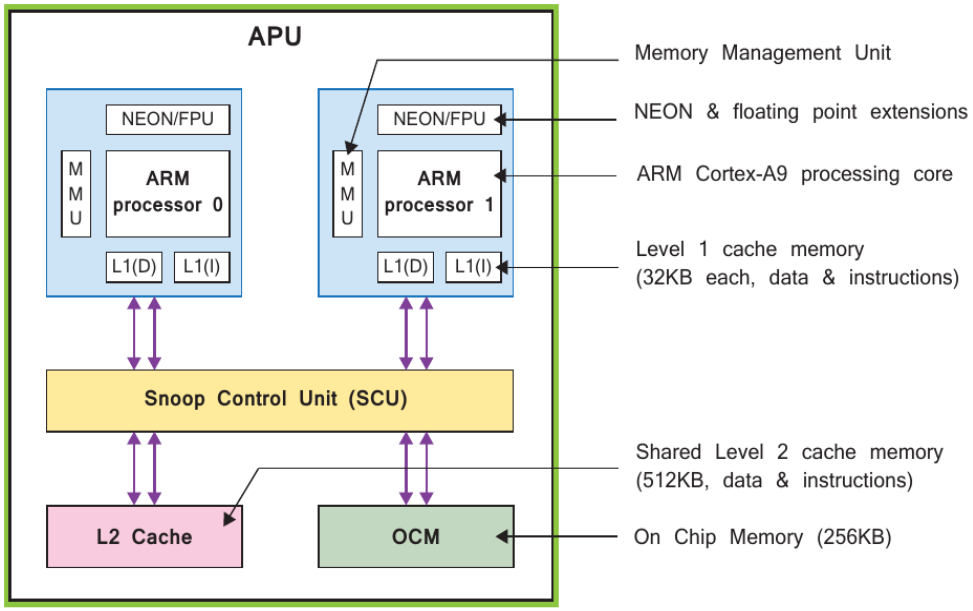


Figura 2.8. Diagrama de bloques de un APU. [14]

14 L. H. Crockett, R. a Elliot, y M. a Enderwitz, The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc. Glasgow: Strathclyde Academic Media, 2014.

El funcionamiento principal del MMU es traducir entre direcciones virtuales y físicas, recordar que las direcciones generadas por los programas en su ejecución no son, necesariamente, aquellas contenidas en la memoria real, ya que las direcciones virtuales suelen seleccionarse dentro de un número mucho mayor que las direcciones disponibles dentro de la memoria física.

El SCU es de suma importancia para la interfaz entre los dos niveles de memoria cache, es el responsable de mantener coherencia entre el nivel y el nivel dos de memoria cache. También inicializa y controla el acceso del segundo nivel.

En cuanto a programación se refiere, el soporte para instrucciones ARM se proporciona a través de un software llamado Xilinx Software Developmet Kit (SDK), el cual incluye todo lo necesario para el desarrollo de software en el procesador ARM.

Como una función adicional del procesador ARM, se encuentra el NEON, que provee "múltiples elementos de procesamiento" (SIMD), que permite la aceleración estratégica de los datos. Como el término SIMD sugiere, NEON puede aceptar múltiples vectores de entrada sobre los cuales se realizarán operaciones para generar simultáneamente una salida. Este estilo de computación se adapta bien a aplicaciones de imagen, video y procesamiento de datos en paralelo.

Por último, se hablará de las interfaces que maneja el PS con componentes externos. Esta comunicación se logra principalmente a través de "entradas/salidas multiplexadas" (MIO). Ciertas conexiones también se pueden generar mediante una versión extendida de este componente (EMIO), pasando y generando recursos de entrada/salida con el PL. Una de la ventajas del EMIO es que permite conectar la PS con un bloque implementado en el PL.

La comunicación con periféricos de entrada/salida dispone de ciertos estándares de comunicación y entradas/salidas de propósitos generales (GPIO), que se pueden usar para diferentes propósitos, como pueden ser conectar con botones, LEDs, switches y otros.

La lista de las interfaces de entrada y salida se pueden observar en la imagen 2.9.

Interfaz de Entrada/Salida	Descripción
SPI (x2)	Interfaz Periférica Serial Estándar para comunicaciones seriales basado en una interfaz de 4 pines. Puede ser usado en modo maestro o esclavo.
I2C (x2)	Bus I²C Conforme a las especificaciones del bus I2C, versión 2. Soporta modo maestro y modo esclavo.
CAN (x2)	Red de Área del Controlador Acorde con ISO 118980-1, estándares CAN 2.0A y CAN 2.0B.
UART (x2)	Transmisor-Receptor Asíncrono Universal Interfaz de módem de datos de baja velocidad para comunicación serial.
GPIO	Entradas y Salidas de Propósito General Tiene 4 bancos GPIO, cada uno de 32 bits.
SD (x2)	Para la conexión con una memoria SD.
USB (x2)	Bus Serial Universal Conforme a USB 2.0, puede ser usado como host, dispositivo o flexible (modo OTG, en el cual se puede cambiar entre los modos mencionados.
GigE (x2)	Ethernet Soporta los modos 10Mbps, 100Mbps y 1Gbps.

Figura 2.9. Interfaces entradas/salidas.[15]

15 L. H. Crockett, R. a Elliot, y M. a Enderwitz, The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc. Glasgow: Strathclyde Academic Media, 2014.

2.3.4 Lógica programable

Otra parte principal de la arquitectura Zynq es la lógica programable, basada en los FPGAs Artix 7 y Kintex 7, en la figura 2.10 se muestra su arquitectura básica.

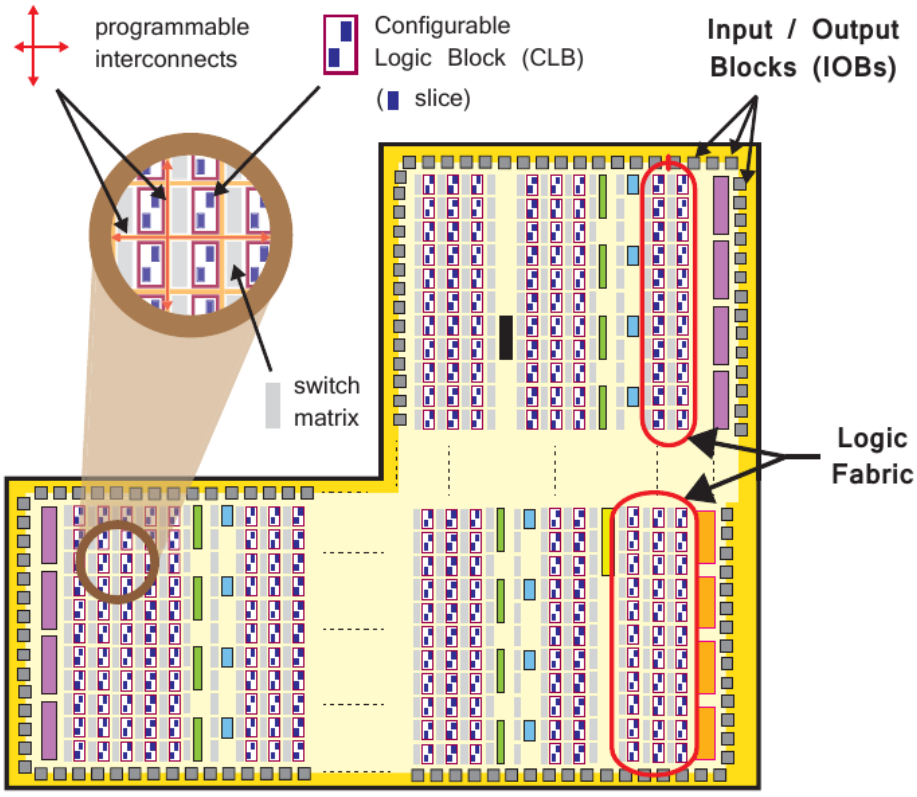


Figura 2.10. Arquitectura básica de un PL.[16]

El PL esta compuesto predominantemente por un FPGA de propósito general, que a su vez está compuesta por pequeñas estructuras lógicas (slices), Bloques de Lógica Configurables (CLB) y bloques de entrada/salida (E/S).

- CLB: son agrupaciones regulares de elementos lógicos que se encuentran en una matriz bidimensional dentro del PL. Cada CLB contiene dos slices y tiene a un lado

16 L. H. Crockett, R. a Elliot, y M. a Enderwitz, The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc. Glasgow: Strathclyde Academic Media, 2014.

una matriz de conmutación, en la figura 2.11 se muestra la configuración básica de un CLB

- Slice: es una subunidad que está dentro del CLB, contiene recursos para implementar lógica secuencial y combinacional. Están compuestos por 4 LUTs y 8 flip-flops.
- LUT: es el encargado de implementar las funciones lógicas, puede combinarse con otros para formar funciones mas complejas, memorias, registros etc.
- Switch Matrix: como se mencionó están a un lado de los CLB y su función es facilitar el enrutamiento para hacer conexiones con otros CLBs.
- Bloques de E/S: estos recursos proveen la interfaz entre el PL y los dispositivos físicos.

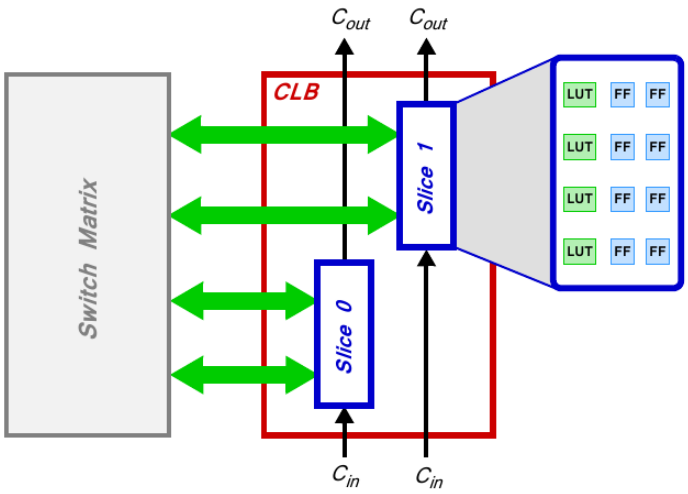


Figura 2.11. Configuración de un CLB.[17]

En adición a la FPGA el PL cuenta con dos componentes de particular interés, como son los bloques de RAM para requerimientos de memoria y el DSP48E para operaciones aritméticas que requieran alta velocidad.

17 L. H. Crockett, R. a Elliot, y M. a Enderwitz, The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc. Glasgow: Strathclyde Academic Media, 2014.

Los bloques de RAM en los Zynq son equivalentes a los contenidos en las FPGAs de la serie 7 de Xilinx. Cada bloque de RAM puede almacenar hasta 36Kb de información, y puede configurarse como un solo bloque de 36Kb o dos independientes de 18Kb, asimismo, si se necesita de más capacidad de almacenamiento, se pueden formar varios bloques de RAM. Los bloques de RAM pueden funcionar a la máxima frecuencia de reloj soportada por el dispositivo.

DSP48E está hecho para poder implementar aritmética de alta velocidad o señales de diferente tamaño de palabra. Principalmente lo compone un pre-sumador/restador, multiplicador y un post-sumador/restador con una unidad lógica. Al igual que los bloques de RAM, puede funcionar a la máxima frecuencia de reloj.

Los bloques de E/S, están organizados en bancos de 50 IOBs. Cada IOB contiene un bloc (pad) que proporciona las conexiones físicas con el mundo exterior. Los bancos de E/S están categorizados como alto rendimiento (HP) o alto rango (HR), soportando una variedad de voltajes y estándares. Las interfaces HP están limitadas para funcionar con 1.8 volts y son usadas generalmente cuando se requiere de alta velocidad. Por otra parte el HR permite voltajes de hasta 3.3 volts y satisface una variedad más amplia de estándares. Para lograr establecer la comunicación de estas interfaces existen módulos ya hechos, embebidos en el PL, éstos son llamados “Hard IP blocks”.

Existen varias interfaces de E/S como puede ser el convertidor analógico digital, relojes, programacion y depuración, entre otros. Pueden estar contenidos en el PL e implementarse por medio de los “Hard IP blocks”.

2.3.5 Estándar AXI

Una de las grandes ventajas de los sistemas Zynq es crear sistemas completos e integrados, ésto se logra usando sus dos partes el PS y el PL. La interfaz AXI (Interfaz de Extensión Avanzada) es el “puente” que se utiliza para conectar ambas partes.

AXI es definido como una interfaz avanzada de conexiones y actualmente se encuentra en la versión AXI4, la cual es parte de ARM AMBA (Arquitectura avanzada de bus para microcontroladores).

AMBA es un estándar desarrollado en 1996, en un principio se hizo con el fin de usarse en microcontroladores, desde ese entonces dicho estándar ha ido evolucionando y ahora ARM lo define como el estándar por defecto para las comunicaciones en un chip. El enfoque ahora está en los SoCs basados en FPGAs, en este caso dispositivos Zynq, de hecho Xilinx contribuyó de una manera muy fuerte en definir AXI4, como la manera más optima de interconectar tecnologías con arquitecturas FPGAs.

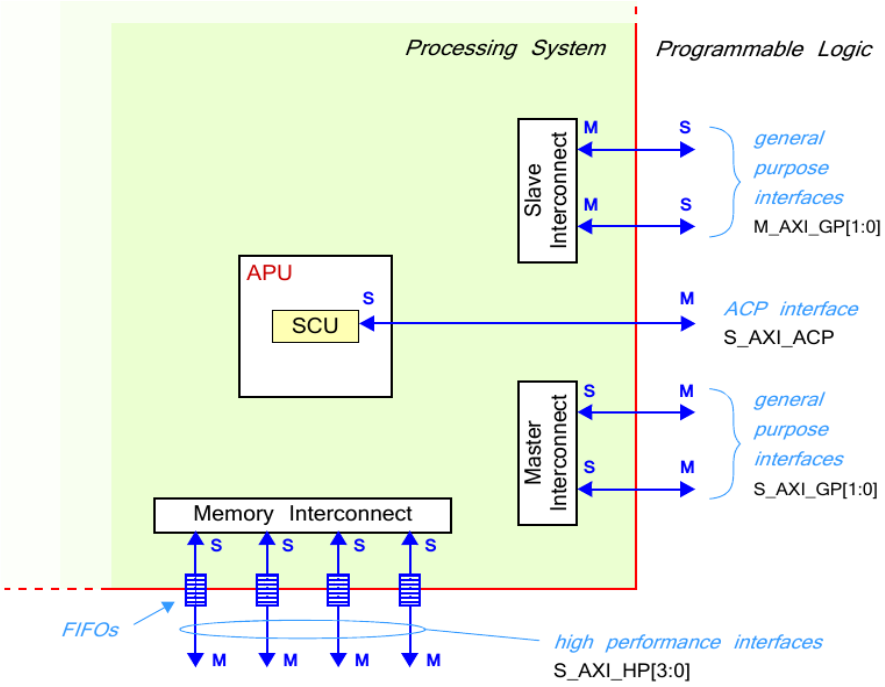
Los buses de AXI pueden ser usados con mucha flexibilidad y en general son usados para conectar el procesador con bloques de IP en sistemas embebidos. Se tienen tres tipos de AXI4, cada uno con diferentes protocolos de buses.

- AXI4 [2]: provee un alto rendimiento, lo que hace es proporcionar una dirección seguida de una transferencia de muchos datos de hasta 256 palabras.
- AXI4-Lite[2]: solo un dato de transferencia por conexión, solo se transfiere una dirección y una palabra sencilla.
- AXI4-Stream[1]: para una alta velocidad en transmisión de datos, pero en este caso no hay mecanismo de direcciones, este tipo de mecanismo funciona directo con el flujo de datos entre fuente y destino.

La principal interfaz entre el PL y PS es con un conjunto de nueve interfaces AXI, cada una compuesta de múltiples canales.

- Interconexiones: una interconexión es un conmutador que maneja y dirige el tráfico entre las interfaces AXI conectadas. Hay varias interconexiones dentro del PS, algunas que están conectadas directamente al PL y otras que son solo para uso interno.
- Interfaz: una conexión punto a punto para pasar datos, direcciones, señales, entre un maestro y esclavo del sistema.

En la figura 2.12 se observa la estructura AXI de las conexiones e interfaces conectadas al PS y PL y una breve descripción de cada una de las interfaces, en donde el maestro y esclavo son indicados.



Nombre de Interfaz	Descripción de interfaz	Maestro	Esclavo
M_AXI_GP0	Propósito General (AXI_GP)	PS	PL
M_AXI_GP1		PS	PL
S_AXI_GP0	Propósito General (AXI_GP)	PL	PS
S_AXI_GP1		PL	PS
S_AXI_ACP	Puerto de Coherencia del Acelerador (ACP)	PL	PS
S_AXI_HP0	Puerto de Alto Rendimiento (AXI_HP) con leer/escribir FIFOs.	PL	PS
S_AXI_HP1		PL	PS
S_AXI_HP2		PL	PS
S_AXI_HP3		PL	PS

Figura 2.12. Estructura de AXI interconexiones e interfaces entre el PL y PS.[18]

18 L. H. Crockett, R. a Elliot, y M. a Enderwitz, The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc. Glasgow: Strathclyde Academic Media, 2014.

Como se puede observar en la figura 2.12 hay diferentes tipos de interfaces AXI, las cuales describiremos a continuación.

- Propósitos Generales AXI: tiene un bus de datos de 32 bits el cual es ideal para una baja y media tasa de comunicación entre el PS-PL. La interfaz es directa y no incluye buffer.
- Puerto de Coherencia del Acelerador: conexión asíncrona ente el PL y el SCU del APU, con un bus de 64 bits.
- Puerto de Alto Rendimiento: las cuatro interfaces AXI de este tipo tienen bufers FIFO (primero en entrar, primero en salir) ya que llega una cantidad considerable de datos, ideal para altas tasas de comunicación, entre el PL y elementos de memoria en el PS, el ancho de datos es de 32 o 64 bits. [12]

2.4 Tarjeta de desarrollo ZYBO Z7-10

En la etapa de investigación se han encontrado distintos dispositivos para implementar sistemas digitales, la tarjeta de ZYBO Z7-10, es una tarjeta de desarrollo de software y circuito digital integrado, pertenece a la familia Xilinx Zynq 7000, dicha familia está basada en la arquitectura Xilinx All Programmable System-on-Chip.

La ZYBO Z7-10 rodea al chip Zynq, con un amplio conjunto de periféricos de multimedia y conectividad, creando un excelente sistema embebido. Dichos periféricos y las características especificas del Zynq serán descritos a continuación.

Es importante mencionar que esta tarjeta de desarrollo tiene dos variantes ZYBO Z-10 y ZYBO Z-20, siendo su principal diferencia la capacidad del chip y el número de periféricos multimedia.

Chip Zynq.

- Procesador de 667 MHz con dos núcleos Cortex-A9.
- Lógica programable: FPGA Artix-7.
- DDR3L control de memoria con 8 canales de DMA.
- Controladores para periféricos con gran ancho de banda: 1G Etherner, USB 2.0, SDIO.
- Controladores para periféricos de bajo ancho de banda: SPI, UART, CAN, I2C.

Memoria.

- 1 Gb DDR3L con 32 bits de bus.
- 16 Mb Quad-SPI flash.
- Ranura para micro SD

Energía.

- Encendido por USB o cualquier fuente externa de 5 volts

USB e Internet.

- Gigabit Ethernet PHY.
- Circuito programador USB-JTAG.
- Puente USB-UART.

Audio y video

- Conexión para cámara Pcam.
- Puertos HDMI de E/S.
- Codificador de audio.

Switches, push-buttons y LEDs.

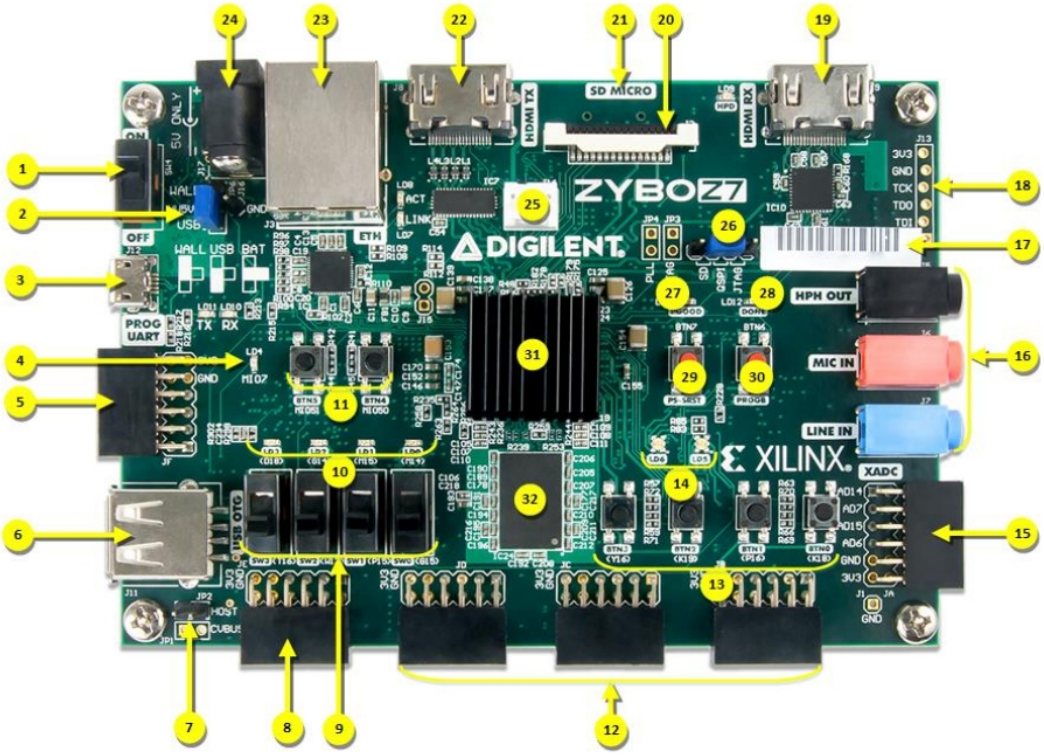
- 6 push-buttons.

- 4 interruptores.
- 5 LEDs.
- 1 LED RGB.

Conexiones para expansión.

- 5 módulos extra de puertos (Pmod).
    - 8 para el procesador E/S.
    - 32 para la FPGA E/S.
    - 4 analógicos de 0 a 1 volt, pares diferenciales para XADC (Convertidor analógico-digital). [13]

En la figura 2.13 se muestra la tarjeta (parte a) y una lista de los elementos (parte b) mencionados.



a) Tarjeta de desarrollo ZYBO Z7-10

Callout	Description	Callout	Description
1	Power Switch	17	Unique MAC address label
2	Power select jumper	18	External JTAG port
3	USB JTAG/UART port	19	HDMI input port
4	MIO User LED	20	Pcam MIPI CSI-2 port
5	MIO Pmod port	21	microSD connector (other side)
6	USB 2.0 Host/OTG port	22	HDMI output port
7	USB Host power enable jumper	23	Ethernet port
8	Standard Pmod port	24	External power supply connector
9	User switches	25	Fan connector (5V, three-wire) *
10	User LEDs	26	Programming mode select jumper
11	MIO User buttons	27	Power supply good LED
12	High-speed Pmod ports *	28	FPGA programming done LED
13	User buttons	29	Processor reset button
14	User RGB LEDs *	30	FPGA clear configuration button
15	XADC Pmod port	31	Zynq-7000
16	Audio codec ports	32	DDR3L Memory

b) Lista de elementos de la tarjeta de desarrollo ZYBO Z-10

Figura 2.13. Tarjeta de desarrollo ZYBO Z-10.[19]

19 “Zybo Z7: Zynq-7000 ARM/FPGA SoC Development Board”, Digilent. [En línea]. Disponible en: https://store.digilentinc.com/zybo-z7-zynq-7000-arm-fpga-soc-development-board/. [Consultado: 04-abr-2019].

3 Diseño e implementación

En este capítulo se realiza una descripción detallada de como se llevó a cabo el proceso de diseño e implementación de una red neuronal multicapa con la tarjeta de desarrollo ZYBO-Z7 .

Como ya se ha explicado, una de las características del sistema embebido ZYNQ es que éste se divide fundamentalmente en dos partes: PS (Procesing System) cuyo elemento principal es un procesador ARM Cortex A9 y PL (Programmable Logic) equivalente a un FPGA Artix-7. Este diseño busca integrar ambas partes para la implementación de una red neuronal artificial multicapa en la cual se puede implementar una aplicación especifica sin necesariamente interactuar con otros dispositivos.

Las herramientas utilizadas para el diseño e implementación de este sistema en la parte PL de la tarjeta ZYBO-Z7 fueron:

- Lenguaje de descripción de hardware: VHDL.
- VIVADO DESIGN SUITE (versión 2018.2)[20]: Entorno de desarrollo de licencia libre para FPGA y SoC desarrollado para dispositivos Xilinx con el cual se configura y describe la parte PL de la tarjeta.

Mientras que para la parte PS:

- Lenguaje de programación: C.
- SDK (versión 2018.2)[21]: Entorno de desarrollo integrado con el software de VIVADO, está dedicado a la programación de PS el cual soporta los lenguajes de programación C o C++.

20 https://www.xilinx.com/products/design-tools/vivado.html

21 https://www.xilinx.com/products/design-tools/embedded-software/sdk.html

3.1 Diseño conceptual (Top Level Design)

El diseño de una red neuronal artificial multicapa, tiene como configuración más sencilla una sola capa oculta, la cual se describirá en este apartado, para posteriormente probar alguna otra configuración y evaluarlas en el protocolo de pruebas.

El sistema embebido con una red neuronal artificial toma un conjunto de datos de entrenamiento previamente cargados para una aplicación específica. Una vez hecho esto el usuario decide a través de un conmutador (switch) de la tarjeta si realizar el entrenamiento de la red o la propagación hacia adelante, en un principio los parámetros de la red son aleatorios, si se decide hacer la propagación hacia adelante sin previo entrenamiento los resultados serán erróneos. En el caso de que el usuario decida primero entrenar la red, se tomará el conjunto de datos de entrenamiento y con éstos se ajustarán los parámetros de la red. Una vez que ha sido entrenada, si el usuario decide realizar una prueba, los resultados serán los esperados ya que los parámetros de la red han sido ajustados.

El primer paso para el diseño del sistema, fue establecer en términos generales las funciones que éste tendría que realizar, las cuales se pueden observar en la figura 3.1:

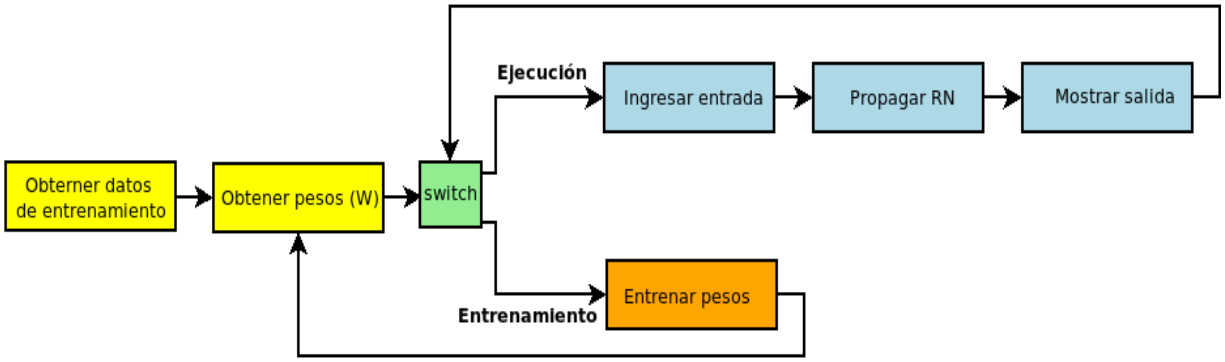


Figura 3.1. Funcionamiento general de la RNA en tarjeta ZYBO-Z7.

Como se mencionó anteriormente, el diseño involucra ambas partes PS y PL de la tarjeta por lo que las funciones anteriores se reparten entre estas dos, de tal manera que el entrenamiento se pueda realizar mediante software en PS y una vez que todos los parámetros de la red neuronal hayan sido ajustados pasen al PL en donde se ejecutará en hardware la propagación

hacia adelante de la red con las nuevas entradas que ingrese el usuario, además se agregan algunas funciones más para la comunicación entre las partes. En la figura 3.2 se pueden ver las funciones entre el PS y PL.

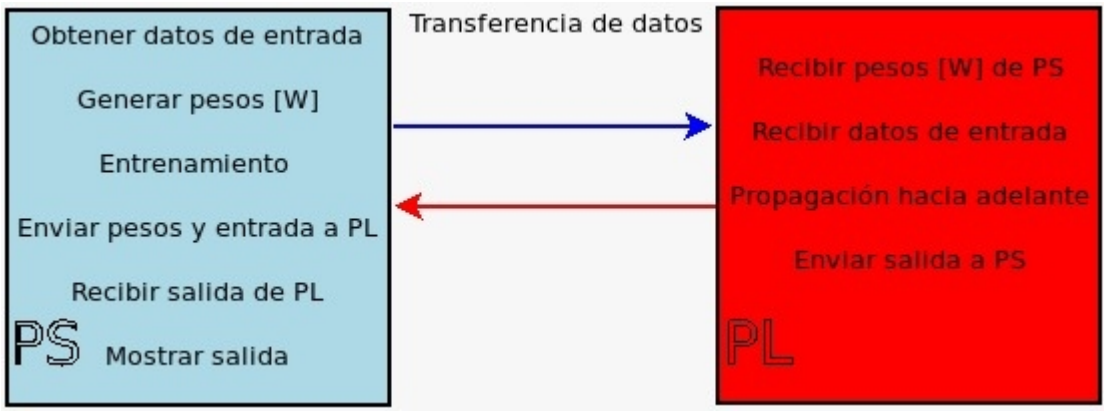


Figura 3.2. Funciones de PS y PL.

Una vez que los pesos han sido ajustados al entrenar la red y PS los ha enviado, los datos quedan guardados en PL, y PS solo se encarga de enviar los nuevos datos de entrada para el modo de prueba.

Ahora que se han definido las funciones y las partes a las que le corresponde cada una, se realiza un diseño de nivel superior (top level design) en donde se representa todo el sistema, en este caso la red neuronal como un bloque dividido entre PS y PL. En la parte de PL habrá varios bloques de hardware en su interior los cuales realizan tareas más especificas y que en conjunto llevarán a cabo las funciones descritas anteriormente. En cuanto PS solo se señalaran los registros con los cuales se comunicará con PL.

En el primer diagrama se observará el diseño en su nivel más alto, posteriormente baja un nivel en el PS y se observa la capa más importante, pues en ella se desarrolla la lógica de la propagación hacia adelante de la red, esta capa es llamada *layer*, se harán instancias de capas (layers) de acuerdo con la cantidad necesaria de capas ocultas. En el mismo nivel el otro bloque lógico de interés es layer_out pues este es el bloque de salida que tendrá la RNA independientemente del número de capas.

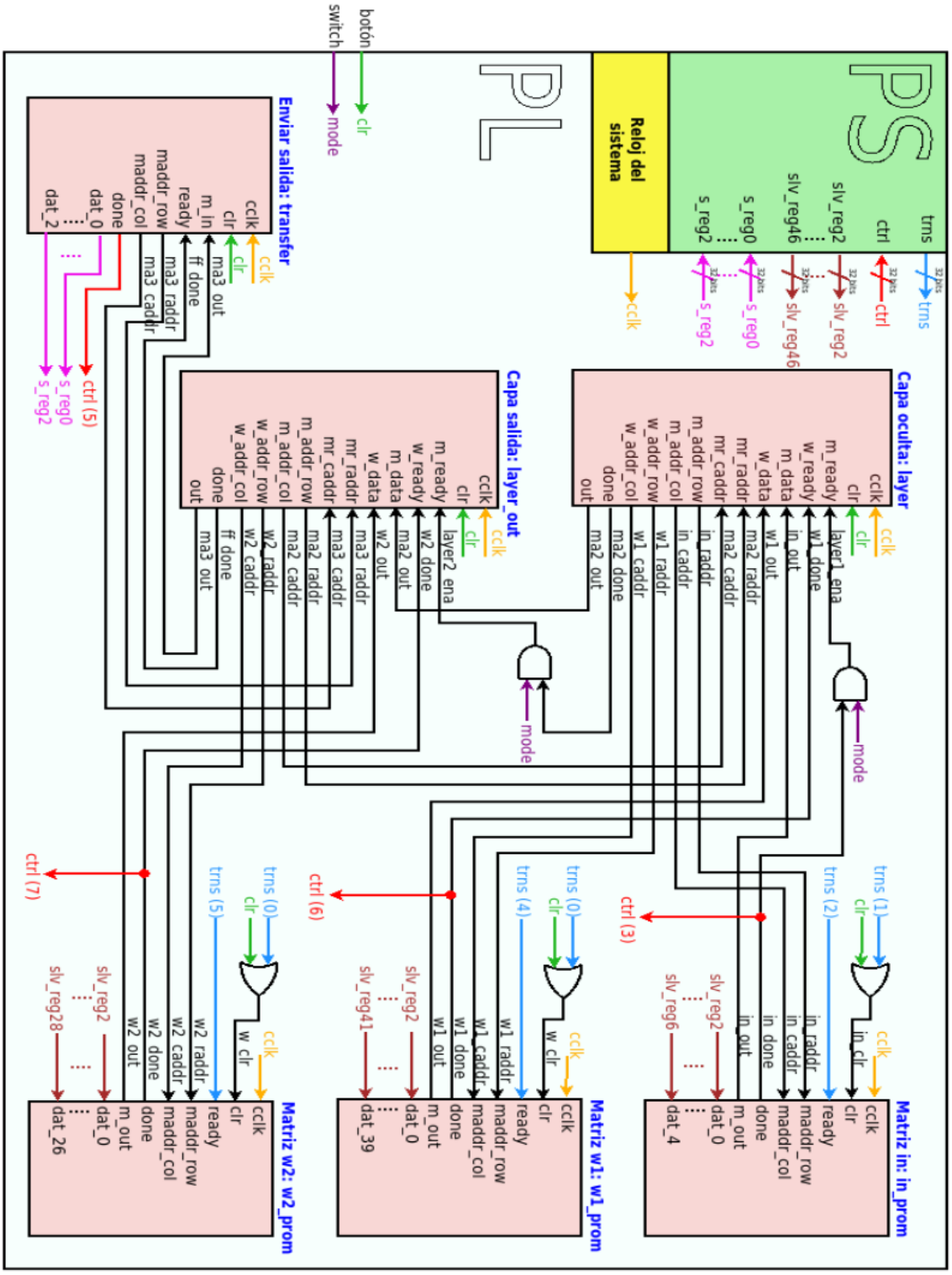


Figura 3.3. Diseño Conceptual.

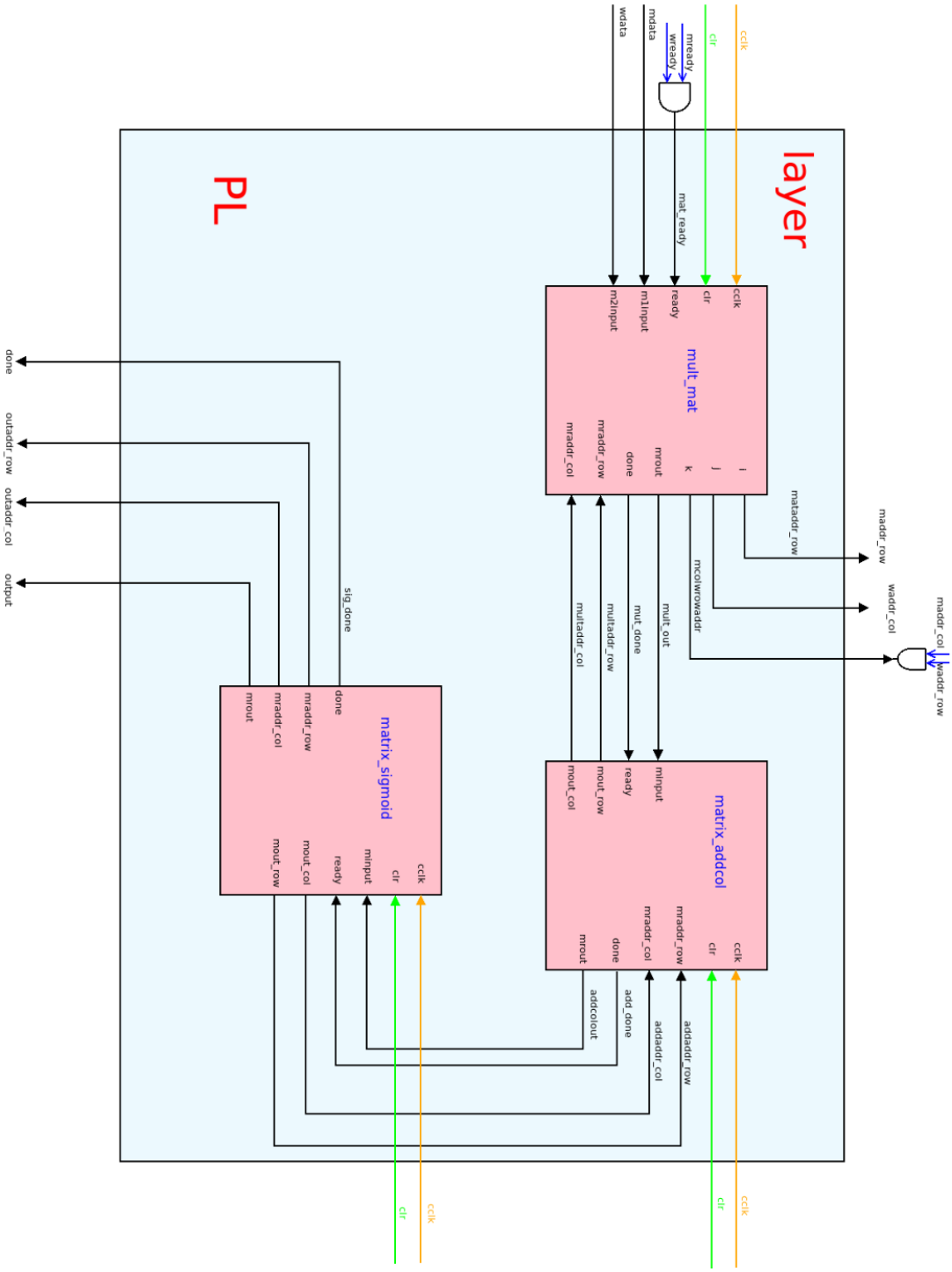


Figura 3.4. Layer (propagación hacia adelante).

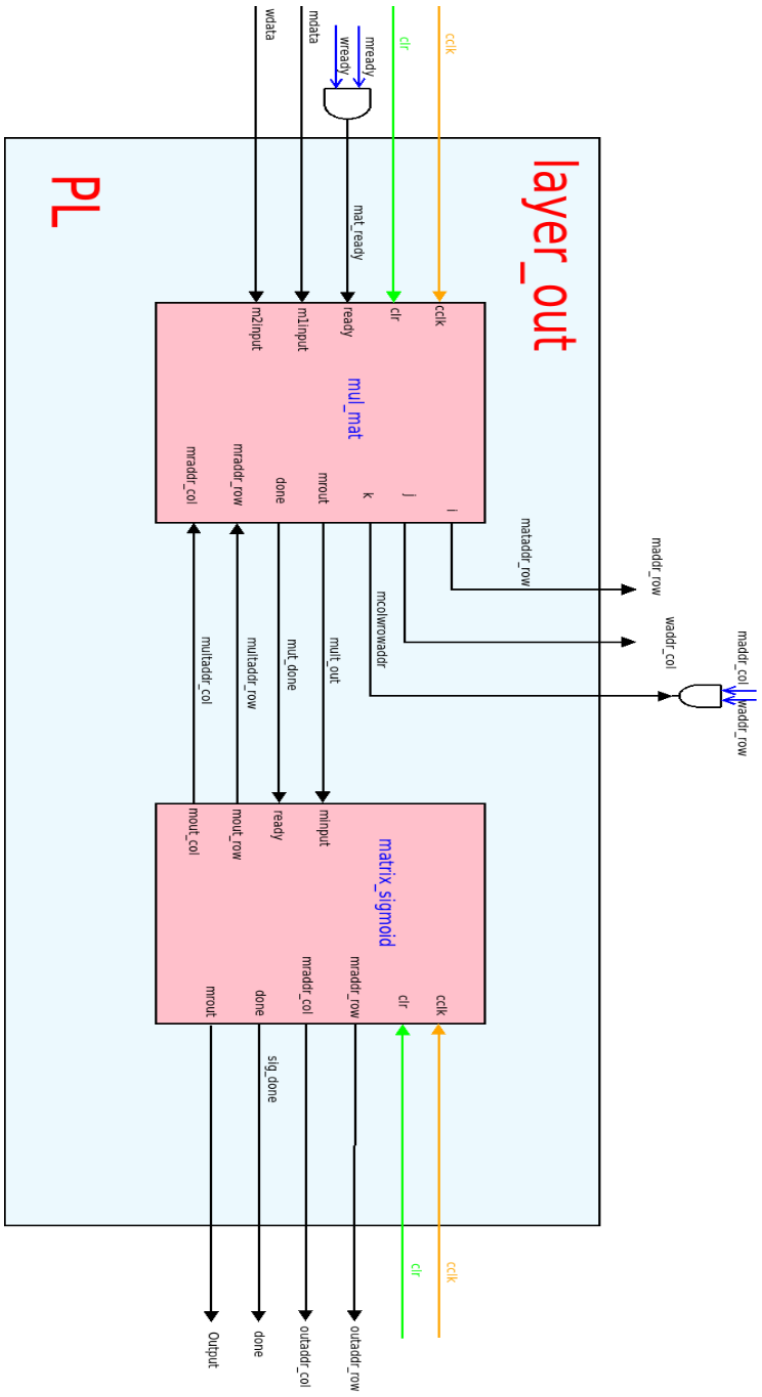


Figura 3.5. Layer out (salida de la red neuronal).

3.2 Pre-procesamiento de datos

Cuando se va a implementar una nueva aplicación en una red neuronal es necesario tener un conjunto de datos de entrenamiento, el cual consta de una lista de ejemplos con valores representativos para dicha aplicación y una lista de destino (target) la cual contiene los valores de las salidas esperadas para cada uno de los ejemplos anteriores. Para poder entrenar la red neuronal no basta con tener estos datos brutos, ya que ésta no podría interpretarlos de manera correcta, por lo tanto, es necesario que se realice un pre-procesamiento, en donde serán convertidos en valores con los que la red pueda trabajar.

Los datos que se pueden presentar para el entrenamiento de una red neuronal pueden ser: datos cuantitativos (como edad, distancia, peso, etc.), datos binarios (como el sexo, que pueden ser masculino o femenino) y los datos categóricos (como una comunidad, que puede ser suburbana, urbana o rural) los cuales deben codificarse en forma numérica. [14]

En la tabla 3.1 se muestra un ejemplo con los tipos de datos que se pueden presentar.

Sexo	Ingresos	Comunidad	Edad	Preferencia política
Masculino	$60,000.00	suburbana	54 años	republicano
Femenino	$24,000.00	urbana	28 años	demócrata
Masculino	$30,000.00	rural	31 años	libertario
Femenino	$30,000.00	suburbana	48 años	republicano
Femenino	$18,000.00	urbana	22 años	demócrata
Masculino	$56.000.00	rural	39 años	otros

Tabla 3.1. Ejemplo de un conjunto de datos de entrenamiento.

En este conjunto de datos de entrenamiento se muestran varios ejemplos, que en este caso corresponden a los datos de diferentes personas en donde el punto a alcanzar (target) es la postura política de cada una.

Para los datos cuantitativos (en este caso ingresos y edad), es necesario normalizarlos con el propósito de representar los datos en una misma escala entre valores definidos. A continuación se muestra la formula para normalizar los datos:

$$x' = \frac{x - \bar{x}}{\sigma}$$

En donde:

x' = valor normalizado

x = valor original

$\bar{x}$ = promedio

σ = desviación estándar

Por lo tanto, la codificación quedaría como se muestra en la tabla 3.2.

	Ingresos	Normalizados	Edad	Normalizados
	$60,000.00	1,359464928	54 años	1.3825251033
	$24,000.00	-0,708453554	28 años	-0.731925055
	$30,000.00	-0,363800474	31 años	-0.487950036
	$30,000.00	-0,363800474	48 años	0.8945750669
	$18,000.00	-1,053106634	22 años	-1.219875091
	$56.000.00	1,129696207	39 años	0.1626500122
Promedio	36333.333333		37	
Desviación Estándar	17408.81003		12.296340919	

Tabla 3.2. Normalización de datos cuantitativos.

En el caso de los datos binarios (sexo), es decir datos en los que solo existen dos posibles opciones, se codifican asignando un valor numérico a cada una, preferentemente -1.00 y +1.00.

hombre = -1.00

mujer = +1.00

Por ultimo, los datos categóricos (comunidad y postura política), que son datos que pueden tener mas de dos opciones, se manejan de forma similar a los binarios, con la diferencia de que en este caso habrá más opciones que deberán representarse. Una de las formas más comunes de codificación es la llamada codificación 1 de C. En donde se configura una matriz de

valores numéricos. El tamaño de la matriz es el número de valores posibles, C. El primer valor categórico tiene valores de 0.0 en todas las posiciones, excepto un solo valor de 1.0 en la última posición. El segundo valor categórico tiene el único 1.0 en la penúltima posición, y 0.0 en las demás posiciones, y así sucesivamente hasta cubrir todas las opciones posibles. En el caso de comunidad, la codificación sería:

suburbana = [0.0, 0.0, 1.0]

rural = [0.0, 1.0, 0.0]

urbana = [1.0, 0.0, 0.0]

Y para postura política:

republicano = [0.0, 0.0, 0.0, 1.0]

demócrata = [0.0, 0.0, 1.0, 0.0]

libertario = [0.0, 1.0, 0.0, 0.0]

otro = [1.0, 0.0, 0.0, 0.0]

Por lo tanto, los datos de entrenamiento una vez han pasado por la etapa de pre-procesamiento están listos para ingresar a la red neuronal para su entrenamiento y quedarían como se muestra en la tabla 3.3:

Sexo	Ingresos	Comunidad	Edad	Preferencia política
-1.0	1,359464928	[0.0, 0.0 ,1.0]	1,382525103	[0.0, 0.0, 0.0, 1.0]
+1.0	-0,708453554	[1.0, 0.0 ,0.0]	-0,731925055	[0.0, 0.0, 1.0, 0.0]
-1.0	-0,363800474	[0.0, 1.0 ,0.0]	-0,487950036	[0.0, 1.0, 0.0, 0.0]
+1.0	-0,363800474	[0.0, 0.0 ,1.0]	0,894575067	[0.0, 0.0, 0.0, 1.0]
+1.0	-1,053106634	[1.0, 0.0 ,0.0]	-1,219875091	[0.0, 0.0, 1.0, 0.0]
-1.0	1,129696207	[0.0, 1.0 ,0.0]	0,162650012	[1.0, 0.0, 0.0, 0.0]

Tabla 3.3. Conjunto de datos después del pre-procesamiento.

También es importante aclarar que esta etapa de pre-procesamiento no solo es necesaria para los datos de entrenamiento, sino también en los nuevos datos de entrada para probar la red, en general para todos los datos que entren a la red neuronal. De igual forma, los datos que tenga la red en su salida estarán bajo estas condiciones, por lo que será necesario convertirlo s o interpretarlos para poder observar valores reales.

El sistema implementado en este proyecto no realiza esta etapa para los datos de entrenamiento, por lo que el conjunto de datos que se cargue, deberá haber sido previamente pre-procesado.

3.3 Sistema de proceso de la red neuronal

En el diagrama de flujo que se muestra en la figura 3.6 se describe como es que está diseñada la parte PS del sistema. Consta de 5 etapas: leer registro de control "ctrl", reset, modo entrenamiento de la red neuronal, enviar los pesos [W] a PL y modo prueba.

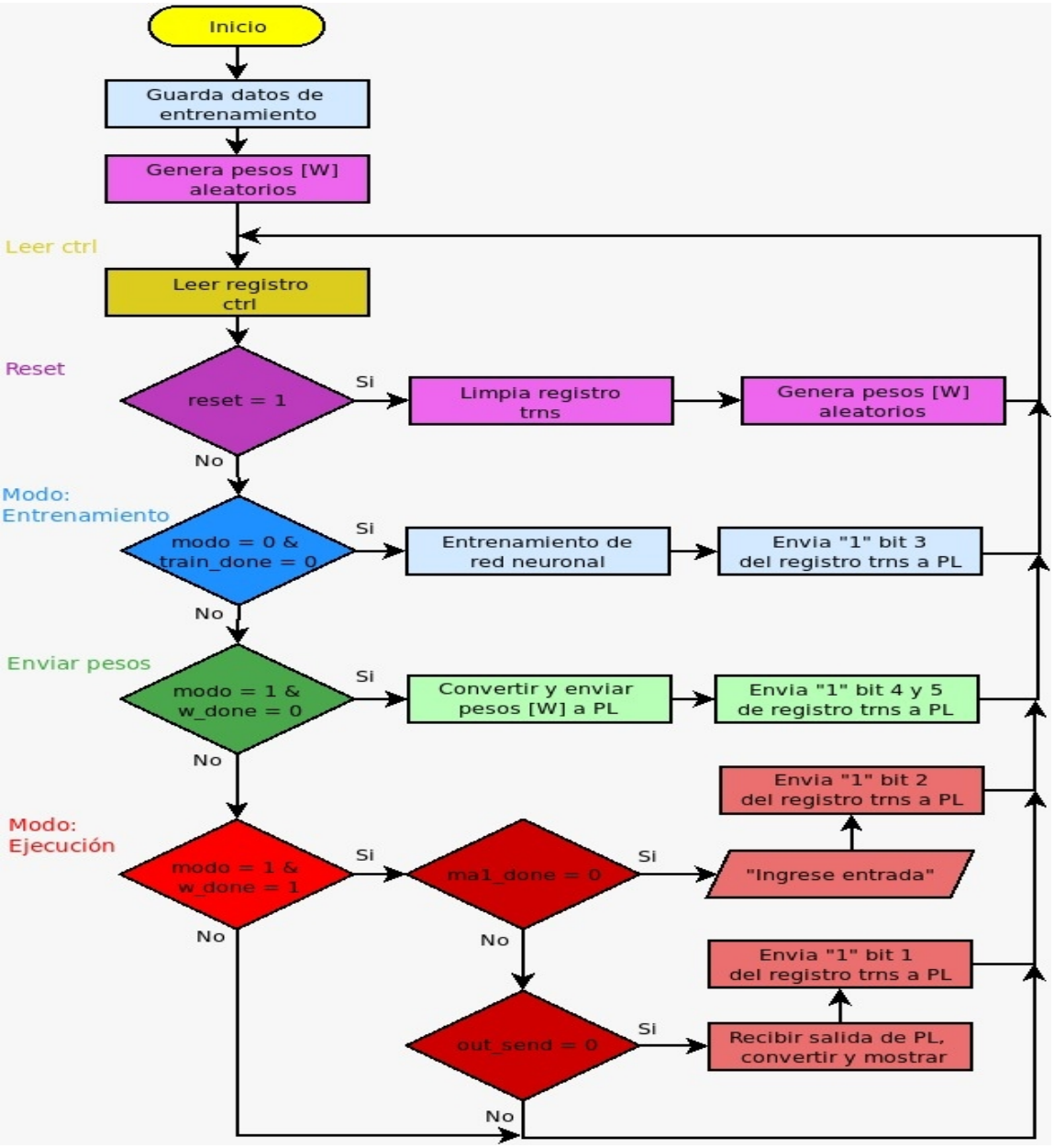


Figura 3.6. Diagrama de flujo PS.

El software presenta algunos parámetros para la configuración de la red neuronal a los cuales se les deben asignar valores antes de cargarlos en la tarjeta y también deben de coincidir con los parámetros que se describirán en PL, estos parámetros son los siguientes:

- inputCount: neuronas de entrada equivalente al número de datos que existen a la entrada de la red.
- hiddenCount1: neuronas en la capa oculta de la red. En caso de agregar una capa adicional habra que agregar una nueva variable (por ejemplo hiddenCount2).
- outputCount: neuronas en la capa de salida, equivalente a los datos de salida que tendrá la red.
- exampleCount: número de ejemplos con los que se entrenará a la red.
- learningRate: factor de aprendizaje.
- loss: porcentaje de error mínimo que se desee obtener para detener el entrenamiento de la red. Mientras este valor sea menor, el tiempo de entrenamiento será mucho mayor.

En la figura 3.7 se muestran dichos parámetros.

```
// Parameters
volatile int inputCount = 4;          // Input neurons
volatile int hiddenCount1 = 4;        // 1st hidden layer neurons
volatile int outputCount = 3;         // Output neurons
volatile int exampleCount = 120;      // Example combinations
volatile float learningRate = 0.3;    // Learning rate
volatile float lossPorcent = 10;      // Loss
```

Figura 3.7. Parámetros de la red neuronal.

- **Leer registro de control “ctrl”**

Función que se ejecuta en un bucle para leer el registro de 32 bits de control **“ctrl”** en donde PL escribe cual de las etapas debe ejecutar PS y se detallará mas adelante.

- **Reset**

Esta parte del programa es la responsable de reiniciar los datos de las variables de la red neuronal en cuanto a PS refiere, las cuales son: valores de los arreglos de pesos [W] y el registro de comunicación **"trns"** en el cuál se envían bits de control para PL, dicho registro se detallará mas adelante. Una vez que se han reiniciado dichos elementos, se carga nuevos valores aleatorios en los arreglos que contienen los pesos [W].

- **Modo de entrenamiento**

Contiene funciones encargadas del entrenamiento de la red neuronal teniendo como parámetros iniciales los arreglos de pesos [W] aleatorios y devolviendo en su salida los nuevos valores que éstos toman una vez que la red neuronal ha sido entrenada. El pseudocódigo siguiente representa el modo entrenamiento.

```
entrenamiento () {
         if (La red aun no ha sido entrenada) {
                  trns (bit 0) = 1; //Indica a PL que realice un reset
                  entrena_red(w1[N][N], w2[N][N]); //Entrena red, modifica valores W1 y W2
                  trns (bit 0) = 0; //Termina reset de PL
                  trns (bit 3) = 1; //Indica a PL que se ha entrenado la red
         }
{
```

- **Enviar pesos [w]**

Esta sección se encarga de tomar los arreglos de tipo flotante con los pesos [W], convertirlos en un formato IEEE 754[22] el cual es necesario para operar con datos en PL. El siguiente pseudocódigo muestra de forma mas detallada como se lleva a cabo este proceso.

```
enviar_pesos_PL () {
         if (PL no ha recibido matriz w1)
                  conv_IEEE754(w1); //Convierte matriz w1 en formato IEEE754
                  enviar_PL(w1); //Se envía matriz w1 a PL
                  trns (bit 4) = 1; //Indica a PL que se ha enviado la matriz w1
         }          else if (PL no ha recibido matriz w2)
```

22 Formato para representar números flotantes

```
                    conv_IEEE754(w2); //Convierte matriz w2 en formato IEEE754
                    enviar_PL(w2); //Se envía matriz w2 a PL
                    trns (bit 5) = 1; //Indica a PL que se ha enviado la matriz w2
          }
}
```

- **Modo ejecución**

Pide al usuario que introduzca, mediante consola, nuevos datos para realizar una prueba en la red neuronal. Realiza el pre-procesamiento de estos datos a partir de los datos obtenidos con el conjunto de entrenamiento de la red. Convierte los datos en formato IEEE 754 y los envía a PL para, finalmente, tomar los valores que PL devuelve en su salida, convertirlos del formato IEEE 754 a tipo flotante y mostrarlos al usuario. El siguiente pseudocódigo representa la forma en que funciona esta etapa.

```
prueba () {
          if (Se han enviado datos de entrada a PL) {
                    scan_mat(vector_entrada); //Pide al usuario introducir datos de entrada
                    normalizar(vector_entrada); //Normaliza los datos
                    conv_IEEE754(vector_entrada); //Convierte vector en formato IEEE754
                    enviar_PL(vector_entrada); //Se envia el vector normalizado a LP
                    trns (bit 2) = 1 //Indica a PL que se han enviado datos de entrada
          }
          else if (PL ha calculado el resultado de la propagación hacia adelante) {
                    recibir_LP(vector_res); //Recibe de PL vector de resultado
                    conv_float(vector_res); //Convierte el vector resultado a tipo flotante
                    mostrar_matriz(vector_res); //Muestra el resultado
          }
}
```

3.4 Comunicación ARM-FPGA

La comunicación entre los dos sistemas (PS y PL) es posible gracias a la interfaz AXI, la cual sera implementada mediante la creación de un propio IP core[23], el cual contendrá las señales y

23 En diseño elctrónico, un módulo IP (intellectual property core), es un bloque lógico que puede ser reutilizado y que tiene propiedad intelectual de un tercero.

los módulos que se han diseñado para el control y propagación de la RNA, a este IP se le agregará el bloque de diseño correspondiente al proceso de sistema ZYNQ, para "empacar" todo y conectar con el PS mediante el software SDK.

3.4.1 IP core

Un IP core es un bloque de lógica diseñado para un fin en especifico, estos ayudan a un diseño mas rápido. Dicho de otra forma un IP core es un modulo de lógica ya diseñado y listo para usar, existen diferentes tipos de IP desde unidades lógicas aritméticas, hasta módulos que permitan habilitar los periféricos de la tarjeta como puede ser el convertidor analógico digital (ADC).

A continuación se describe el proceso, en el ambiente de desarrollo, para crear y editar un módulo IP.

1 Como se puede ver en la figura 3.8, en el software VIVADO crear un nuevo proyecto (***file > Project > new***), nombrar el proyecto y guardarlo en una ubicación deseada.

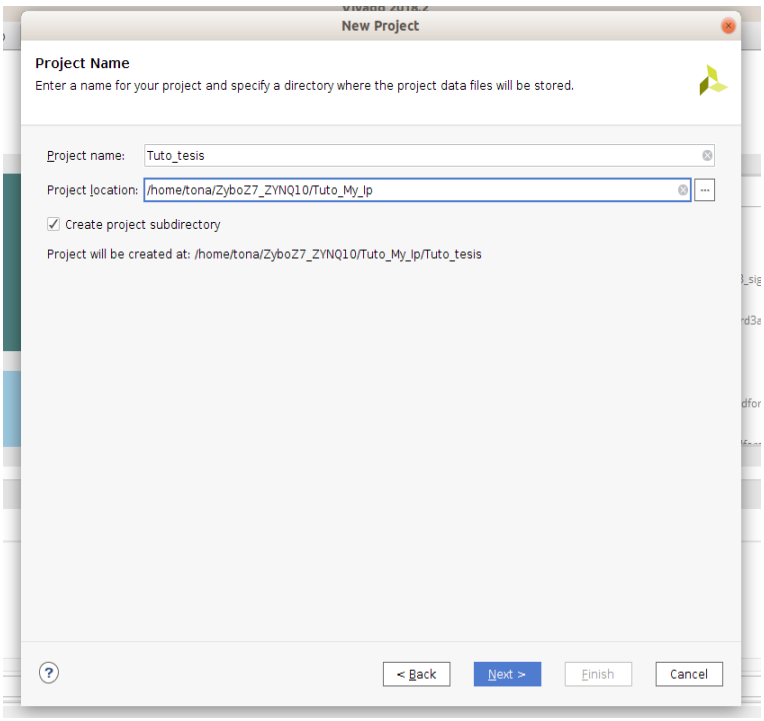


Figura 3.8. Creación de nuevo proyecto.

1.1 Seleccionar RTL Project, asegurándose de no habilitar la opción "***Do not specify source at this time***." Figura 3.9.

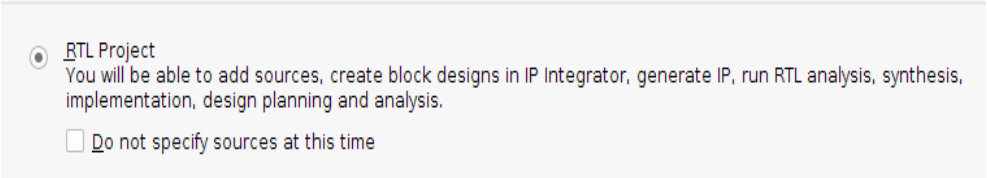


Figura 3.9. Proyecto RTL.

Nota: VHDL como lenguaje, es posible agregar algunos módulos desde este momento y el archivo que conecta con los puertos de la tarjeta (constructor), aunque también se puede hacer después.

1.2 Seleccionar la tarjeta de desarrollo a usar, para nuestro caso Zybo Z-10 y finalizar.

Hasta este momento solo se ha creado un proyecto, a continuación se editará el módulo IP, como se deja ver en la figura 3.10.

2 ***Tools > Create and package new IP*** y clic en ***next***.

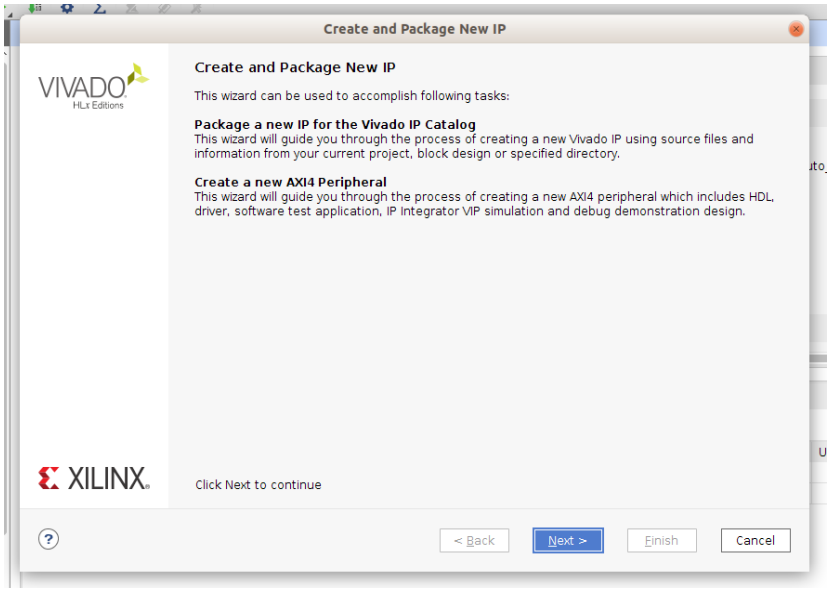


Figura 3.10. Nuevo IP.

2.1 Habilitar la interfaz AXI como se puede observa en la figura 3.11 "***Create AXI4 peripheral***" y click ***next***.

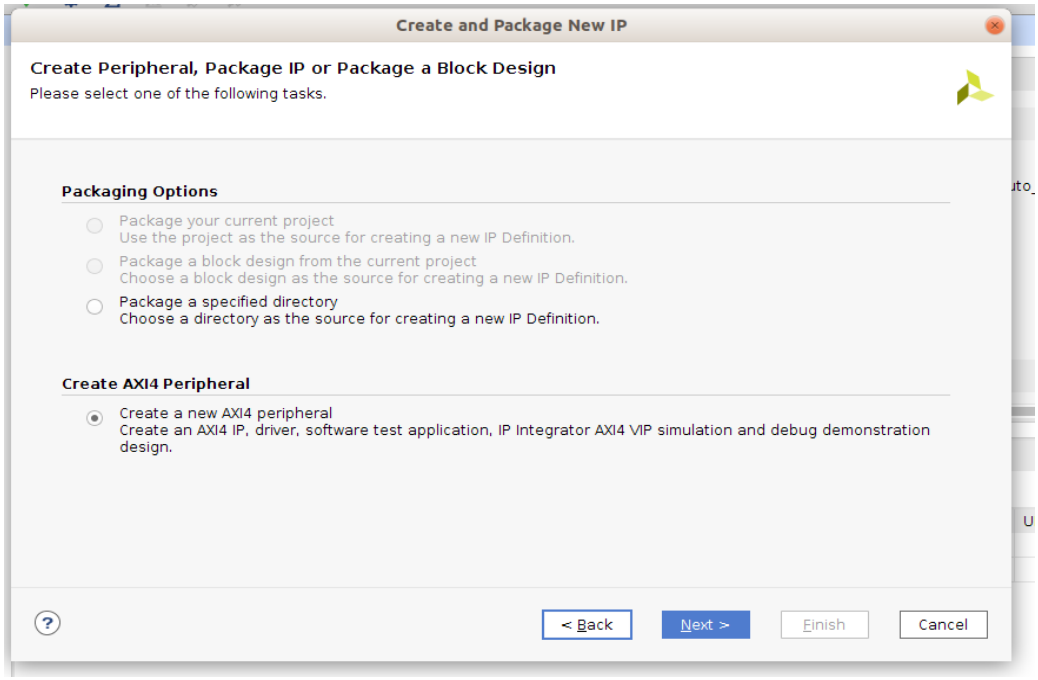


Figura 3.11. Interfaz AXI.

2.2 Nombrar al IP que se esta implementado y clic en next.

2.3 Como se puede ver en la figura 3.12 tenemos los datos de la interfaz AXI se pueden ver: nombre, tipo, modo, el ancho de datos de sus buses, y el número de registros (el cual se puede definir por el usuario).

2.4 Seleccionar "***Edit IP***" y finalizar. Figura 3.13.

Es importante señalar que se modificará el número de registros que se utilizará, dependiendo de la cantidad de pesos a transferir, el largo de la palabra de datos de los registros permanecerá en 32 bits. Y en la interfaz seguirá siendo esclavo.

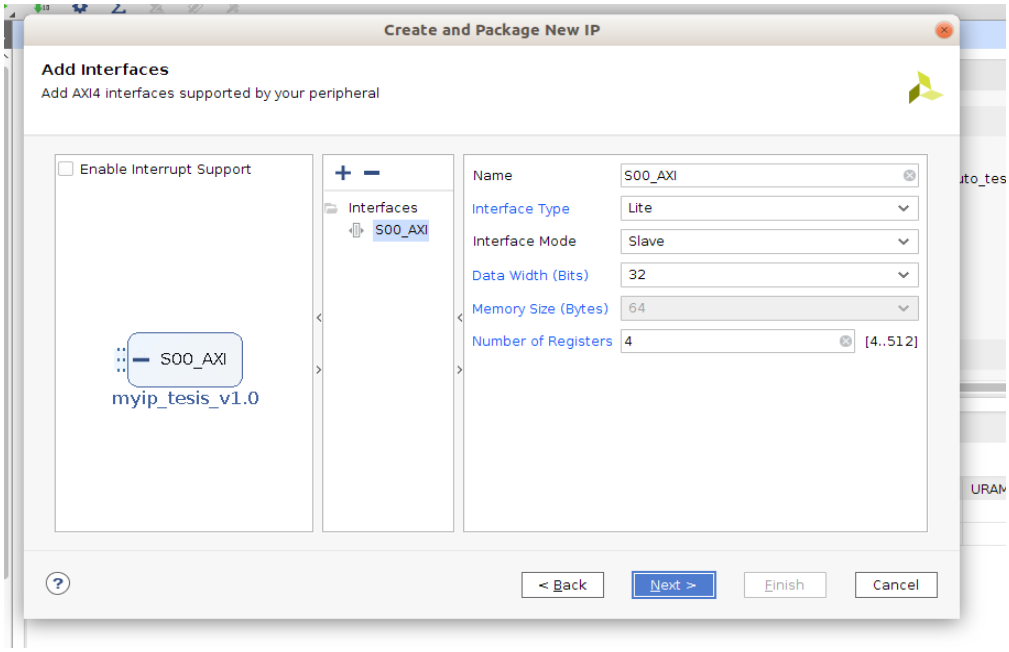


Figura 3.12. Datos de la interfaz AXI.

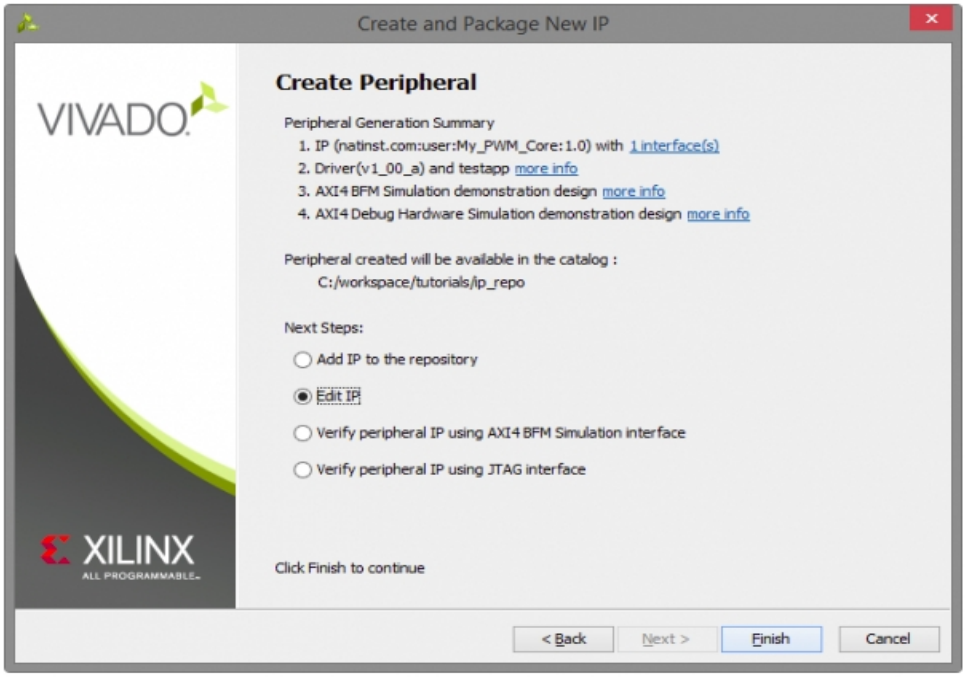


Figura 3.13. Editar IP.

3 Se abrirá una nueva ventana en ***VIVADO,*** la cual permitirá editar el IP. Figura 3.14.

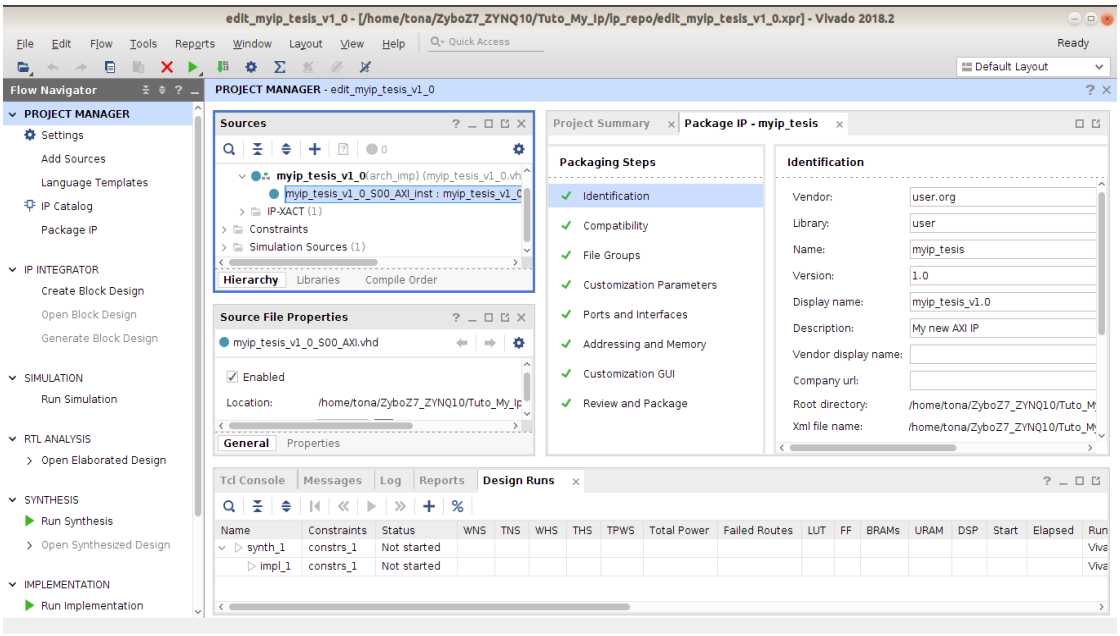


Figura 3.14. Nueva instancia para editar el IP.

3.1 Al expandir el nivel mas alto del IP, se encontrará una instancia, como se puede ver en la figura 3.15. Se entra a la instancia para implementar los parámetros, señales de control y la lógica del diseño, de esta manera se esta creando el IP.

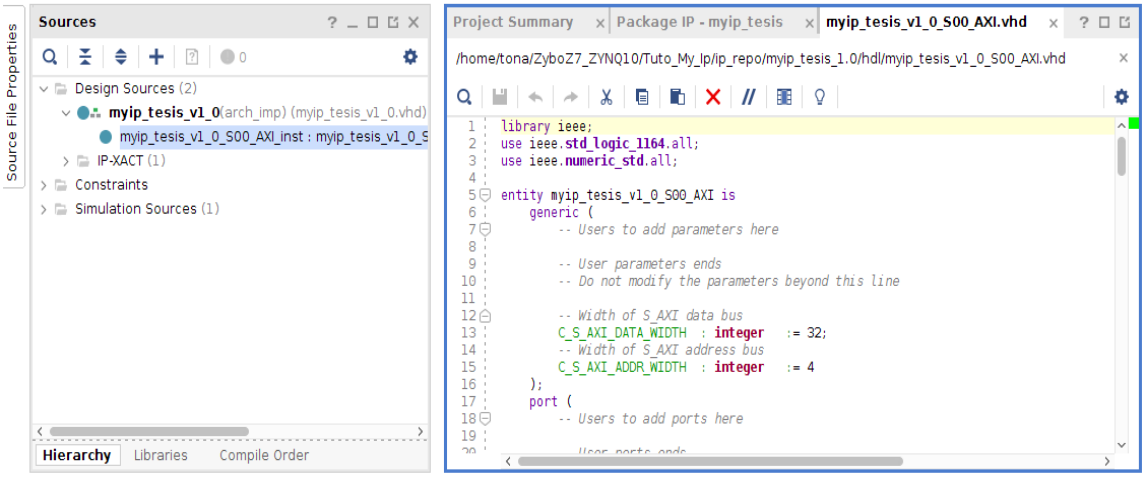


Figura 3.15. Edición de un IP.

3.2 Al terminar de editar el IP, se regresa al nivel superior para instanciar los parámetros y señales. Observe la figura 3.16.

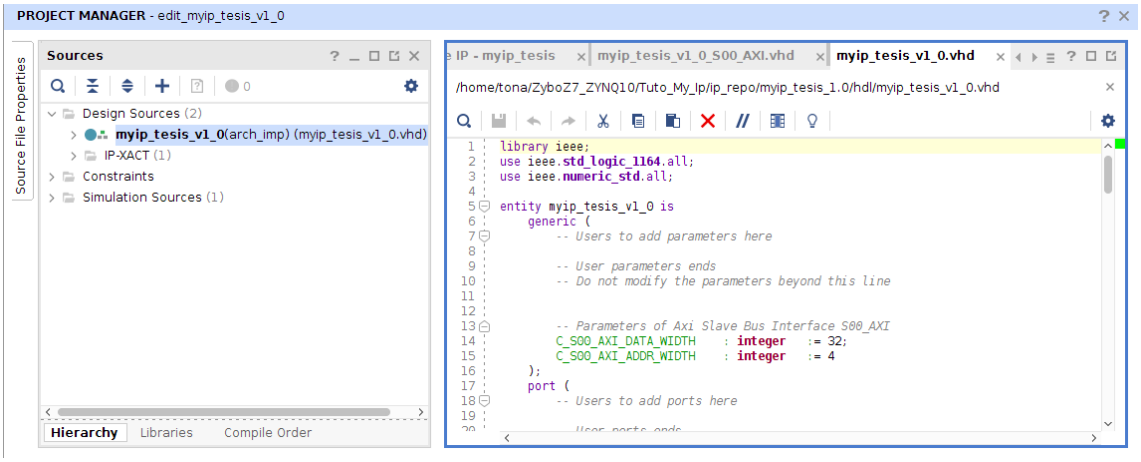


Figura 3.16. Nivel superior de un propio IP.

4 Una vez que se ha escrito en el IP core es momento de unirlo todo para crear un nuevo IP.

4.1 Ir a ***Package IP > Compatibility***. Hay que asegurar que Artix 7 y Zynq estén presentes, de no encontrase se agregan dando clic en el la opción "+". Observar figura 3.17.

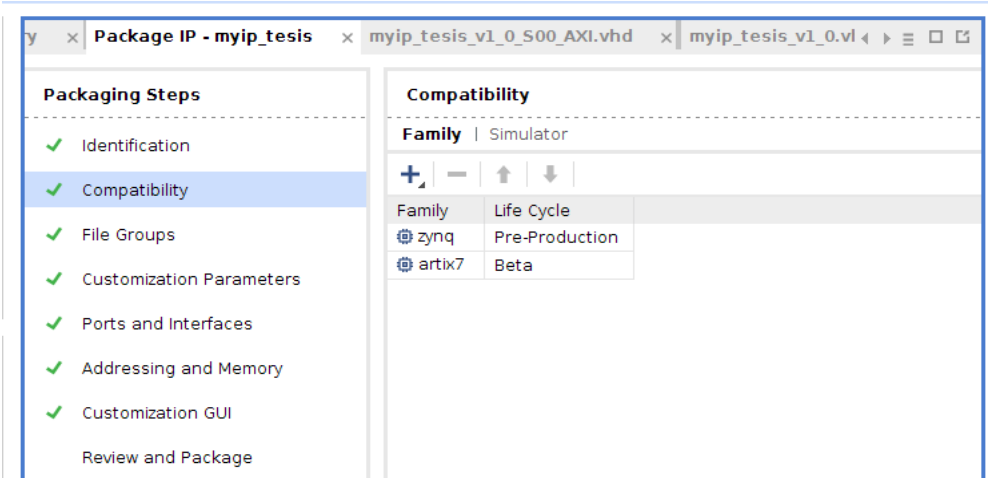


Figura 3.17. Package IP.

4.2 Con lo anterior el IP core está completo, ahora se va a ***Review and Package*** y clic en ***Re-package IP,*** como se puede ver en la figura 3.18.

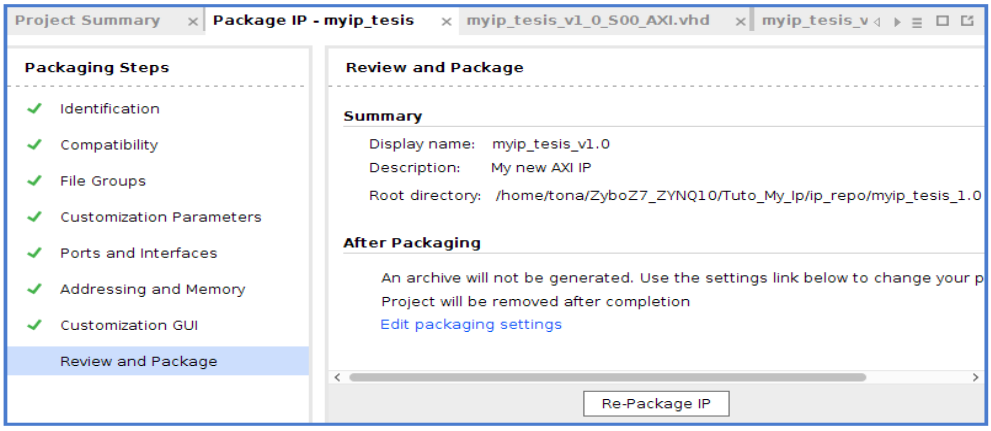


Figura 3.18. Empacar el IP.

Aparecerá una ventana preguntando si se desea cerrar el proyecto, clic en ***yes***, con esto se ha creado un IP nuevo.

3.4.2 Zynq System

Una vez que se creó un IP core, es momento de añadir el bloque del sistema Zynq, este bloque es el que genera las conexiones para hacer el "puente" entre el PL y PS.

1 En el ***Project Manager*** del diseño original seleccionar ***Create Block Design***, aparecerá una ventana donde se podrá nombrar el diseño, y se da clic en ***ok***.

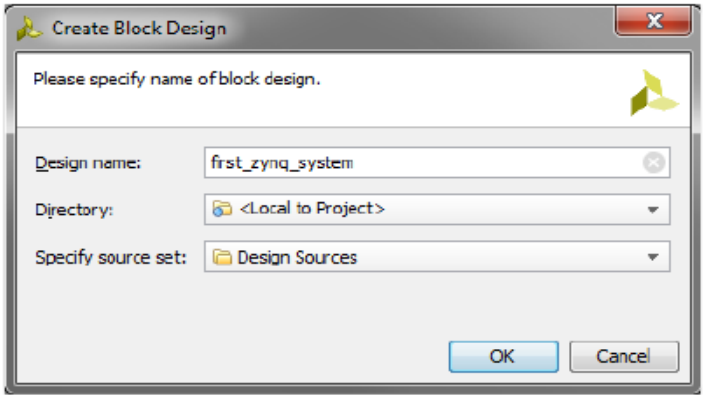


Figura 3.19. Nombrar al bloque del sistema Zynq.

1.1 Se abrirá una nueva ventana en el proyecto la cual permitirá agregar el bloque del sistema Zynq, mediante la opción ***Add IP***. Observar figura 3.20.

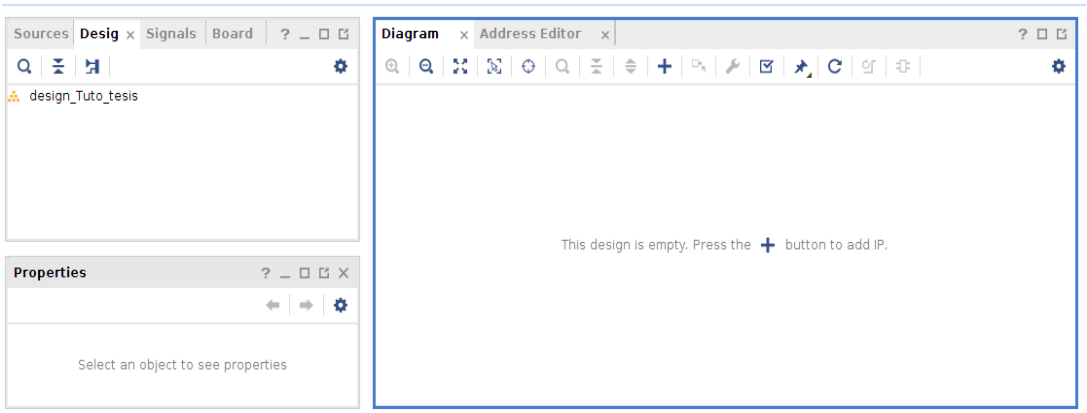


Figura 3.20. Agregar bloque del sistema Zynq.

1.2 Como se puede ver en la figura 3.21 se selecciona ***Add IP***, aparecerá una barra donde se puede teclear el nombre del bloque a agregar, en este caso: ***Zynq Processing System***.

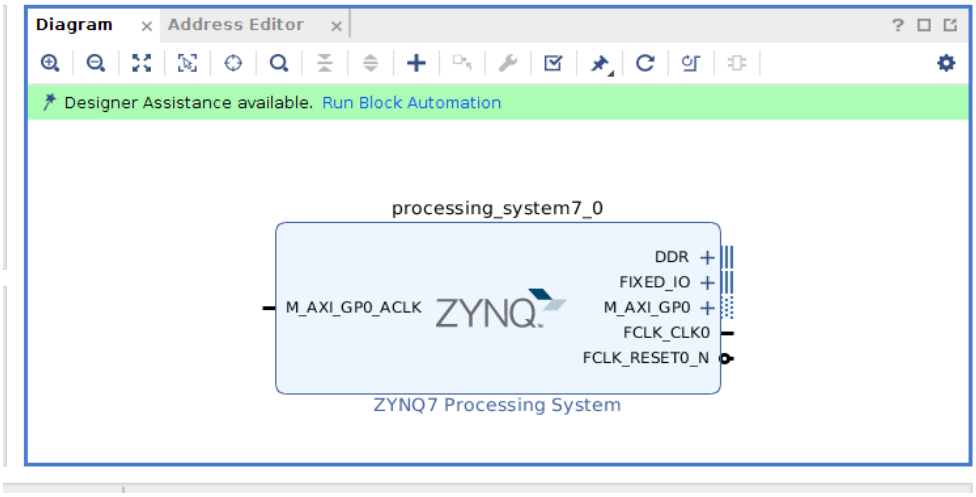


Figura 3.21. Bloque Proceso del sistema Zynq.

1.3 De la misma manera que se hizo con el bloque de diseño, se agregará el IP core que se creo. Figura 3.22.

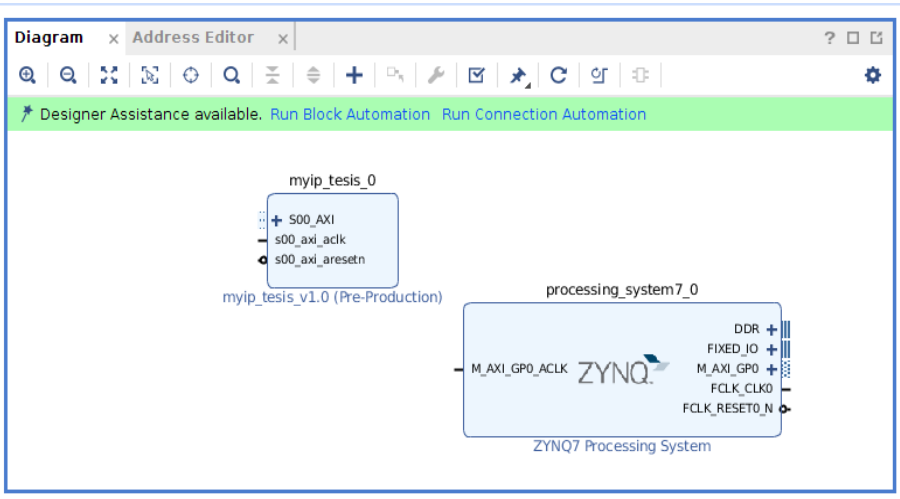


Figura 3.22. IP core y proceso del sistema Zynq.

1.4 Se da clic en ***Run Block Automation*** y posteriormente ***ok***. Esto habilitará los puertos de entrada y salida de la tarjeta.

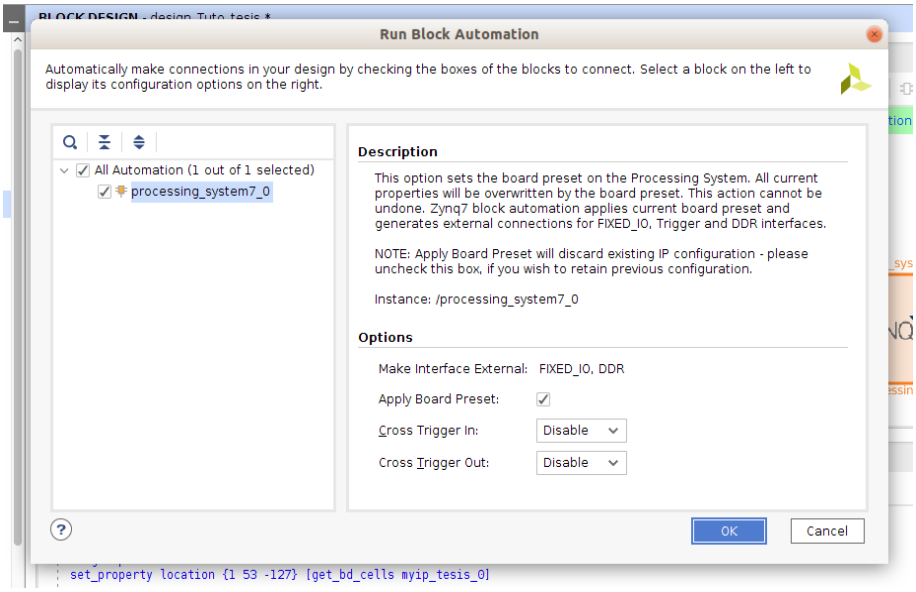


Figura 3.23. Run Block Automation.

1.5 Clic en ***Run Connection Automation***, de esta manera se generan las conexiones del IP con el proceso de sistema Zynq. El diseño se debe de ver como en la figura 3.24.

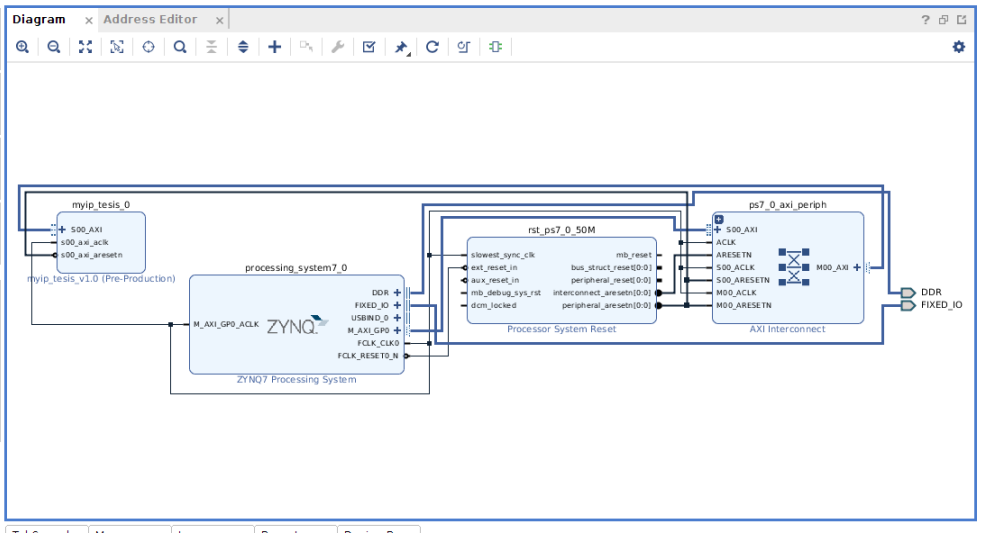


Figura 3.24. Run Connection Automation.

1.6 Es necesario habilitar los puertos de entrada y salida del IP, esto se hará dando clic derecho en cada señal de dicho módulo y se seleciona ***Create Port***.

1.7 Una vez creados los puertos se valida el diseño, ***tools > Validate Design*** desde la barra de menú. Aparecerá un diálogo de que dicha validación fue exitosa, se presiona ***ok*** y listo.

Con la validación del diseño se pueden generar los archivos para nuestro sistema.

Ir a la tabla donde están los archivo fuente seleccionando ***window > sources***.

1.8 En la ventana de “Source” clic derecho en el nivel más alto del sistema y seleccionar ***Create HDL Wrapper***.

1.9 Después de realizar la síntesis se pasa a generar el archivo de bits dando clic en "Generate Bitstream". Se abrirá un diálogo solicitando la implementación y síntesis antes de generar el bitstream, clic en "yes" para aceptar.

Por último se exporta el diseño al SDK donde se crea la aplicación de software.

1.10 ***File > Export > Export Hardware***. Se abrirá una ventana, como se deja ver en la figura 3.25 asegúrese que esté habilitada la opción incluye bitstream y clic ok.

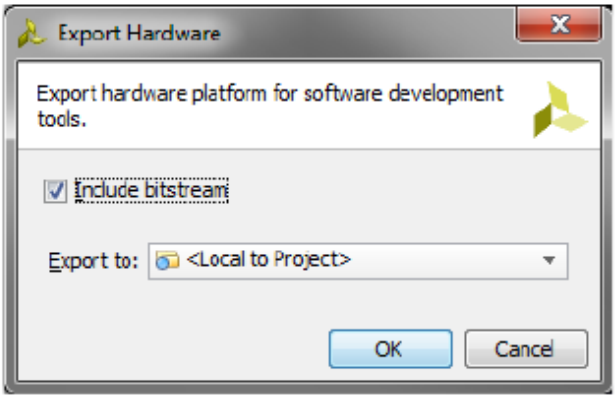


Figura 3.25. Exportar Hardware.

1.11 Cambiar hacia el software SDK desde VIVADO seleccionando:

File > Launch SDK y clic ***ok***.

Con esto se ha creado la conexión entre el PL y PS, añadiendo por medio de un IP core la lógica (propagación de la red neuronal) y señales de control. Todo el hardware ha sido generado y exportado al SDK donde se desarrollara la aplicación de software (entrenamiento de la red neuronal).

3.4.3 Aplicación de software en SDK

El kit de desarrollo de software (SDK) de Xilinx es el entorno donde se desarrollarán las aplicaciones de los procesadores de la serie Zynq.

En el apartado anterior ya se ha exportado el hardware, y al seleccionar ***Launch SDK*** el software se abrirá automáticamente.

Una vez que el software este listo es momento de crear ls aplicación de software.

1 Seleccionar ***File > New > Aplication Project.***

2 Se abrirá una nueva ventana como se puede ver en la figura 3.26 donde se podrá dar nombre al diseño, elegir el lenguaje de programación, entre otras cosas mas, configurar y clic en ***Finish.***

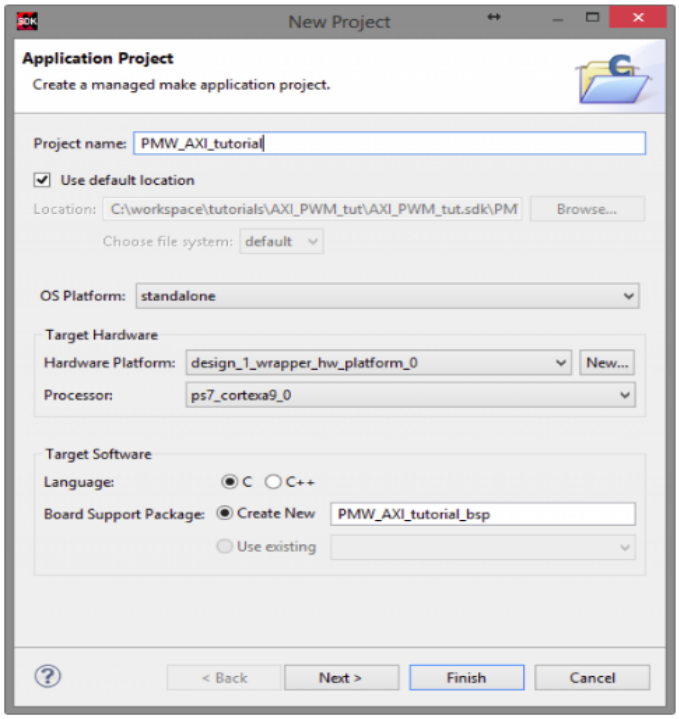


Figura 3.26. Aplicación de proyecto.

3 Se creará una carpeta con el nombre del diseño. Al expandir dicha carpeta dirigirse a s***rc > new > File.*** Como se ve en la figura 3.27 en dicha ubicación crear un archivo y nombrar ***main.c***

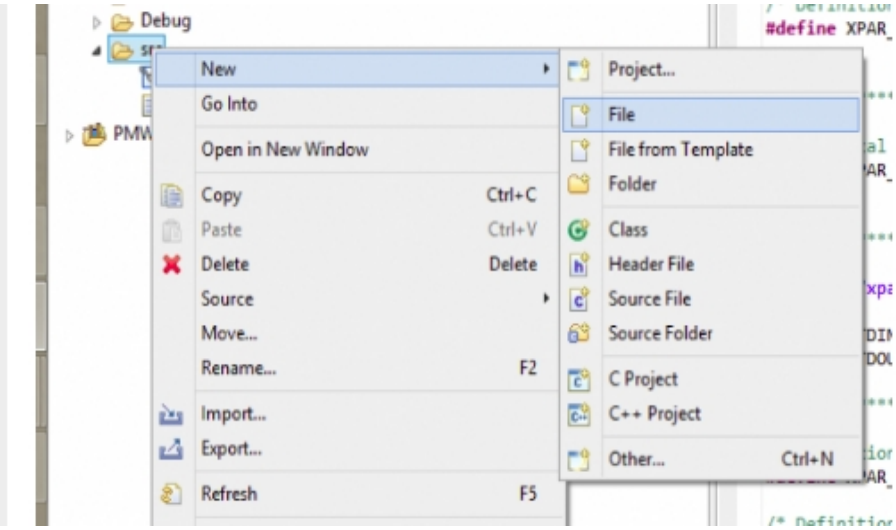


Figura 3.27. Creación de archivo main.c.

En este archivo se implementara el código. Como bien se sabe en el ***main*** se ejecuta el programa principal, si se necesitan otras librerías, se agregarán de la misma manera en la que se creo el ***main***.

De esta forma se crea toda la programación de la red neuronal en lenguaje C. Al finalizar la programación es momento de implementar todo el diseño en la tarjeta.

Asegúrese de que la Zybo Z-10 esté en el modo encendido y de programación adecuado, colocando los jumpers como se muestra en la figura 3.28.

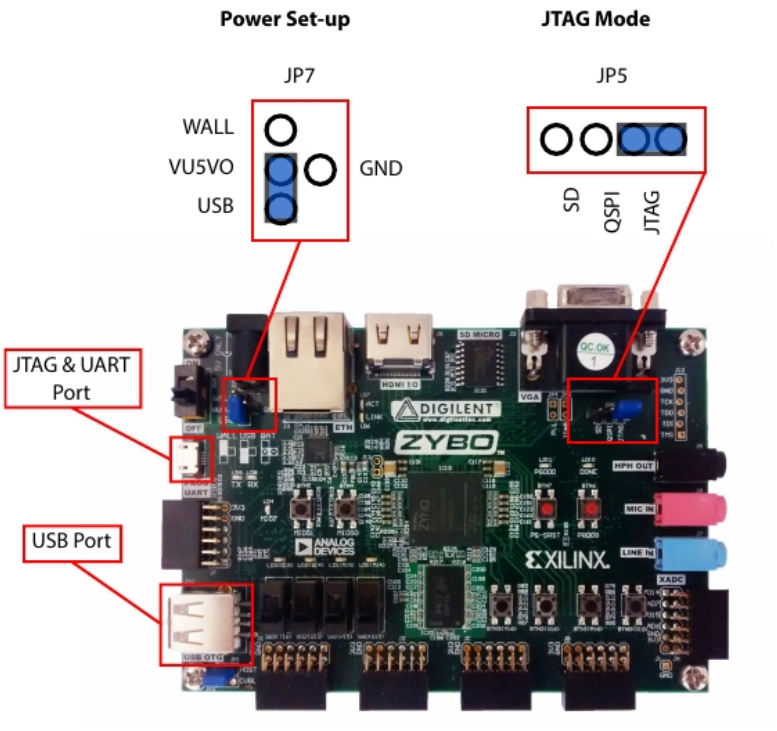


Figura 3.28. Configuración para programar desde SDK en la Zybo.[24]

24 L. H. Crockett, R. A. Elliot, y M. A. Enderwitz, The Zynq Book Tutorials for Zybo and ZedBoard. Glasgow: Strathclyde Academic Media, 2015.

4 Primero se carga el ***bitstream*** en la FPGA seleccionando ***Xilinx Tools > Program FPGA.*** Figura 3.29.

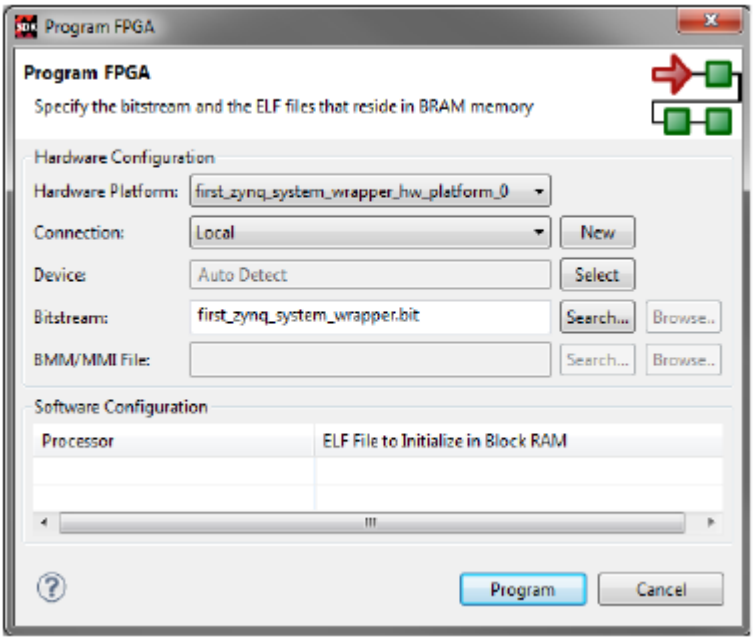


Figura 3.29. Programando el PL.

5 Una vez que el PL fue programado, se puede programar el PS. Seleccione el proyecto, clic derecho y seleccione ***Run AS > Launch on Hardware (GDB).***

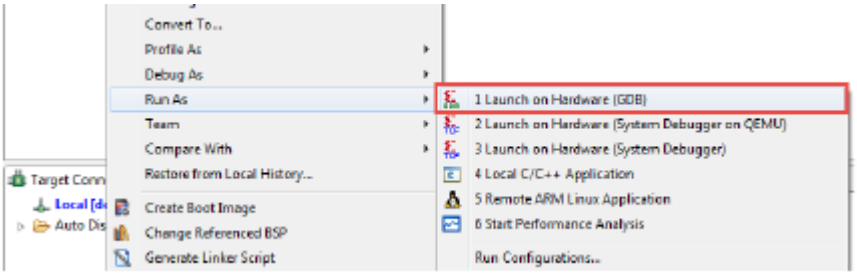


Figura 3.30. Programando el PS.

De esta manera se ha implementado el proyecto en la tarjeta programando tanto el FPGA (PS) como el procesador Cortex-A9 (PS) y manteniendo una comunicación y transferencia de datos constante. [15]

3.4.4 Uso de registros y señales de control

Dado que es de interés la transferencia de datos, tanto PS-PL como PL-PS es importante mantener una comunicación constante entre ambos sistemas. En esta sección se mostrará el uso de registros y como escribir en ellos para crear señales de control y así tener una transferencia de datos y una comunicación óptimas.

Hay que recordar que cuando se edita el IP core permite seleccionar la cantidad de registros y el ancho de datos, siendo éstos de 32 bits

Registro 0 trns.

Registro en el que el PS escribe bits de control para PL.

31	...	12	11	10	9	8	7	6	5	4	3	2	1	0

- w_clr (bit 0): bit que habilita un reset para los módulos de PL que cargan las matrices de pesos (w), se pone en "1" el bit del registro.
- in_clr (bit 1): reset para el módulo de PL que carga el vector de entrada y borra los valores previamente cargados, se pone en "1" el bit del registro.
- in_ready (bit 2): cuando el bit esta en nivel alto indica que PS ha enviado el vector con los datos de entrada y habilita el módulo en PL encargado de guardar dichas entradas.
- tran_done (bit 3): el bit tendrá un nivel alto cuando el PS haya terminado el entrenamiento de la red neuronal.
- w1_ready (bit 4): PS escribirá un nivel alto en este bit cuando haya terminado de escribir la matriz de pesos W_1 en los registros de entrada de PL. Habilita el módulo que recibe y guarda la matriz de pesos W_1 en el PL.
- w2_ready (bit 5): PS escribirá un nivel alto en este bit cuando haya terminado de escribir la matriz de pesos W_2 en los registros de entrada de PL. Habilita el módulo que recibe y guarda la matriz de pesos W_2 en el PL.

Nota: si se agregán mas capas a la red neuronal se agregaran mas pesos, con lo cual será necesario usar los otros bits para añadir las señales de control con el mismo funcionamiento de las señales w1_ready y w2_ready.

Registro 1 ctrl.

Registro en el que PL escribe bits de control para el PS.

31	...	12	11	10	9	8	7	6	5	4	3	2	1	0

- clr (bit 0): bit conectado directamente a un push button de la tarjeta, habilita un reset general para todos los módulos en PL y el programa en PS.
- mode (bit 1): bit conectado directamente al switch de la tarjeta. Selecciona la parte de programa que se ejecutará cuando esté en 0 entrenará y en 1 la ejecución.
- w_done (bit 2): el bit se pondrá en alto cuando todas las matrices de pesos han sido recibidas y guardadas en el PL.
- in_done (bit 3): el bit se pondrá en alto cuando el vector de entrada ha sido recibido y guardado en el PL. Cuando el bit se encuentre en 0 le indicará al programa que le solicite una entrada al usuario.
- ff_done (bit 4): indica con un nivel alto cuando el PL terminó de realizar la propagación hacia adelante.
- out_send (bit 5): indica con un nivel alto cuando PL ha enviado el vector de salida para que sea leído por el PS.
- w1_done (bit 6): el bit se pone en “1” cuando el PL ha recibido y guardado la matriz w_1.
- w2_done (bit 7): el bit se pone en “1” cuando el PL ha recibido y guardado la matriz w_2.

3.5 Red neuronal en VHDL

En esta sección se describe el desarrollo e implementación de la propagación hacia adelante de una red neuronal artificial en VHDL.

Este diseño se compone principalmente de 4 partes:

1) Multiplicación matricial: multiplica las entradas de cada neurona con sus pesos, independientemente de la capa en que se encuentre. Al procesar varias neuronas y por consecuencia varios pesos, se colocan en matrices para facilitar su procesamiento.

2) Adición de columna: una vez realizada la multiplicación se le agregará una columna, la cual se agrega para incluir el bias a cada neurona.

3) Función sigmoide: caracteriza las salidas de las neuronas, para irse propagando capa tras capa.

4) Capa de salida: realiza la multiplicación de los pesos con las entradas de las últimas neuronas que "ve", caracteriza la salida mediante la función sigmoide y arroja el resultado de la red, si dicha red ha sido entrenada, la salida será la deseada, de lo contrario arrojará un resultado erróneo.

Los primeros tres elementos estarán relacionados formando un diseño de alto nivel, este diseño puede ser parte de otro como una instancia, tendremos instancias cuantas capas ocultas se deseen. Por otra parte la capa de salida solo podrá ser una instancia por una vez.

El diseño de alto nivel se puede ver en la figura 3.4 y 3.5.

A continuación se describirán a detalle dichos módulos.

3.5.1 Números punto flotante en VHDL

En la etapa de entrenamiento de una RNA, se ajustan los pesos para que al propagarse, la salida sea correcta, estos pesos suelen ser de punto flotante pues son sometidos a muchas iteraciones haciendo casi imposible que sean números enteros.

Al implementar la propagación hacia adelante de la red neuronal en VHDL, se tendrá que implementar notación de punto flotante, pues es la manera de representar números binarios con punto decimal, esta notación se desarrolla bajo el estándar IEEE 754.

Estándar IEEE 754

La representación de punto flotante está basada en notación científica. El punto decimal no se halla en una posición fija dentro de la secuencia de bits, si no que su posición se indica como una potencia de la base.

En todo número en punto flotante se distinguen tres componentes:

1. Signo: indica el signo del número 0 para positivo, 1 para negativo.
2. Mantisa: contiene la magnitud del numero en binario puro.
3. Exponente: contiene el valor de la potencia de la base.

Existen cuatro formatos de dicho estándar, pero los dos mas usados son la de precisión simple y precisión doble.

- Precisión simple (32 bits): 1 bit de signo, 8 bits de exponente y 23 bits de mantisa.
- Precisión doble (64 bits): 1 bit de signo, 11 bits de exponente y 52 bits de mantisa.

Dicho estándar cuenta con ciertas particularidades:

- La secuencia de bits es: primero el bit de signo, seguido del exponente y finalmente la mantisa.
- El exponente no tiene signo, en su lugar se realiza un desplazamiento (127 para sencilla y 1023 para doble).

- Se asume que el bit más significativo de la mantisa es 1 y se omite, excepto para casos especiales. [16]

Formato para punto flotante de precisión simple (32 bits) en IEEE 754:

1 bit de signo	8 bits de exponente + 127	23 bits de mantisa

Para este trabajo se utilizará la precisión simple. Es importante señalar que todas las operaciones que involucra la propagación hacia adelante tendrán que desarrollarse bajo este formato.

Para ejemplificar se realizara la conversión del número decimal 45.25 a número de punto flotante en formato IEEE 754:

- Conversión a binario: Pasar parte entera y parte decimal a binario.

  $Parte\,entera = 45_{10} = 101101_2$

  $Parte\,decimal = 0.25_{10} = 0.01_2$

- Notación científica en binario: Recorrer el punto hasta que el primer número sea un 1 y multiplicarlo por 2 elevado al número de posiciones que se recorrió el punto.

  $101101.01_2 = 1.0110101_2 * 2^5$

- Mantisa: Es el número binario que queda a la derecha del punto, completando los bits que demanda el formato con ceros si es necesario.

  $Mantisa = 01101010000000000000000_2$

- Exponente: Es el número binario que se obtiene sumando 127 al exponente de la notación científica.

  $Exponente = 5_{10} + 127_{10} = 132_{10} = 10000100_2$

- Signo: 0 para número positivo, 1 para negativo.

  $Signo = 0$

- Ordenar: Signo-Exponente-Mantisa.

$$45.25_{10} = 01000010001101010000000000000000_{2}$$

El desarrollo en VHDL de dicho formato fue implementado de dos formas. En la primera se tomó como base un paquete desarrollado en la universidad de Johns Hopkins, el paquete desarrolla aritmética en dicho estándar, esto lo logra mediante una máquina de estados la cual primero convierte el número a dicho formato para después operarlo, se tomó esta librería y se adaptó al diseño. [17] La segunda fue mediante un IP core que el software VIVADO ofrece.

La utilización de una u otra será especificada en los siguientes capítulos.

3.5.2 Multiplicación matricial con números punto flotante en VHDL

Ya que la todos los datos que entran a la red neuronal en PL son de punto flotante y están representados con el formato IEEE 754, todas las operaciones que se hagan con estos datos deben de respetar este formato.

Una de las operaciones mas importantes que fue necesaria implementar para que PL realice la propagación hacia adelante de la red neuronal fue la multiplicación de matrices. La cual tiene como condición que el número de columnas de la primera matriz debe de ser igual al número de filas de la segunda matriz.

La figura 3.31 muestra la forma en que se desarrolla.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} =$$

$$\begin{bmatrix} a_{11} \cdot b_{11} + a_{12} \cdot b_{21} + a_{13} \cdot b_{31} & a_{11} \cdot b_{12} + a_{12} \cdot b_{22} + a_{13} \cdot b_{32} \\ a_{21} \cdot b_{11} + a_{22} \cdot b_{21} + a_{23} \cdot b_{31} & a_{21} \cdot b_{12} + a_{22} \cdot b_{22} + a_{23} \cdot b_{32} \end{bmatrix}$$

Figura 3.31. Multiplicación de matrices.

Como puede observarse, esta operación se compone de varias sumas de productos, por lo tanto a partir de la suma y multiplicación en formato IEEE 754 se inicia la implementación de la multiplicación de matrices.

El siguiente ejemplo muestra el algoritmo de la suma y resta de números flotantes en representación IEEE 754:

$A = 35.75_{10} = 0100\,0010\,0000\,1111\,0000\,0000\,0000\,0000_2$

$B = 20.5_{10} = 0100\,0001\,1010\,0100\,0000\,0000\,0000\,0000_2$

Calcular $R = A + B$

- Separar los número en formato IEEE 754 en signo (S), exponente (E) y mantisa (M).

  $SA = 0_2$

  $EA = 10000100_2 = 132_{10} - 127_{10} = 5_{10}$

  $MA = 1.0001111_2 * 2^5$

  $SB = 0_2$

  $EB = 1000\,0011_2 = 131_{10} - 127_{10} = 4_{10}$

  $MB = 1.01001_2 * 2^4$

- Diferenciar los exponente de ambos números y tomar el exponente más grande como el exponente tentativo del resultado.

  $ER = EA = 5_{10}$

- Desplazar la mantisa del número con el menor exponente.

  $MB = 0.101001_2 * 2^5$

- Sumar o restar ambas mantisas según indique el bit de signo o las instrucciones que se tengan y tomar el resultado como mantisa provisional.

$$MR = MA + MB = 1.0001111_2 * 2^5 + 0.101001_2 * 2^5 = 1.1100001_2 * 2^5$$

- Normalización de exponente y mantisa.

$$R = 0100\,0010\,0110\,0001\,0000\,0000\,0000\,0000_2 = 56.25_{10}$$

En la figura 3.32 se muestra el diagrama de flujo para la implementación de esta operación.

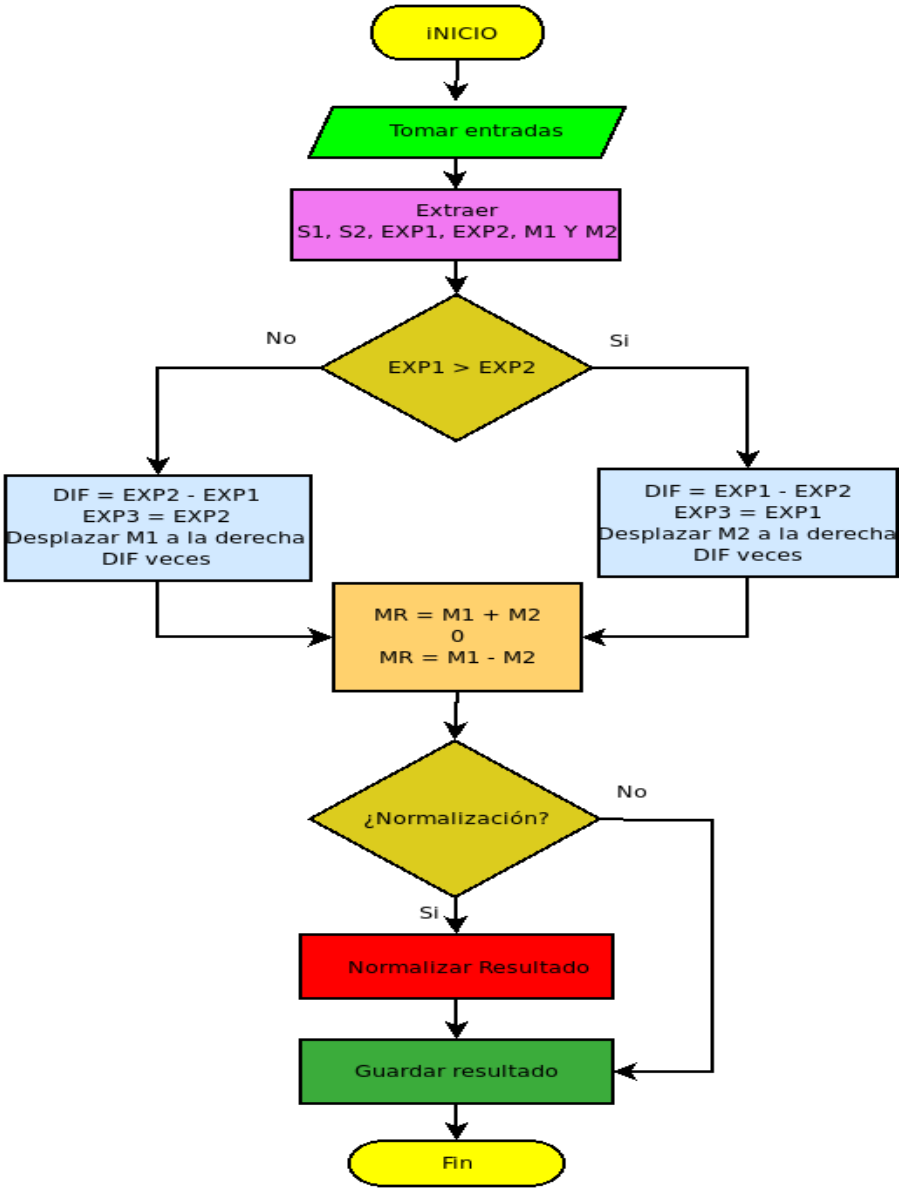


Figura 3.32 Diagrama de flujo de la suma en formato IEEE 754.

Para la multiplicación en formato IEEE 754 el algoritmo se describe con el siguiente ejemplo:

$A = -0.3_{10} = 1011\,1110\,1001\,1001\,1001\,1001\,1001\,1010_2$

$B = 500.25_{10} = 0100\,0011\,1111\,1010\,0010\,0000\,0000\,0000_2$

Calcular $R = A * B$

- Separar los número en formato IEEE 754 en signo (S), exponente (E) y mantisa (M).

  $SA = 1_2$

  $EA = 0111\,1101_2 = 125_{10} - 127_{10} = -2_{10}$

  $MA = 1.00110011001100110011010_2 * 2^{-2}$

  $SB = 0_2$

  $EB = 1000\,0111_2 = 135_{10} - 127_{10} = 8_{10}$

  $MB = 1.1111010001_2 * 2^8$

- Sumar ambos exponentes y tomar el resultado provisional.

  $ER = EA + EB = -2_{10} + 8_{10} = 6_{10}$

- Multiplicar ambas mantisas.

  $MR = MA * MB = 1.00110011001100110011010_2 * 1.1111010001_2$

  $MR = 1.00110011001100110011101_2$

- Seleccionar bit de signo.

  $SR = SA\ XOR\ SB = 1\ XOR\ 0 = 1$

- Normalización.

  $R = 1100\,0011\,0001\,0110\,0001\,0011\,0011\,0100_2 = -150.07501_{10}$

El diagrama de flujo correspondiente a esta operación se muestra en la figura 3.33.

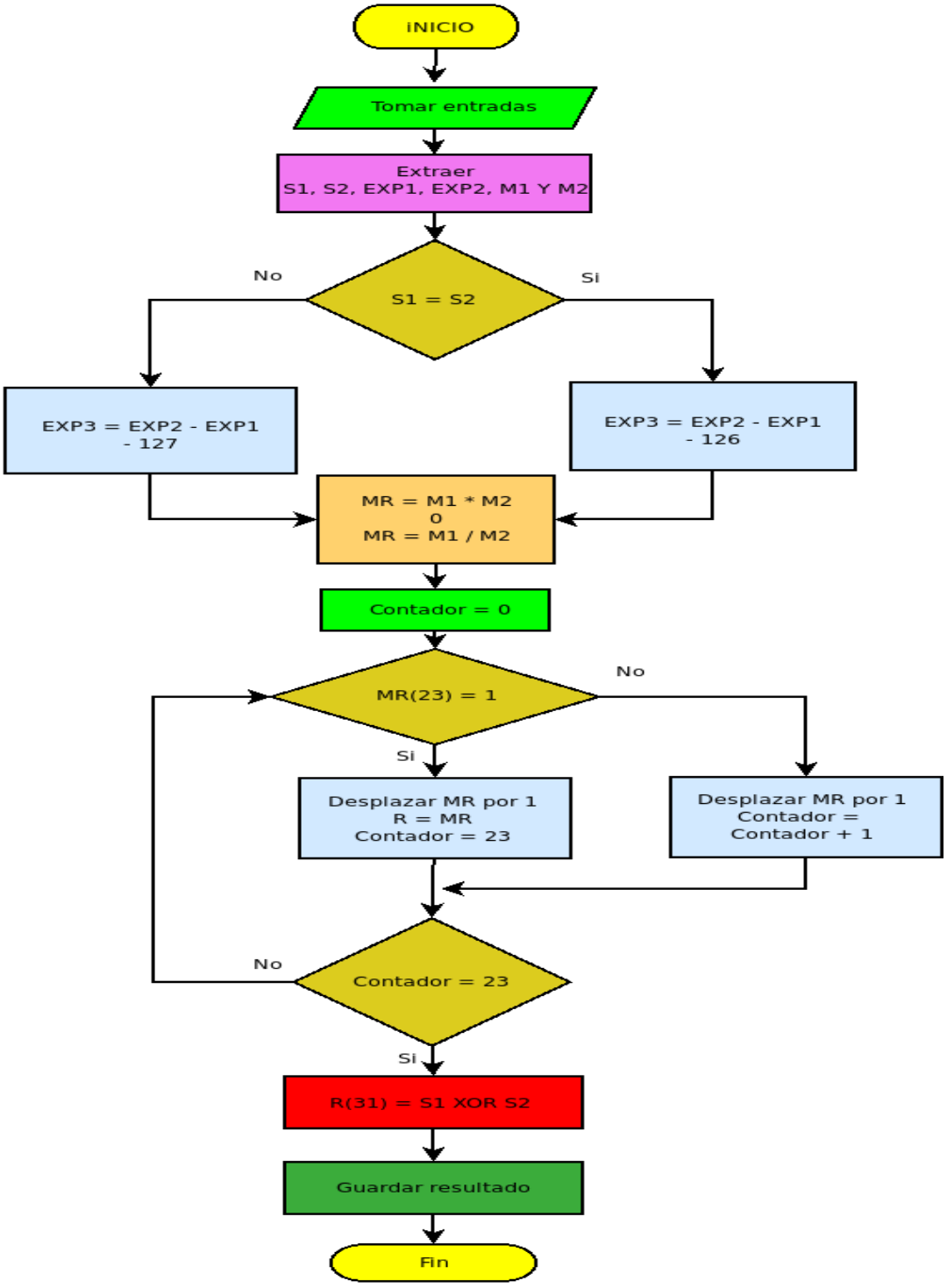


Figura 3.33. Diagrama de flujo de la suma en formato IEEE 754.

Por ultimo se utilizaron los módulos anteriores para implementar el módulo de la multiplicación de matrices por medio de una máquina de estados en VHDL en donde cada estado realiza una tarea en específico como: esperar, multiplicar, sumar, guardar resultado y actualizar los valores de filas y columnas que recorren las matrices a multiplicar.

```
begin
          when waitm --Inicia maquina de estados
                    mat_mult <= multiplica;
          when multiplica --Estado donde se realiza la multiplicación
                    AB = A * B; --Multiplicación en formato IEEE 754
                    mat_mult <= suma
          when suma --Estado donde se realiza la suma
                   --Suma en formato IEEE 754
                    sum = AB + C; --Donde C es un acumulador de suma
                    mat_mult <= resultado
          when resultado --Estado que asigna el valor actual en matriz de resultado
                    res = C
                    mat_mult <= actualizar
          when actualizar --Estado que actualiza posiciones que se deben leer
            if (k < m1col) then
              C <= sum; --Agrega valor
              k := k + 1; --Incrementa columna matriz 1 y fila de matriz 2
              mat_mult <= waitm;
            elsif (j < m2col ) then
              C <= (others => '0'); --Reinicia acumulador
              j := j + 1; --Incrementa columna de matriz 2
              k := 0; --Reinicia columna matriz 1 y fila de matriz 2
              mat_mult <= waitm;
            elsif (i < m1row ) then
              C <= (others => '0'); -Reinicia acumulador
              i := i + 1; --Incrementa fila de matriz 1
              j := 0; --Reinicia columna de matriz 2
              k := 0; --Reinicia columna matriz 1 y fila de matriz 2
              mat_mult <= waitm;
            end if;
end;
```

3.5.3 Función sigmoide con números punto flotante en VHDL

Una operación más compleja que se implementó en PL con números de tipo flotante en formato IEEE 754 fue la función de activación de una matriz, mas específicamente la función sigmoide, que como se vio en el capitulo uno tiene la siguiente forma:

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

Como se puede observar, además de la suma, esta función contiene operaciones como la división y el exponente, las cuales son más complejas de desarrollar con el formato IEEE 754. Afortunadamente, una de las herramientas que el entorno de desarrollo Vivado proporciona es un gran catálogo de módulos IP (Intellectual Property) que pueden agregarse al diseño. Uno de estos módulos es Floating-Point Operator el cual permite escoger entre varias operaciones aritméticas, dentro de las cuales se encuentran la división y el exponente para hacerlas con la representación IEEE 754.

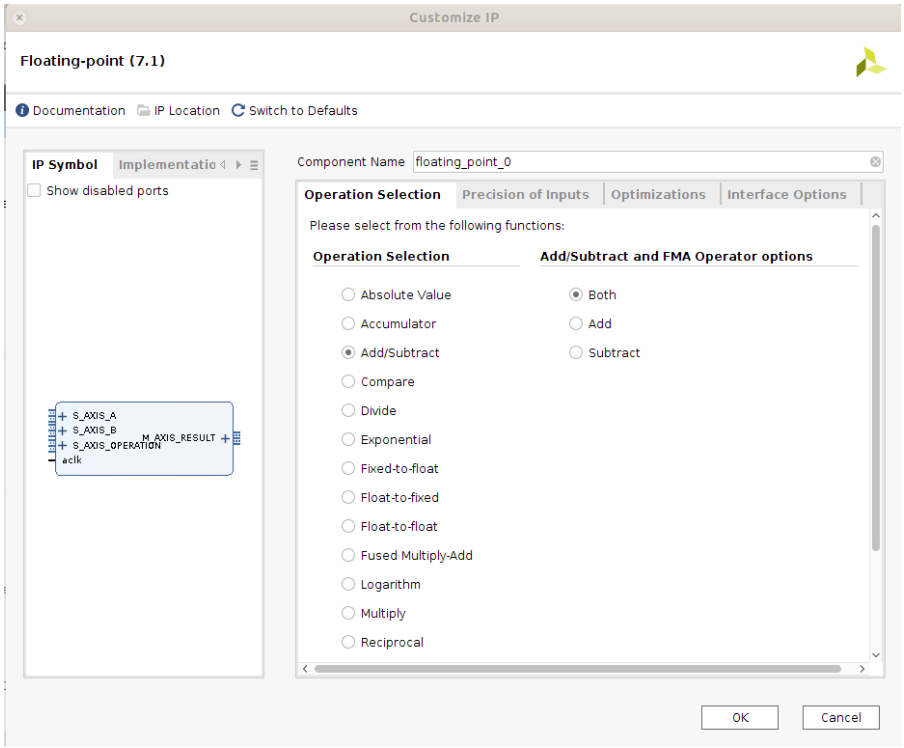


Figura 3.34. Configuración de Floating-point IP.

Ya que se crearon todos los módulos floating-point necesarios, se procedió a realizar instancias dentro del proyecto como cualquier otro bloque, asignando sus respectivas señales de entrada y señales de salida, conectándolos entre sí para poder obtener la función sigmoide de un número.

En la figura 3.35 se puede observar cómo se hizo la conexión de los módulos floating-point para obtener la función sigmoide.

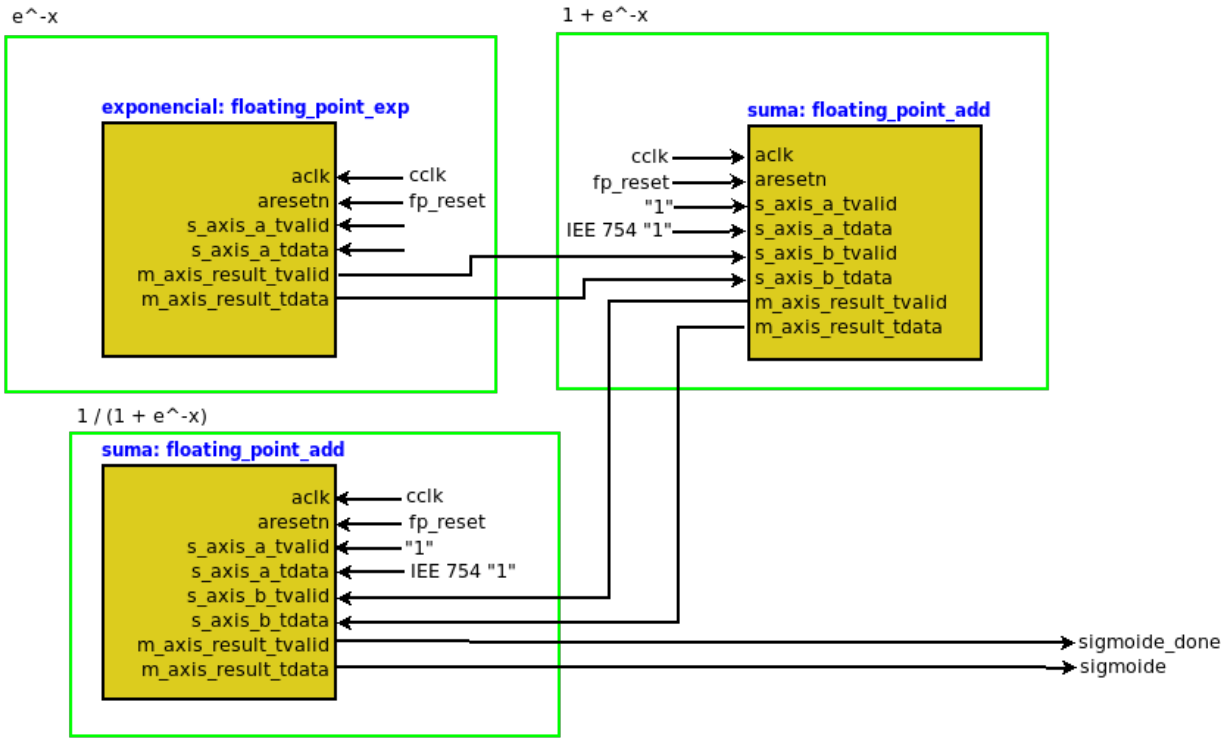


Figura 3.35. Conexión de módulos floating-point IP para obtener función sigmoide.

Una vez que se crearon, se configuraron los módulos necesarios para la función sigmoide (exponencial, suma y división) y se instanciaron en el proyecto, se diseño nuevamente una máquina de estados para ir recorriendo la matriz de entrada a la que se le aplicará la función sigmoide.

```
begin
    when waitm --Inicia maquina de estados
                    mat_sig <= guardar;
    when guardar --Guarda el valor de la función sigmoide en la posición actual de la matriz
                    MR(i)(j) = result; --result = valor actual de función sigmoide
                    mat_sig <= actualiza
    when actualiza --Actualiza los contadores de posición para la matriz de entrada
                    actualiza;
                    mat_sig <= guardar
end;
```

4 Aplicación

Para probar la red neuronal que se ha implementado, se decidió entrenarla con el conjunto de datos "iris"[25] que es un ejemplo clásico para probar algoritmos de aprendizaje automático. El conjunto de datos fue introducido por el biólogo y estadístico Ronald Fisher y contiene 50 muestras de cada una de las tres especies de la flor iris, como se puede ver en la figura 4.1 (Iris setosa, Iris virginica e Iris versicolor), cada muestra incluye las mediciones en centímetros de cuatro rasgos: largo de sépalo, ancho de sépalo, largo de pétalo, ancho de pétalo.

Figura 4.1. Iris setosa, Iris versicolor e Iris virginica.

El entrenamiento se realizó utilizando un 80% del conjunto de datos (40 muestras por especie). Una vez que la red haya sido entrenada, será capaz de clasificar en una de las diferentes especies de flor, nuevas pruebas con valores iguales o diferentes a los que se usaron para su entrenamiento. Se tomará el 20% que se reservó del conjunto de datos del entrenamiento para asegurar que los datos que se ingresen en esta etapa son reales y para poder verificar que la clasificación que la red neuronal realice sea correcta.

25 "UCI Machine Learning Repository: Iris Data Set". [En línea]. Disponible en: http://archive.ics.uci.edu/ml/datasets/iris. [Consultado: 12-abr-2019].

Muestra	Largo sépalo	Ancho sépalo	Largo pétalo	Ancho pétalo	Especie
1	5.1	3.5	1.4	0.2	Iris-setosa
2	4.9	3.0	1.4	0.2	Iris-setosa
3	4.7	3.2	1.3	0.2	Iris-setosa
4	4.6	3.1	1.5	0.2	Iris-setosa
5	5.0	3.6	1.4	0.2	Iris-setosa
6	5.4	3.9	1.7	0.4	Iris-setosa
7	4.6	3.4	1.4	0.3	Iris-setosa
8	5.0	3.4	1.5	0.2	Iris-setosa
9	4.4	2.9	1.4	0.2	Iris-setosa
10	4.9	3.1	1.5	0.1	Iris-setosa
11	5.4	3.7	1.5	0.2	Iris-setosa
12	4.8	3.4	1.6	0.2	Iris-setosa
13	4.8	3.0	1.4	0.1	Iris-setosa
14	4.3	3.0	1.1	0.1	Iris-setosa
15	5.8	4.0	1.2	0.2	Iris-setosa
16	5.7	4.4	1.5	0.4	Iris-setosa
17	5.4	3.9	1.3	0.4	Iris-setosa
18	5.1	3.5	1.4	0.3	Iris-setosa
19	5.7	3.8	1.7	0.3	Iris-setosa
20	5.1	3.8	1.5	0.3	Iris-setosa
21	5.4	3.4	1.7	0.2	Iris-setosa
22	5.1	3.7	1.5	0.4	Iris-setosa
23	4.6	3.6	1.0	0.2	Iris-setosa
24	5.1	3.3	1.7	0.5	Iris-setosa
25	4.8	3.4	1.9	0.2	Iris-setosa
26	5.0	3.0	1.6	0.2	Iris-setosa
27	5.0	3.4	1.6	0.4	Iris-setosa
28	5.2	3.5	1.5	0.2	Iris-setosa
29	5.2	3.4	1.4	0.2	Iris-setosa
30	4.7	3.2	1.6	0.2	Iris-setosa
31	4.8	3.1	1.6	0.2	Iris-setosa
32	5.4	3.4	1.5	0.4	Iris-setosa
33	5.2	4.1	1.5	0.1	Iris-setosa
34	5.5	4.2	1.4	0.2	Iris-setosa
35	4.9	3.1	1.5	0.1	Iris-setosa
36	5.0	3.2	1.2	0.2	Iris-setosa
37	5.5	3.5	1.3	0.2	Iris-setosa
38	4.9	3.1	1.5	0.1	Iris-setosa
39	4.4	3.0	1.3	0.2	Iris-setosa
40	5.1	3.4	1.5	0.2	Iris-setosa
41	5.0	3.5	1.3	0.3	Iris-setosa
42	4.5	2.3	1.3	0.3	Iris-setosa
43	4.4	3.2	1.3	0.2	Iris-setosa
44	5.0	3.5	1.6	0.6	Iris-setosa
45	5.1	3.8	1.9	0.4	Iris-setosa
46	4.8	3.0	1.4	0.3	Iris-setosa
47	5.1	3.8	1.6	0.2	Iris-setosa
48	4.6	3.2	1.4	0.2	Iris-setosa
49	5.3	3.7	1.5	0.2	Iris-setosa
50	5.0	3.3	1.4	0.2	Iris-setosa
51	7.0	3.2	4.7	1.4	Iris-versicolor
52	6.4	3.2	4.5	1.5	Iris-versicolor
53	6.9	3.1	4.9	1.5	Iris-versicolor
54	5.5	2.3	4.0	1.3	Iris-versicolor
55	6.5	2.8	4.6	1.5	Iris-versicolor
56	5.7	2.8	4.5	1.3	Iris-versicolor
57	6.3	3.3	4.7	1.6	Iris-versicolor
58	4.9	2.4	3.3	1.0	Iris-versicolor
59	6.6	2.9	4.6	1.3	Iris-versicolor
60	5.2	2.7	3.9	1.4	Iris-versicolor
61	5.0	2.0	3.5	1.0	Iris-versicolor
62	5.9	3.0	4.2	1.5	Iris-versicolor
63	6.0	2.2	4.0	1.0	Iris-versicolor
64	6.1	2.9	4.7	1.4	Iris-versicolor
65	5.6	2.9	3.6	1.3	Iris-versicolor
66	6.7	3.1	4.4	1.4	Iris-versicolor
67	5.6	3.0	4.5	1.5	Iris-versicolor
68	5.8	2.7	4.1	1.0	Iris-versicolor
69	6.2	2.2	4.5	1.5	Iris-versicolor
70	5.6	2.5	3.9	1.1	Iris-versicolor
71	5.9	3.2	4.8	1.8	Iris-versicolor
72	6.1	2.8	4.0	1.3	Iris-versicolor
73	6.3	2.5	4.9	1.5	Iris-versicolor
74	6.1	2.8	4.7	1.2	Iris-versicolor
75	6.4	2.9	4.3	1.3	Iris-versicolor
76	6.6	3.0	4.4	1.4	Iris-versicolor
77	6.8	2.8	4.8	1.4	Iris-versicolor

78	6.7	3.0	5.0	1.7	Iris-versicolor
79	6.0	2.9	4.5	1.5	Iris-versicolor
80	5.7	2.6	3.5	1.0	Iris-versicolor
81	5.5	2.4	3.8	1.1	Iris-versicolor
82	5.5	2.4	3.7	1.0	Iris-versicolor
83	5.8	2.7	3.9	1.2	Iris-versicolor
84	6.0	2.7	5.1	1.6	Iris-versicolor
85	5.4	3.0	4.5	1.5	Iris-versicolor
86	6.0	3.4	4.5	1.6	Iris-versicolor
87	6.7	3.1	4.7	1.5	Iris-versicolor
88	6.3	2.3	4.4	1.3	Iris-versicolor
89	5.6	3.0	4.1	1.3	Iris-versicolor
90	5.5	2.5	4.0	1.3	Iris-versicolor
91	5.5	2.6	4.4	1.2	Iris-versicolor
92	6.1	3.0	4.6	1.4	Iris-versicolor
93	5.8	2.6	4.0	1.2	Iris-versicolor
94	5.0	2.3	3.3	1.0	Iris-versicolor
95	5.6	2.7	4.2	1.3	Iris-versicolor
96	5.7	3.0	4.2	1.2	Iris-versicolor
97	5.7	2.9	4.2	1.3	Iris-versicolor
98	6.2	2.9	4.3	1.3	Iris-versicolor
99	5.1	2.5	3.0	1.1	Iris-versicolor
100	5.7	2.8	4.1	1.3	Iris-versicolor
101	6.3	3.3	6.0	2.5	Iris-virginica
102	5.8	2.7	5.1	1.9	Iris-virginica
103	7.1	3.0	5.9	2.1	Iris-virginica
104	6.3	2.9	5.6	1.8	Iris-virginica
105	6.5	3.0	5.8	2.2	Iris-virginica
106	7.6	3.0	6.6	2.1	Iris-virginica
107	4.9	2.5	4.5	1.7	Iris-virginica
108	7.3	2.9	6.3	1.8	Iris-virginica
109	6.7	2.5	5.8	1.8	Iris-virginica
110	7.2	3.6	6.1	2.5	Iris-virginica
111	6.5	3.2	5.1	2.0	Iris-virginica
112	6.4	2.7	5.3	1.9	Iris-virginica
113	6.8	3.0	5.5	2.1	Iris-virginica
114	5.7	2.5	5.0	2.0	Iris-virginica
115	5.8	2.8	5.1	2.4	Iris-virginica
116	6.4	3.2	5.3	2.3	Iris-virginica
117	6.5	3.0	5.5	1.8	Iris-virginica
118	7.7	3.8	6.7	2.2	Iris-virginica
119	7.7	2.6	6.9	2.3	Iris-virginica
120	6.0	2.2	5.0	1.5	Iris-virginica
121	6.9	3.2	5.7	2.3	Iris-virginica
122	5.6	2.8	4.9	2.0	Iris-virginica
123	7.7	2.8	6.7	2.0	Iris-virginica
124	6.3	2.7	4.9	1.8	Iris-virginica
125	6.7	3.3	5.7	2.1	Iris-virginica
126	7.2	3.2	6.0	1.8	Iris-virginica
127	6.2	2.8	4.8	1.8	Iris-virginica
128	6.1	3.0	4.9	1.8	Iris-virginica
129	6.4	2.8	5.6	2.1	Iris-virginica
130	7.2	3.0	5.8	1.6	Iris-virginica
131	7.4	2.8	6.1	1.9	Iris-virginica
132	7.9	3.8	6.4	2.0	Iris-virginica
133	6.4	2.8	5.6	2.2	Iris-virginica
134	6.3	2.8	5.1	1.5	Iris-virginica
135	6.1	2.6	5.6	1.4	Iris-virginica
136	7.7	3.0	6.1	2.3	Iris-virginica
137	6.3	3.4	5.6	2.4	Iris-virginica
138	6.4	3.1	5.5	1.8	Iris-virginica
139	6.0	3.0	4.8	1.8	Iris-virginica
140	6.9	3.1	5.4	2.1	Iris-virginica
141	6.7	3.1	5.6	2.4	Iris-virginica
142	6.9	3.1	5.1	2.3	Iris-virginica
143	5.8	2.7	5.1	1.9	Iris-virginica
144	6.8	3.2	5.9	2.3	Iris-virginica
145	6.7	3.3	5.7	2.5	Iris-virginica
146	6.7	3.0	5.2	2.3	Iris-virginica
147	6.3	2.5	5.0	1.9	Iris-virginica
148	6.5	3.0	5.2	2.0	Iris-virginica
149	6.2	3.4	5.4	2.3	Iris-virginica
150	5.9	3.0	5.1	1.8	Iris-virginica

Tabla 4.1. Conjunto de datos de entrenamiento.

5 Protocolo de pruebas

En este apartado se describe detalladamente el proceso de prueba del sistema que se implementó, utilizando el conjunto de datos de la flor Iris descrito en el apartado anterior para la clasificación de nuevas muestras.

Antes de programar la tarjeta de desarrollo ZYBO-Z7 es necesario agregar la configuración que tendrá la red neuronal y agregar el conjunto de datos de entrenamiento.

La configuración se debe realizar tanto en el PS (sofware) como en el PL (hardware) y deben de coincidir ambas partes, de lo contrario el funcionamiento será erróneo. La configuración que se ingresó fue para una red multicapa con una capa oculta y con los parámetros siguientes:

- 4 neuronas en la capa de entrada correspondiente al número de datos de entrada, que en el caso de esta aplicación son: largo de sépalo, ancho de sépalo, largo de pétalo, ancho de pétalo.
- 8 neuronas en la capa oculta. El número de elementos se determinó tras comparar con otros valores, comprobando que este valor ofrecía el rendimiento mas óptimo para una capa oculta.
- 3 neuronas en la capa de salida que es equivalente al número de salidas que se obtendrán. En este ejemplo la salida solo es una (o es Iris setosa, Iris virginica o Iris versicolor) pero la salida se codificará asignando 1 bit para cada opción, por lo tanto se necesitan 3 salidas para representar las tres opciones.
- 120 muestras de entrenamiento que contienen 40 casos de cada uno de los tres tipos de especie de la flor Iris.
- Factor de aprendizaje de 0.3 elegido tras realizar varias pruebas y verificar que es un valor óptimo para el entrenamiento de esta aplicación.
- Pérdida de 10% que representa la aproximación que tendrá en entrenamiento de la red con el conjunto de datos de entrenamiento. Se ha probado que para esta aplicación el porcentaje elegido es suficiente para realizar pruebas de manera eficiente sin necesidad

de disminuirlo, ya que además de ser innecesario implicaría un tiempo de entrenamiento mayor.

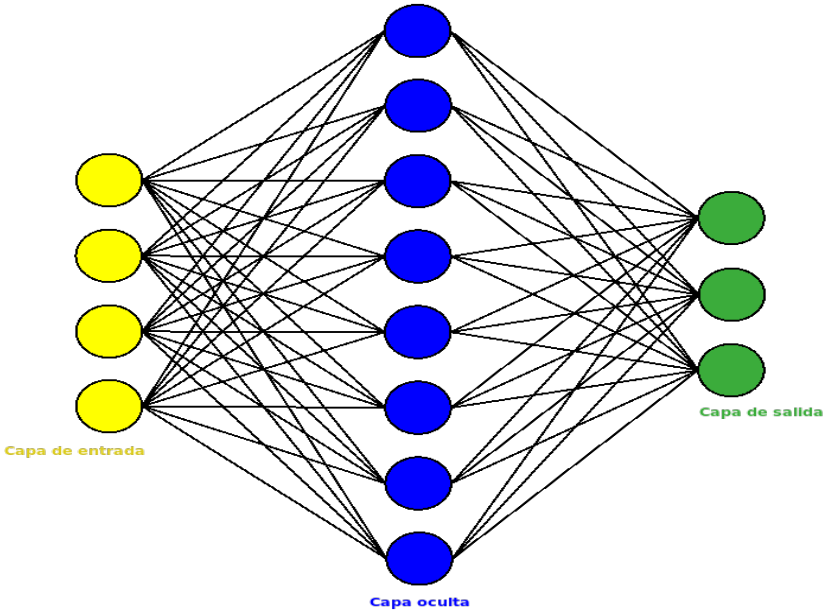


Figura 5.1. Red neuronal para clasificación de flor Iris.

El conjunto de datos que se usará para el entrenamiento de la red, debe de estar previamente codificado y normalizado, ya que la etapa de pre-procesamiento no esta implementada.

Debido a que todos los datos del conjunto a excepción de "Especie" son cuantitativos, se deben normalizar para escalarlos. La tabla 5.1 muestra los valores que se obtuvieron al calcular el promedio y la desviación estándar de cada una de las entradas (columnas) del conjunto de datos, estos valores servirán para normalizar todos los valores que ingresen a la red neuronal.

	Largo sépalo	Ancho sépalo	Largo pétalo	Ancho pétalo
Promedio	5.89000	3.06000	3.79583	1.19083
Desviación estandar	0.84589	0.44181	1.79286	0.75737

Tabla 5.1. Promedio y desviación estándar.

Para codificar los datos de "Especie" los cuales son categóricos, se agrega un bit por cada opción y se asignan de la manera siguiente:

Iris setosa = [0, 0, 1], Iris virginica = [0, 1, 0], Iris versicolor = [1, 0, 0]

El conjunto de datos ya está listo para agregarse a la red queda, como se muestra en la tabla 5.2

Largo sépalo	Ancho sépalo	Largo pétalo	Ancho pétalo	Especie
-0,93393	0,99591	-1,33632	-1,30825	[0, 0, 1]
-1,17037	-0,13581	-1,33632	-1,30825	[0, 0, 1]
-1,40680	0,31688	-1,39209	-1,30825	[0, 0, 1]
-1,52502	0,09054	-1,28054	-1,30825	[0, 0, 1]
-1,05215	1,22225	-1,33632	-1,30825	[0, 0, 1]
-0,57927	1,90128	-1,16899	-1,04418	[0, 0, 1]
-1,52502	0,76957	-1,33632	-1,17622	[0, 0, 1]
-1,05215	0,76957	-1,28054	-1,30825	[0, 0, 1]
-1,76146	-0,36215	-1,33632	-1,30825	[0, 0, 1]
-1,17037	0,09054	-1,28054	-1,44029	[0, 0, 1]
-0,57927	1,44860	-1,28054	-1,30825	[0, 0, 1]
-1,28858	0,76957	-1,22476	-1,30825	[0, 0, 1]
-1,28858	-0,13581	-1,33632	-1,44029	[0, 0, 1]
-1,87968	-0,13581	-1,50365	-1,44029	[0, 0, 1]
-0,10640	2,12763	-1,44787	-1,30825	[0, 0, 1]
-0,22462	3,03300	-1,28054	-1,04418	[0, 0, 1]
-0,57927	1,90128	-1,39209	-1,04418	[0, 0, 1]
-0,93393	0,99591	-1,33632	-1,17622	[0, 0, 1]
-0,22462	1,67494	-1,16899	-1,17622	[0, 0, 1]
-0,93393	1,67494	-1,28054	-1,17622	[0, 0, 1]
-0,57927	0,76957	-1,16899	-1,30825	[0, 0, 1]
-0,93393	1,44860	-1,28054	-1,04418	[0, 0, 1]
-1,52502	1,22225	-1,55943	-1,30825	[0, 0, 1]
-0,93393	0,54322	-1,16899	-0,91215	[0, 0, 1]
-1,28858	0,76957	-1,05743	-1,30825	[0, 0, 1]
-1,05215	-0,13581	-1,22476	-1,30825	[0, 0, 1]
-1,05215	0,76957	-1,22476	-1,04418	[0, 0, 1]
-0,81571	0,99591	-1,28054	-1,30825	[0, 0, 1]
-0,81571	0,76957	-1,33632	-1,30825	[0, 0, 1]
-1,40680	0,31688	-1,22476	-1,30825	[0, 0, 1]
-1,28858	0,09054	-1,22476	-1,30825	[0, 0, 1]
-0,57927	0,76957	-1,28054	-1,04418	[0, 0, 1]
-0,81571	2,35397	-1,28054	-1,44029	[0, 0, 1]
-0,46105	2,58031	-1,33632	-1,30825	[0, 0, 1]
-1,17037	0,09054	-1,28054	-1,44029	[0, 0, 1]
-1,05215	0,31688	-1,44787	-1,30825	[0, 0, 1]
-0,46105	0,99591	-1,39209	-1,30825	[0, 0, 1]
-1,17037	0,09054	-1,28054	-1,44029	[0, 0, 1]
-1,76146	-0,13581	-1,39209	-1,30825	[0, 0, 1]
-0,93393	0,76957	-1,28054	-1,30825	[0, 0, 1]
1,31223	0,31688	0,50431	0,27617	[0, 1, 0]
0,60292	0,31688	0,39276	0,40821	[0, 1, 0]
1,19401	0,09054	0,61587	0,40821	[0, 1, 0]
-0,46105	-1,72021	0,11388	0,14414	[0, 1, 0]
0,72113	-0,58849	0,44854	0,40821	[0, 1, 0]
-0,22462	-0,58849	0,39276	0,14414	[0, 1, 0]
0,48470	0,54322	0,50431	0,54025	[0, 1, 0]
-1,17037	-1,49387	-0,27656	-0,25197	[0, 1, 0]
0,83935	-0,36215	0,44854	0,14414	[0, 1, 0]
-0,81571	-0,81484	0,05810	0,27617	[0, 1, 0]
-1,05215	-2,39924	-0,16501	-0,25197	[0, 1, 0]
0,01182	-0,13581	0,22543	0,40821	[0, 1, 0]
0,13004	-1,94655	0,11388	-0,25197	[0, 1, 0]
0,24826	-0,36215	0,50431	0,27617	[0, 1, 0]
-0,34283	-0,36215	-0,10923	0,14414	[0, 1, 0]
0,95757	0,09054	0,33698	0,27617	[0, 1, 0]
-0,34283	-0,13581	0,39276	0,40821	[0, 1, 0]
-0,10640	-0,81484	0,16965	-0,25197	[0, 1, 0]

0,36648	-1,94655	0,39276	0,40821	[0, 1, 0]
-0,34283	-1,26752	0,05810	-0,11993	[0, 1, 0]
0,01182	0,31688	0,56009	0,80432	[0, 1, 0]
0,24826	-0,58849	0,11388	0,14414	[0, 1, 0]
0,48470	-1,26752	0,61587	0,40821	[0, 1, 0]
0,24826	-0,58849	0,50431	0,01210	[0, 1, 0]
0,60292	-0,36215	0,28121	0,14414	[0, 1, 0]
0,83935	-0,13581	0,33698	0,27617	[0, 1, 0]
1,07579	-0,58849	0,56009	0,27617	[0, 1, 0]
0,95757	-0,13581	0,67165	0,67228	[0, 1, 0]
0,13004	-0,36215	0,39276	0,40821	[0, 1, 0]
-0,22462	-1,04118	-0,16501	-0,25197	[0, 1, 0]
-0,46105	-1,49387	0,00232	-0,11993	[0, 1, 0]
-0,46105	-1,49387	-0,05345	-0,25197	[0, 1, 0]
-0,10640	-0,81484	0,05810	0,01210	[0, 1, 0]
0,13004	-0,81484	0,72742	0,54025	[0, 1, 0]
-0,57927	-0,13581	0,39276	0,40821	[0, 1, 0]
0,13004	0,76957	0,39276	0,54025	[0, 1, 0]
0,95757	0,09054	0,50431	0,40821	[0, 1, 0]
0,48470	-1,72021	0,33698	0,14414	[0, 1, 0]
-0,34283	-0,13581	0,16965	0,14414	[0, 1, 0]
-0,46105	-1,26752	0,11388	0,14414	[0, 1, 0]
0,48470	0,54322	1,22941	1,72856	[1, 0, 0]
-0,10640	-0,81484	0,72742	0,93635	[1, 0, 0]
1,43045	-0,13581	1,17364	1,20042	[1, 0, 0]
0,48470	-0,36215	1,00631	0,80432	[1, 0, 0]
0,72113	-0,13581	1,11786	1,33246	[1, 0, 0]
2,02154	-0,13581	1,56407	1,20042	[1, 0, 0]
-1,17037	-1,26752	0,39276	0,67228	[1, 0, 0]
1,66688	-0,36215	1,39674	0,80432	[1, 0, 0]
0,95757	-1,26752	1,11786	0,80432	[1, 0, 0]
1,54867	1,22225	1,28519	1,72856	[1, 0, 0]
0,72113	0,31688	0,72742	1,06839	[1, 0, 0]
0,60292	-0,81484	0,83898	0,93635	[1, 0, 0]
1,07579	-0,13581	0,95053	1,20042	[1, 0, 0]
-0,22462	-1,26752	0,67165	1,06839	[1, 0, 0]
-0,10640	-0,58849	0,72742	1,59653	[1, 0, 0]
0,60292	0,31688	0,83898	1,46449	[1, 0, 0]
0,72113	-0,13581	0,95053	0,80432	[1, 0, 0]
2,13976	1,67494	1,61985	1,33246	[1, 0, 0]
2,13976	-1,04118	1,73140	1,46449	[1, 0, 0]
0,13004	-1,94655	0,67165	0,40821	[1, 0, 0]
1,19401	0,31688	1,06208	1,46449	[1, 0, 0]
-0,34283	-0,58849	0,61587	1,06839	[1, 0, 0]
2,13976	-0,58849	1,61985	1,06839	[1, 0, 0]
0,48470	-0,81484	0,61587	0,80432	[1, 0, 0]
0,95757	0,54322	1,06208	1,20042	[1, 0, 0]
1,54867	0,31688	1,22941	0,80432	[1, 0, 0]
0,36648	-0,58849	0,56009	0,80432	[1, 0, 0]
0,24826	-0,13581	0,61587	0,80432	[1, 0, 0]
0,60292	-0,58849	1,00631	1,20042	[1, 0, 0]
1,54867	-0,13581	1,11786	0,54025	[1, 0, 0]
1,78510	-0,58849	1,28519	0,93635	[1, 0, 0]
2,37620	1,67494	1,45252	1,06839	[1, 0, 0]
0,60292	-0,58849	1,00631	1,33246	[1, 0, 0]
0,48470	-0,58849	0,72742	0,40821	[1, 0, 0]
0,24826	-1,04118	1,00631	0,27617	[1, 0, 0]
2,13976	-0,13581	1,28519	1,46449	[1, 0, 0]
0,48470	0,76957	1,00631	1,59653	[1, 0, 0]
0,60292	0,09054	0,95053	0,80432	[1, 0, 0]
0,13004	-0,13581	0,56009	0,80432	[1, 0, 0]
1,19401	0,09054	0,89475	1,20042	[1, 0, 0]

Tabla 5.2. Conjuntos de datos de entrenamiento codificado y normalizado.

Ya que se ha configurado el sistema y se ha agregado el conjunto de datos de entrenamiento codificado y normalizado, se procede a programar PL y PS desde el entorno de desarrollo SDK.

Para probar la red neuronal se utilizarán los datos de la tabla 5.3, los cuales no fueron incluidos en el entrenamiento de la red. Los resultados de clasificación deberán coincidir para demostrar que la red funciona correctamente.

Largo sépalo	Ancho sépalo	Largo pétalo	Ancho pétalo	Especie
5.0	3.5	1.3	0.3	Iris-setosa
4.5	2.3	1.3	0.3	Iris-setosa
4.4	3.2	1.3	0.2	Iris-setosa
5.0	3.5	1.6	0.6	Iris-setosa
5.1	3.8	1.9	0.4	Iris-setosa
4.8	3.0	1.4	0.3	Iris-setosa
5.1	3.8	1.6	0.2	Iris-setosa
4.6	3.2	1.4	0.2	Iris-setosa
5.3	3.7	1.5	0.2	Iris-setosa
5.0	3.3	1.4	0.2	Iris-setosa
5.5	2.6	4.4	1.2	Iris-versicolor
6.1	3.0	4.6	1.4	Iris-versicolor
5.8	2.6	4.0	1.2	Iris-versicolor
5.0	2.3	3.3	1.0	Iris-versicolor
5.6	2.7	4.2	1.3	Iris-versicolor
5.7	3.0	4.2	1.2	Iris-versicolor
5.7	2.9	4.2	1.3	Iris-versicolor
6.2	2.9	4.3	1.3	Iris-versicolor
5.1	2.5	3.0	1.1	Iris-versicolor
5.7	2.8	4.1	1.3	Iris-versicolor
6.7	3.1	5.6	2.4	Iris-virginica
6.9	3.1	5.1	2.3	Iris-virginica
5.8	2.7	5.1	1.9	Iris-virginica
6.8	3.2	5.9	2.3	Iris-virginica
6.7	3.3	5.7	2.5	Iris-virginica
6.7	3.0	5.2	2.3	Iris-virginica
6.3	2.5	5.0	1.9	Iris-virginica
6.5	3.0	5.2	2.0	Iris-virginica
6.2	3.4	5.4	2.3	Iris-virginica
5.9	3.0	5.1	1.8	Iris-virginica

Tabla 5.3. Conjuntos de datos para prueba de la red neuronal.

A diferencia de los datos de entrenamiento, a los datos de prueba ya no es necesario pre-procesarlos ya que el programa en PS se encarga de hacerlo con los valores de los promedios y desviacion estándar que se calcularon y se cargaron previamente.

La siguiente prueba se dividirá en tres partes: prueba sin red entrenada, entrenamiento de red neuronal y prueba con con red entrenada, la interacción con el usuario para ingresar datos y para mostrar resultados se dará a través de la terminal que SDK tiene incluido.

Prueba sin red neuronal entrenada

Para entrar en modo prueba, el switch de la tarjeta debe de estar en "1", como el sistema acaba de iniciarse procederá a solicitar valores para iniciar propagación hacia adelante sin haber entrenado la red antes, como se puede deja ver en la figura 5.2.

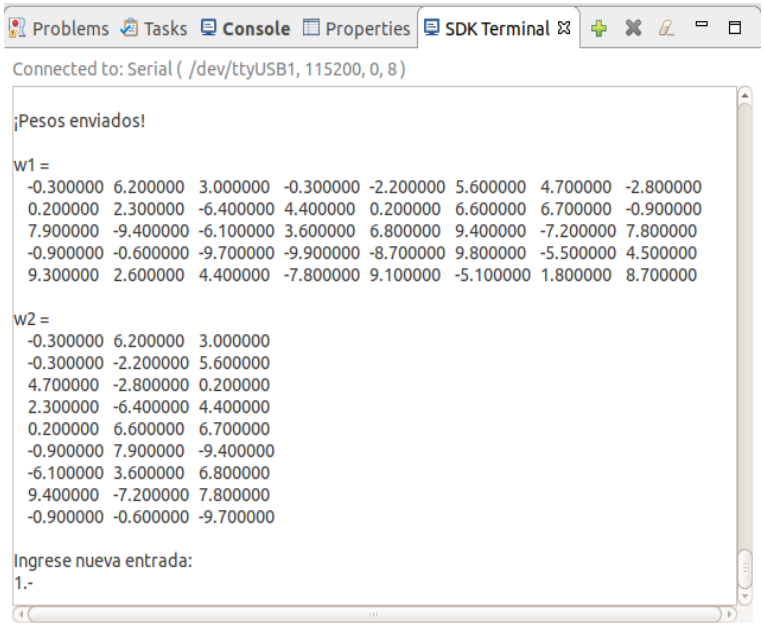


Figura 5.2. Pesos enviados a PL para modo prueba.

En este caso los pesos que se enviaron a PL son totalmente aleatorios, lo que significa que la salida de la red después de que PL haya terminado la propagación hacia adelante estará basada en estos números aleatorios.

Se ingresarán los siguientes valores tomados de la tabla de pruebas para comprobar que la red no está entrenada: largo de sépalo = 5, ancho de sépalo = 3.5, largo de pétalo = 1.5 y ancho de pétalo = 0.3 y la salida correspondiente debería ser Iris setosa. Observar figura 5.3

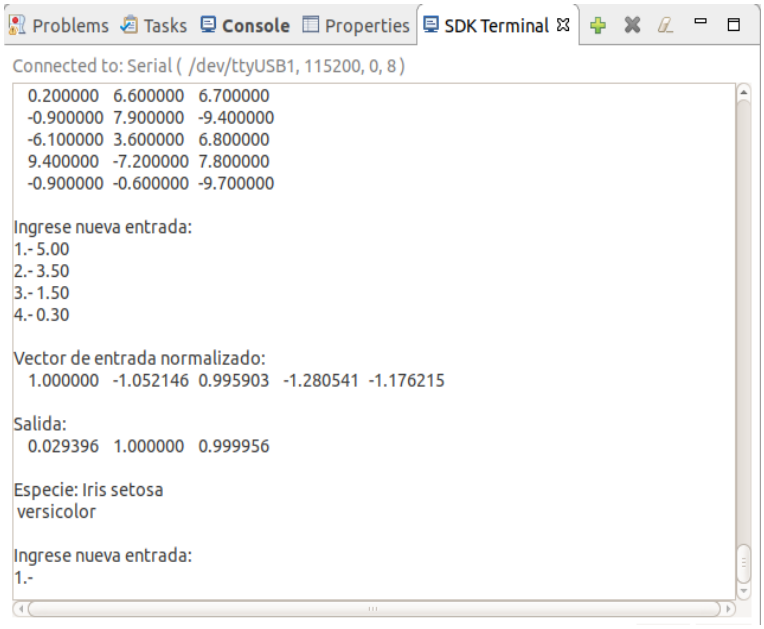


Figura 5.3. Resultados sin entrenamiento.

El vector de salida es [0, 1, 1], como ya se había anticipado la salida es errónea ya que a pesar de que clasifica la entrada como Iris setosa, también la clasifica como Iris versicolor, por cual no podemos decir sea una clasificación es correcta.

Entrenamiento de red neuronal

Para hacer que el sistema realice el entrenamiento de la red neuronal, es necesario pasar a “0” el switch de modo, lo cuál indica al sistema que debe pasar a modo entrenamiento. Inmediatamente el software toma los pesos que se habían generado aleatoriamente y los datos de entrenamiento e inicia el entrenamiento de la red, deteniendo el proceso hasta que el porcentaje de perdida sea menor que el 10% que es el que se fijo para esta aplicación. Una vez finalizado el entrenamiento el sistema actualiza las matrices de pesos, dejándolas listas para enviarla a PL y realizar una nueva prueba. Este proceso se puede observar en la figura 5.4.

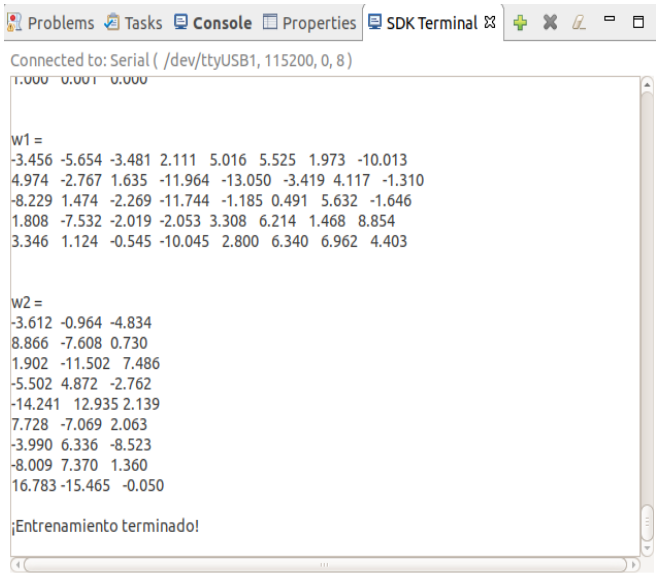


Figura 5.4. Actualización de pesos [w] después del entrenamiento.

Prueba con red neuronal entrenada

Pasamos el sistema de nuevo a modo prueba regresando el switch a la posición inicial (“1”). Enseguida se envían a PL las nuevas matrices de pesos [w] ajustadas y el sistema solicita una entrada nueva. Introduciremos la misma muestra que utilizamos para la red no entrenada.

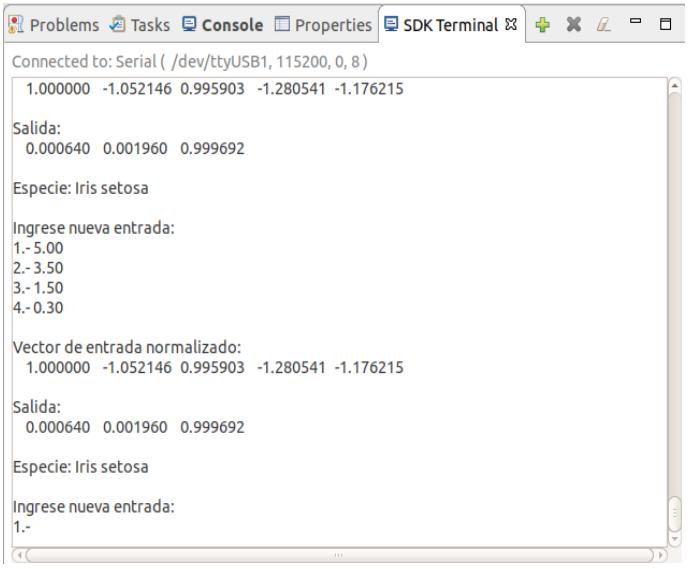


Figura 5.5. Resultados con entrenamiento.

El vector de salida tiende a ser [0, 0, 1] correspondiente a la especie Iris setosa, que concuerda especie de la muestra que se ingresó, es decir que la clasificación se ha realizado exitosamente.

En las figuras 5.6 y 5.7 se ven los resultados de dos pruebas mas con los datos de prueba, uno para Iris versicolor y otro para Iris virginica respectivamente.

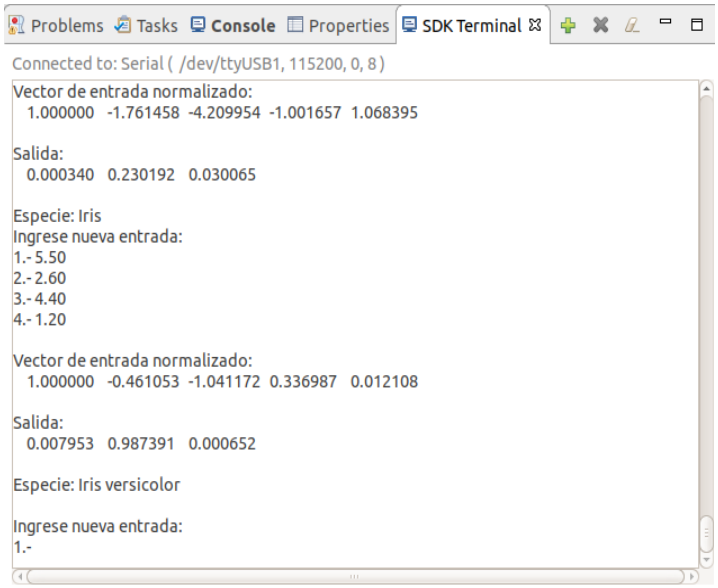


Figura 5.6. Resultados con entrenamiento(2).

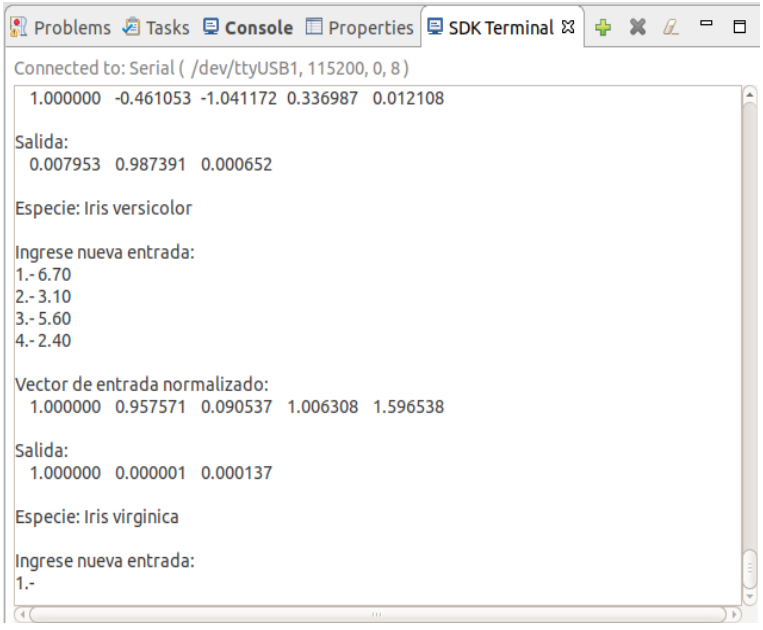


Figura 5.7. Resultados con entrenamiento(3).

De igual forma se comprobó el resto de muestras de prueba de la tabla 6.3 y todos los resultados fueron correctos, la tabla 5.4 agrega las salidas que devolvió el sistema después de haber realizado la propagación hacia adelante.

Muestras de prueba					Clasificación por Red Neuronal	
Largo sépalo	Ancho sépalo	Largo pétalo	Ancho pétalo	Especie	Vector de salida	Especie
5.0	3.5	1.3	0.3	Iris-setosa	[0.00064, 0.00196, 0.99969]	Iris-setosa
4.5	2.3	1.3	0.3	Iris-setosa	[0.00001, 0.06364, 0.99751]	Iris-setosa
4.4	3.2	1.3	0.2	Iris-setosa	[0.00059, 0.00208, 0.99967]	Iris-setosa
5.0	3.5	1.6	0.6	Iris-setosa	[0.00047, 0.00293, 0.99955]	Iris-setosa
5.1	3.8	1.9	0.4	Iris-setosa	[0.00011, 0.01056, 0.99972]	Iris-setosa
4.8	3.0	1.4	0.3	Iris-setosa	[0.00035, 0.00335, 0.99957]	Iris-setosa
5.1	3.8	1.6	0.2	Iris-setosa	[0.00052, 0.00235, 0.99971]	Iris-setosa
4.6	3.2	1.4	0.2	Iris-setosa	[0.00057, 0.00215, 0.99967]	Iris-setosa
5.3	3.7	1.5	0.2	Iris-setosa	[0.00050, 0.00241, 0.99970]	Iris-setosa
5.0	3.3	1.4	0.2	Iris-setosa	[0.00054, 0.00225, 0.99966]	Iris-setosa
5.5	2.6	4.4	1.2	Iris-versicolor	[0.00795, 0.98731, 0.00065]	Iris-versicolor
6.1	3.0	4.6	1.4	Iris-versicolor	[0.13322, 0.83754, 0.00028]	Iris-versicolor
5.8	2.6	4.0	1.2	Iris-versicolor	[0.00431, 0.99269, 0.00051]	Iris-versicolor
5.0	2.3	3.3	1.0	Iris-versicolor	[0.00186, 0.99525, 0.00039]	Iris-versicolor
5.6	2.7	4.2	1.3	Iris-versicolor	[0.00309, 0.99453, 0.00070]	Iris-versicolor
5.7	3.0	4.2	1.2	Iris-versicolor	[0.00000, 1.00000, 0.00095]	Iris-versicolor
5.7	2.9	4.2	1.3	Iris-versicolor	[0.00000, 0.99999, 0.00101]	Iris-versicolor
6.2	2.9	4.3	1.3	Iris-versicolor	[0.00136, 0.99761, 0.00060]	Iris-versicolor
5.1	2.5	3.0	1.1	Iris-versicolor	[0.00059, 0.99401, 0.00117]	Iris-versicolor
5.7	2.8	4.1	1.3	Iris-versicolor	[0.00013, 0.99967, 0.00084]	Iris-versicolor
6.7	3.1	5.6	2.4	Iris-virginica	[1.00000 ,0.00000, 0.00013]	Iris-virginica
6.9	3.1	5.1	2.3	Iris-virginica	[0.99911, 0.00164, 0.00004]	Iris-virginica
5.8	2.7	5.1	1.9	Iris-virginica	[0.98407, 0.02090, 0.00217]	Iris-virginica
6.8	3.2	5.9	2.3	Iris-virginica	[0.99999, 0.00001, 0.00006]	Iris-virginica
6.7	3.3	5.7	2.5	Iris-virginica	[1.00000, 0.00000, 0.00013]	Iris-virginica
6.7	3.0	5.2	2.3	Iris-virginica	[0.99999, 0.00001, 0.00008]	Iris-virginica
6.3	2.5	5.0	1.9	Iris-virginica	[0.99818, 0.00291, 0.00080]	Iris-virginica
6.5	3.0	5.2	2.0	Iris-virginica	[0.99999, 0.00001, 0.00022]	Iris-virginica
6.2	3.4	5.4	2.3	Iris-virginica	[0.99996, 0.00005, 0.00019]	Iris-virginica
5.9	3.0	5.1	1.8	Iris-virginica	[0.99993, 0.00013, 0.00031]	Iris-virginica

Tabla 5.4. Salidas de red neuronal entrenada.

6 Análisis de resultados

Una de las características más importantes a la hora de diseñar una RNA es el número de capas ocultas y neuronas que ésta tendrá, pues de esto depende el aprendizaje de la red. Tanto las capas ocultas como el número de neuronas puede variar dependiendo de la aplicación que se tenga.

Para nuestra aplicación se diseñó en un principio una red neuronal de una capa oculta (Figura 6.1) y posteriormente una de dos capas ocultas(Figura 6.2), en ambas lograba dar respuesta al problema planteado, sin embargo, en cuanto a recursos utilizados de la tarjeta, la red de dos capas ocultas utilizaba muchos mas recursos en PL que la de una. Observe las siguientes imágenes.

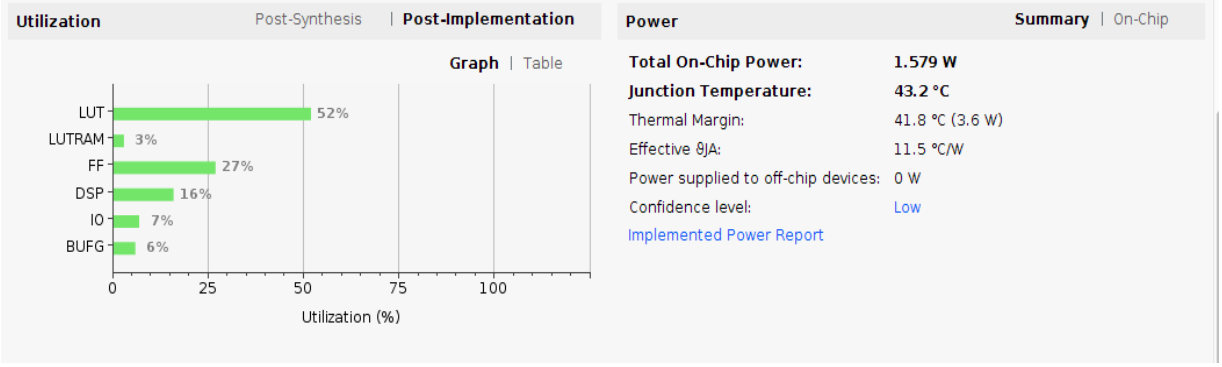


Figura 6.1. Red Neuronal de una capa oculta.

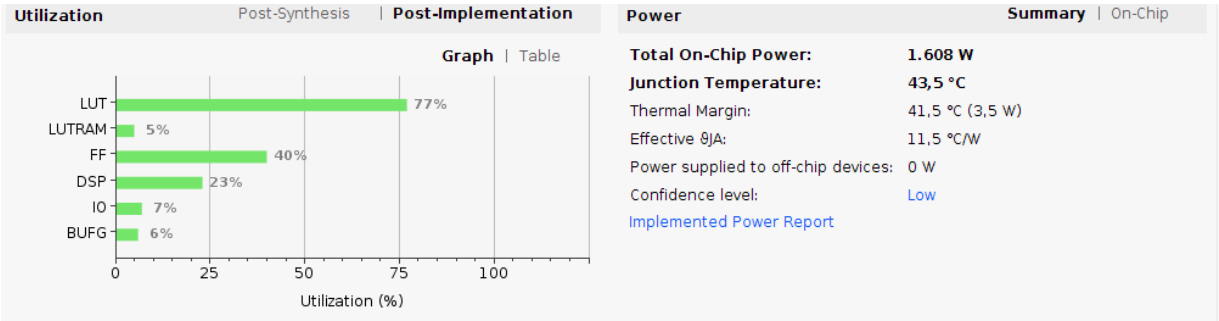


Figura 6.2. Red Neuronal de dos capas ocultas.

Como se puede observar la cantidad de bloques lógicos que requiere una red neuronal con dos capas ocultas es más que con la de una y en ambos casos la salida que arroja es la deseada, es por eso que se decidió implementar la red neuronal de una sola capa oculta pues resuelve el problema y ocupa menos recursos.

El número de neuronas dentro de la capa oculta se puede elegir mediante la regla de la pirámide geométrica. la cual dice que para una red de tres capas (una sola capa oculta) como es nuestro caso:

$$h=\sqrt{m\times n}$$

Donde:

-n es el número de neuronas de entrada.

-m es el número de neuronas de salida.

-h es el número inicial de neuronas en la capa oculta.

Para nuestra aplicación tenemos 4 entradas y 3 salidas. Según la fórmula anterior el número de neuronas en la capa oculta es igual a 3.46 por lo que se probó con 3 y 4 neuronas respectivamente, no obstante, la red bajo esta configuración y para el caso de 3 neuronas no lograba aprender mientras que con 4 tardaba demasiado en arrojar el resultado, esto nos llevó a aumentar el número de neuronas hasta encontrar un funcionamiento óptimo, el cual fue cuando llegó a 8 neuronas.

En las siguientes imágenes de la terminal se muestran los valores que la RNA obtiene durante el entrenamiento, con los cuales podemos ver como funcionan los parámetros establecidos.

- Epoch (épocas). Muestra el número de iteraciones que ha realizado la red, hasta que el entrenamiento se termine. A menor número de épocas, menos tiempo de entrenamiento.
- Loss (pérdida). Muestra el porcentaje de error actual de los pesos con respecto al conjunto de datos de entrenamiento. El usuario define el porcentaje de perdida con el cual el entrenamiento terminará. Para este caso se definió con un valor de 10.

Con 3 neuronas, se toma 73487 épocas y no se resuelve, la pila destinada para la memoria dinamica del ARM se llena. Esto se deja ver en la figura 6.3

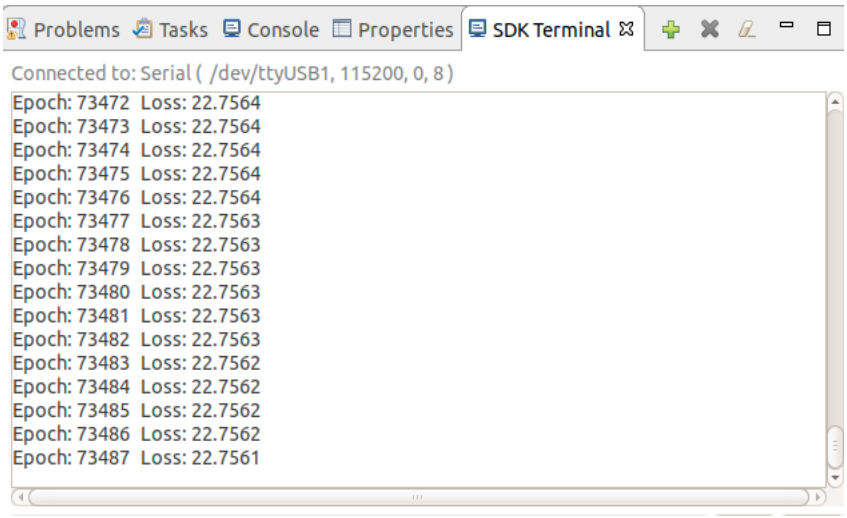


Figura 6.3 Tres neuronas en la capa oculta.

Como se ve en la figura 3.4 con 4 neuronas logra resolver, pero tarda demasiado (36725 épocas).

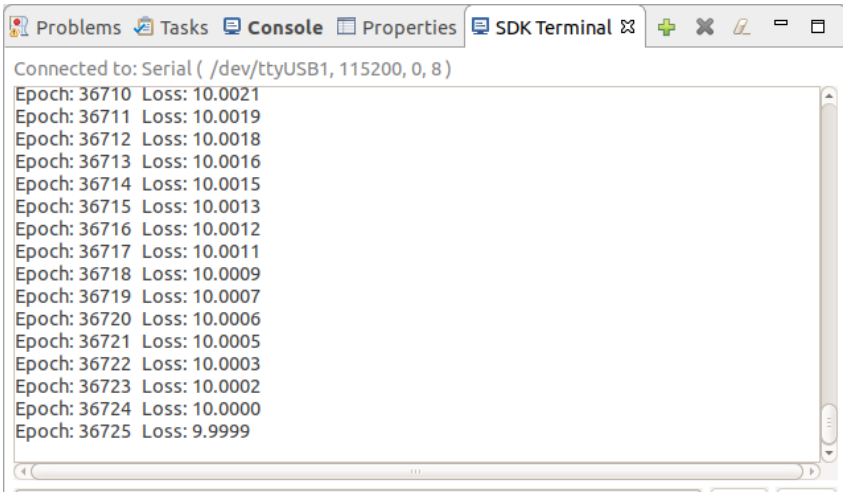


Figura 6.4 Cuatro neuronas en la capa oculta.

En la figura 6.5 se puede observa el funcionamiento óptimo con 8 neuronas.

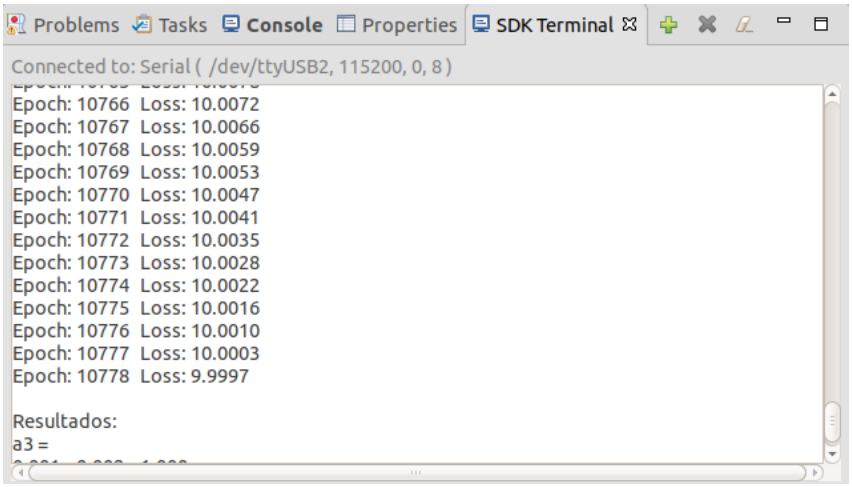


Figura 6.5. Ocho neuronas en la capa oculta.

De esta manera se analiza el desempeño de la red neuronal en la tarjeta de desarrollo Zybo Z-10, pues por una parte analizamos el PL en cuanto a los recursos que le toma realizar la propagación hacia adelante, por otro lado, la etapa de entrenamiento nos permitió analizar el desempeño de PS tomando en cuenta la cantidad de iteraciones que realiza para entrenar la red.

Conclusiones

Se diseñó e implementó una red neuronal multicapa en una tarjeta de desarrollo la cual cuenta con un sistema de un sólo chip (SoC) aprovechando sus recursos tanto en software para realizar el entrenamiento de la red, como en hardware para realizar la ejecución de la red.

El diseño puede utilizarse como base para la implementación de sistemas embebidos que necesiten utilizar una red neuronal multicapa, aplicaciones como la clasificación de datos, reconocimiento de patrones y la predicción pueden ser implementadas de manera óptima para cada caso, debido a que es totalmente ajustable.

Otra de las principales ventajas del sistema es que debido a que la etapa de entrenamiento de la red neuronal se realiza en la parte de lógica programable, una vez que la red ha sido entrenada, se puede disponer casi en su totalidad de los recursos del procesador para ejecutar otras funciones, mientras la FPGA realiza la ejecución de la red, lo cual se traduce como una reducción en el tiempo de ejecución.

Una de las desventajas que implica el hecho de que los parámetros de la red sean ajustables, es que dichos cambios deben realizarse tanto en el procesador como en la lógica programable, lo que puede provocar errores si no se tiene cuidado en el momento de establecerlos, ya que los parámetros deben de ser los mismos en ambos casos.

Alguno de los elementos que se podrían añadir al sistema con el fin de tener un mayor alcance de aplicaciones que se le puedan dar, sería más funciones de activación. El inconveniente es que cuando se busca programar dichas funciones en hardware, al igual que en cualquier otro modulo que implique la representación de punto flotante, la dificultad es mayor.

El usuario puede re-dimensionar el número de neuronas, capas ocultas y cambiar parámetros para diferentes aplicaciones, desafortunadamente se observó en la implementación que estos ajustes no pueden realizarse libremente, ya que la parte de lógica ocupa demasiados recursos, que incrementarán a medida que los elementos de la red lo hagan, lo que puede ocasionar que los recursos necesarios para implementar el sistema sean mucho mayores que los que la tarjeta ofrece, lo cual quiere decir que la implementación esta limitada por la capacidad lógica de la FPGA. Esto también indica que la forma en que se diseñó la red, repartiendo la etapa de

entrenamiento en el procesador y la etapa de ejecución en lógica programable fue la más óptima para el desarrollo de redes neuronales en dispositivos APSoC, ya que implementar ambas etapas en el en lógica programable no sería viable, debido a que la demanda de recursos sería muy alta.

Los resultados de la implementación de la red nos muestran que el número de capas repercute de manera directa en los recursos destinados para la lógica, mientras que el número de neuronas impacta en el rendimiento del procesador.

Gran parte de los recursos para bloques lógicos son ocupados como memorias que almacenan datos que la red utiliza. Una mejora que se propone para desarrollar en el futuro es disponer de los bloques de memoria que la tarjeta de desarrollo posee para almacenar todos los datos que la etapa de prueba utilice y con esto liberar recursos de la FPGA que se podrán aprovechar mejor para la lógica de más elementos de la red neuronal en la etapa de ejecución.

La implementación de este diseño nos permite observar los recursos en hardware que una red neuronal necesita, asimismo el diseño en hardware brinda un panorama mucho más amplio en cuanto a aplicaciones se refiere, el uso de una FPGA es ideal para aplicaciones de bioinformática, de esta manera se propone como aplicación a futuro caracterizar señales o imágenes médicas mediante redes neuronales.

Apéndice

Código para programar PS

Código correspondiente al programa principal (main).

```
#include <stdio.h>
#include "xil_printf.h"
#include "xil_io.h"
#include "dynamic_matrix.h"
#include "FPGA.h"
#include "signals.h"
#include "Matrix.h"

#define data_reg 0x43C00000
#define trns data_reg // Registro 0 para escribir cuando se ha terminado de enviar datos
#define ctrl data_reg + 4  // Registro 1 para leer cuando se han recibido señales de control
#define N 20
#define LED_DELAY 10000000            /* Software delay length */
volatile int Delay;
// Parameters
volatile int inputCount = 4;          // Input neurons
volatile int hiddenCount1 = 4;        // 1st hidden layer neurons
volatile int outputCount = 3;         // Output neurons
volatile int exampleCount = 120;      // Example combinations
volatile float learningRate = 0.3;    // Learning rate
volatile float lossPorcent = 10;      // Loss
float w1[N][N], w2[N][N];             // Weigths
float R[N][N];                        // Feedforward res
float mx[N][N] = {{1, 0, 0, 0, 0}};   // Input data
int main() {
        unsigned int control = 0;
        unsigned int trans = 0;
        // Variables para el registro de control fpga: ctrl
        int reset, mode, w_done, ma1_done, out_send, w1_done, w2_done;
        int train_done = 0;
        Random_M(w1, inputCount + 1, hiddenCount1, 100);
        Random_M(w2, hiddenCount1 + 1, outputCount, 100);
        while(1) {
                // Read ctrl output from register 1
                control = Xil_In32(ctrl);
                trans = Xil_In32(trns);
                reset = consultar(0,control);           // Reset (bit 0)
                mode = consultar(1,control);            // Mode (bit 1)
                w_done = consultar(2,control);          // w_done (bit 2)
                ma1_done = consultar(3,control);        // ma1_done (bit 3)
```

```
out_send = consultar(5,control);        // Output (bit 5)
w1_done = consultar(6,control);         // W1 done (bit 6)
w2_done = consultar(7,control);         // W2 done (bit 7)

if (reset == 1) {                                       /*RESET*/
        xil_printf("Reset...\n");
        Random_M(w1, inputCount + 1, hiddenCount1, 5);
        Random_M(w2, hiddenCount1 + 1, outputCount, 5);
        train_done = 0;
        Xil_Out32(trns, 0b00000000000000000000000000000000);
}
else if (mode == 0 && train_done == 0) {                /*TRAIN*/
        printf("Entrenando...\n");
        Xil_Out32(trns, 0b00000000000000000000000000000001);
        train(w1,w2);
        train_done = 1;
        Xil_Out32(trns, trans | (train_done << 3));// train_done = 1 (bit 3)
        printf("¡Entrenamiento terminado!\n");
}
else if (mode == 1 && w_done == 0) {                    /* LOAD WEIGHTS*/
        if (w1_done == 0) {
                // Write w1 data to register 2 to register...
                write_matrix(inputCount + 1, hiddenCount1, w1);
                // w1_ready = '1' (bit 4)
                trans = 0b00000000000000000000000000010000;
                Xil_Out32(trns, trans);
        }
        else if (w2_done == 0) {
                // Write w2 data to register 2 to register...
                write_matrix(hiddenCount1 + 1, outputCount, w2);
                // w2_ready = '1' (bit 5)
                trans = 0b00000000000000000000000000100000;
                Xil_Out32(trns, trans);
                printf("\n¡Pesos enviados!\n\nw1 = \n");
                Show_FloatM(w1, inputCount + 1, hiddenCount1);
                printf("w2 = \n");
                Show_FloatM(w2, hiddenCount1 + 1, outputCount);
        }
}
else if (mode == 1 && w_done == 1) {        // FEED FORWARD
        if (ma1_done == 0) {
                printf("Ingrese nueva entrada: \n");
                scan_iris(1, inputCount + 1, mx);
                printf("\nVector de entrada normalizado: \n");
                Show_FloatM(mx, 1, inputCount +1);
```

```
                        // Write mx data to register 2 to register...
                        write_matrix(1, inputCount + 1, mx);
                        // mx_ready = '1' (bit 2)
                        trans = 0b00000000000000000000000000000100;
                        Xil_Out32(trns, trans);
                    }
                    else if (out_send == 1) {
                        printf("Salida: \n");
                        read_matrix(1, outputCount, R);
                        Show_FloatM(R, 1, outputCount);
                        printf("Especie: Iris");
                        clasifica_iris(R);
                        printf("\n");
                        // in_clr = '1' (bit 1)
                        Xil_Out32(trns, trans | (1 << 1));
                    }
                }
            }
            return 0;
}
```

Código para programar PL

Código del modulo “layer_out.vhd”

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

entity layer_out is
Generic (
 width: integer;     -- 32 bits
 mrow: integer;
 mcol: integer;      -- + 1 biass
 wrow: integer;      -- # of row to w1
 wcol: integer       -- # of col to w1
);

Port (
 clk         : in STD_LOGIC;
 clr         : in STD_LOGIC;
 mdata       : in STD_LOGIC_VECTOR (width - 1 downto 0);
 maddr_row : out integer;
 maddr_col : out integer;
 mready    : in STD_LOGIC;
 wdata     : in STD_LOGIC_VECTOR (width - 1 downto 0);
 waddr_row : out integer;
 waddr_col : out integer;
 wready    : in STD_LOGIC;
 outaddr_row :  in integer;
 outaddr_col :  in integer;
 done        :  out STD_LOGIC;
 output      :  out STD_LOGIC_VECTOR (width - 1 downto 0)
);
end layer_out;

architecture Behavioral of layer_out is
-----------------------------------------------------------------------------------------
--SEÑALES PARA MULT_MAT
-- señales para la multiplicacion de matrices
 signal mataddr_row, mcolwrowaddr : integer := 0;
 signal multaddr_row, multaddr_col : integer := 0;
 signal mat_ready, mult_done : std_logic;
 signal multout     : std_logic_vector(width - 1 downto 0);
-----------------------------------------------------------------------------------------
--SEÑALES PARA MATRIX_ADDCOL
```

```
 -- señales para la agregar columna a matriz
 signal addcolout : std_logic_vector(width - 1 downto 0);
 signal addaddr_row, addaddr_col : integer := 0;
 signal add_done : std_logic;
-----------------------------------------------------------------------------------------
--SEÑALES PARA MATRIX_SIGMOID
 -- señales para la agregar columna a matriz
 signal sig_done : std_logic;
-----------------------------------------------------------------------------------------
begin
---------------------------------------------------------------------------------------- Instanciacion de mult_mat en
donde se realiza la multiplicación entre dos matrices
mat_ready <= mready and wready;
maddr_row <= mataddr_row;
maddr_col <= mcolwrowaddr;
waddr_row <= mcolwrowaddr;

multiplicacion_mw : entity work.mult_mat
 Generic Map (
   -- Valores constantes de las matrices m1 y m2 para realizar la multiplicacion
   width    => width,   -- Tamaño de cada elemento de la matriz
   m1row    => mrow,   -- Filas de min1
   m1col    => mcol,   -- Columnas de min1
   m2row    => wrow,   -- Filas de w1
   m2col    => wcol    -- Columnas de w1
 )
 Port Map (
 -- ENTRADAS
   cclk         => clk,           -- Reloj del sistema
   clr          => clr,           -- Reset
   ready        => mat_ready,     -- Entrada para habilitar la multiplicación
   m1input      => mdata,       -- Entrada para el vector del arreglo prom de la matriz de entrada m1
   m2input      => wdata,        -- Entrada para el vector del arreglo prom de la matriz de entrada m2
   mraddr_row   => multaddr_row,    -- Entrada para seleccionar direccion en filas de la matriz resultante en el vector de
salida mrout
   mraddr_col   => multaddr_col,    -- Entrada para seleccionar direccion en columnas de la matriz resultante en el vector de
salida mrout
 -- SALIDAS
   -- Salidas de las pociciones que se solicitan en las matrices m1 y m2 para realizar la multiplicacion
   kout         => mcolwrowaddr,    --= maddr_col Contador para
   jout         => waddr_col,    -- Contador para w1_col
   iout         => mataddr_row,   -- Contador para min_row
   done         => mult_done,     -- Salida para indicar cuando se ha terminado la multiplicacion de matrices
   mrout        => multout             -- Vector de salida en donde se muestra la posicion de la matriz resultante
seleccionada por mr_row y mr_col
 );
```

```
-----------------------------------------------------------------------------------------------
-- Instanciacion de matrix_sigmoid para obtener función sigmoide de los elementos de una matriz
done <= sig_done;
funcion_sigmoide : entity work.matrix_sigmoid
 Generic Map (
   -- Valores constantes de la matriz m1 que será mostrada en los leds
   width    => width,      -- Tamaño de cada elemento de la matriz
   mrow     => mrow,       -- Filas de m
   mcol     => wcol        -- Columnas de m
 )
 Port Map (
 -- ENTRADAS
   cclk         => clk,          -- Reloj del sistema
   clr          => clr,          -- Reset
   ready        => mult_done,     -- Entrada para habilitar el modulo matrix_addcol
   minput       => multout,       -- Entrada para el vector del arreglo prom de la matriz de entrada
   mraddr_row   => outaddr_row,--sigaddr_row,  -- Entrada para seleccionar direccion en filas de la matriz resultante en el
vector de salida mrout
   mraddr_col   => outaddr_col,--sigaddr_col,  -- Entrada para seleccionar direccion en columnas de la matriz resultante en el
vector de salida mrout
 -- SALIDAS
   mout_row     => multaddr_row,   -- Salida para las direcciones en fila que se necesitan leer del arreglo prom de la matriz
de entrada
   mout_col     => multaddr_col,   -- Salida para las direcciones en columna que se necesitan leer del arreglo prom de la
matriz de entrada
   done         => sig_done,      -- Salida para indicar cuando se ha terminado la la función del modulo
   mrout        => output--sigmoidout      -- Vector de salida en donde se muestra la posicion de la matriz resultante
seleccionada por mr_row y mr_col
 );
-----------------------------------------------------------------------------------------------
end Behavioral;
```

Código del modulo “matrix_sigmod.vhd”

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


entity matrix_sigmoid is
 Generic (
   width : integer;           -- 32 bits
   mrow  : integer;           -- # of row to m1
   mcol  : integer            -- # of col to m1
 );
 Port(
   cclk       : in STD_LOGIC;
   clr        : in STD_LOGIC;
   ready      : in STD_LOGIC;
```

```
    mraddr_row : in integer;
    mraddr_col : in integer;
    minput     : in STD_LOGIC_VECTOR(width - 1 downto 0);
    mrout      : out STD_LOGIC_VECTOR(width - 1 downto 0);
    done       : out STD_LOGIC;
    mout_row   : out integer;
    mout_col   : out integer
  );
end matrix_sigmoid;

architecture Behavioral of matrix_sigmoid is

COMPONENT floating_point_exp
  PORT (
    aclk : IN STD_LOGIC;
    aresetn : IN STD_LOGIC;
    s_axis_a_tvalid : IN STD_LOGIC;
    s_axis_a_tready : OUT STD_LOGIC;
    s_axis_a_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    m_axis_result_tvalid : OUT STD_LOGIC;
    m_axis_result_tready : IN STD_LOGIC;
    m_axis_result_tdata : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
  );
END COMPONENT;

COMPONENT floating_point_add
  PORT (
    aclk : IN STD_LOGIC;
    aresetn : IN STD_LOGIC;
    s_axis_a_tvalid : IN STD_LOGIC;
    s_axis_a_tready : OUT STD_LOGIC;
    s_axis_a_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    s_axis_b_tvalid : IN STD_LOGIC;
    s_axis_b_tready : OUT STD_LOGIC;
    s_axis_b_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    m_axis_result_tvalid : OUT STD_LOGIC;
    m_axis_result_tready : IN STD_LOGIC;
    m_axis_result_tdata : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
  );
END COMPONENT;

COMPONENT floating_point_div
  PORT (
    aclk : IN STD_LOGIC;
    aresetn : IN STD_LOGIC;
```

```
    s_axis_a_tvalid : IN STD_LOGIC;
    s_axis_a_tready : OUT STD_LOGIC;
    s_axis_a_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    s_axis_b_tvalid : IN STD_LOGIC;
    s_axis_b_tready : OUT STD_LOGIC;
    s_axis_b_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    m_axis_result_tvalid : OUT STD_LOGIC;
    m_axis_result_tready : IN STD_LOGIC;
    m_axis_result_tdata : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
  );
END COMPONENT;
-------------------------------------------------------------------------------------------------------------------------
-- SEÑALES PARA MATRIZ SIGMOIDE
 -- Matriz de resultado
 type matrizr is array (0 to mrow - 1) of std_logic_vector(0 to (mcol * width) - 1);
 signal mr  : matrizr;
 signal full : std_logic;
-- State machine declaration
 type ADD_SM is (ADD, DELAY, INC);
 signal FSIGMOID             : ADD_SM;
 attribute INIT             : string;
 attribute INIT of FSIGMOID : signal is "ADD";
-- Señales para ir guardando resultados parciales
signal x, exponencial, denominador, result : STD_LOGIC_VECTOR(31 downto 0);
-- Señales para habilitar instancias de las operaciones de punto flotante
signal expdone, susdone, sigdone, valid : STD_LOGIC := '0';
signal fp_reset : STD_LOGIC := '1';

begin
-----------------------------------------------------------------------------------------
-- SECCION PARA RECORRER ARREGLO PROM DE MATRIZ DE ENTRADA
-- Proceso que llena la matriz 'A' con  los datos del arreglo
MatrixA: process (cclk, clr, minput)
variable i, j,count : integer:= 0;
 begin
    if clr = '1' then   -- Reset
        full <= '0';
        done <= '0';
        i := 0;
        j := 0;
        mout_row <= 0;
        mout_col <= 0;
        valid <= '0';
        fp_reset <= '1';
        mr <= (others => "0");
```

```
    FSIGMOID <= ADD;
elsif (rising_edge(cclk)) then
  case (FSIGMOID) is

    -- Guarda valor de resultado de la función sigmoide en la posicion del arreglo rom de la matriz
    when ADD =>
        if (full = '0' and ready = '1') then
            mr(i)(j * width to (j * width) + width - 1) <= result(width - 1 downto 0);
            valid <= '1';   -- Se indica que se puede iniciar la operacion
            if (sigdone = '1') then -- Una vez que se ha realizado la operacion, se cambia al siguiente estado
                fp_reset <= '0';    -- Inicia reset para IPs
                FSIGMOID <= DELAY;
            end if;
        else
            FSIGMOID <= ADD;
        end if;

    -- Delay para reset de los IP floating point
    when DELAY =>
        if (count < 2) then
            count := count + 1;
            FSIGMOID <= DELAY;
        else
            count := 0;
            FSIGMOID <= INC;
        end if;

    -- Incremento de los contadores de posición para recorrer el arreglo de la matriz de entrada
    when INC =>
        if (j < mcol-1) then
            j := j + 1;
            FSIGMOID <= ADD;
        elsif (i < mrow-1) then
            i := i + 1;
            j := 0;
            FSIGMOID <= ADD;
        else
            full <= '1';    -- La matriz ha llenada totalmente
        end if;
        valid <= '0';
        fp_reset <= '1';    -- Termina reset de IPs
        -- Señales de salida
        done <= full;
        mout_row <= i;
        mout_col <= j;

    when others => FSIGMOID <= ADD;      --Just in case.
```

```
      end case;
    end if;
 end process;
-- Termina sección de llenado de las matriz
-------------------------------------------------------------------------------------------
-- SECCION PARA REALIZAR FUNCION SIGMOIDE

-- Multiplica por -1 el valor de la entrada
x(31) <= minput(31) xor '1';            -- -X
x(30 downto 0) <= minput(30 downto 0);

-- exponencial = e^-x
exponecial : floating_point_exp
    PORT MAP (
      aclk                  => cclk,
      aresetn               => fp_reset,
      s_axis_a_tvalid       => valid,
      s_axis_a_tready       => open,
      s_axis_a_tdata        => x(31 downto 0),
      m_axis_result_tvalid  => expdone,
      m_axis_result_tready  => '1',
      m_axis_result_tdata   => exponencial
    );

-- denominador = 1 - exponencial
suma : floating_point_add
    PORT MAP (
      aclk                  => cclk,
      aresetn               => fp_reset,
      s_axis_a_tvalid       => '1',
      s_axis_a_tready       => open,
      s_axis_a_tdata        => "00111111100000000000000000000000",    -- 1
      s_axis_b_tvalid       => expdone,
      s_axis_b_tready       => open,
      s_axis_b_tdata        => exponencial,
      m_axis_result_tvalid  => susdone,
      m_axis_result_tready  => '1',
      m_axis_result_tdata   => denominador
    );

-- output = 1 / denominador
division : floating_point_div
    PORT MAP (
      aclk                  => cclk,
      aresetn               => fp_reset,
      s_axis_a_tvalid       => '1',
      s_axis_a_tready       => open,
```

```
      s_axis_a_tdata        => "00111111100000000000000000000000",      -- 1
      s_axis_b_tvalid       => susdone,
      s_axis_b_tready       => open,
      s_axis_b_tdata        => denominador,
      m_axis_result_tvalid  => sigdone,
      m_axis_result_tready  => '1',
      m_axis_result_tdata   => result
    );
-- Termina sección funcion sigmoide

----------------------------------------------------------------------------------
-- SELECCION DE DIRECCION PARA LA SALIDA
-- Proceso que coloca en el vector de salida la dirección de la matriz resultante que se solicita
SelectorM1 : process (mraddr_row, mraddr_col, full)
 begin
     if (full = '1') then
         mrout(31 downto 0) <= mr(mraddr_row)(mraddr_col * width to (mraddr_col * width) + width - 1); -- Muestra la dirección
a la que se apunta
     end if;
 end process;
 -- Termina sección de dirección de la salida
end Behavioral;
```

Código del modulo “matrix_addcol.vhd”

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

entity matrix_addcol is
 Generic (
   width: integer;           -- 32 bits
   mrow: integer;           -- # of row to m1
   mcol: integer           -- # of col to m1
 );
 Port(
   cclk       : in STD_LOGIC;
   clr        : in STD_LOGIC;
   ready      : in STD_LOGIC;
   mraddr_row : in integer;
   mraddr_col : in integer;
   minput     : in STD_LOGIC_VECTOR(width - 1 downto 0);
   mrout      : out STD_LOGIC_VECTOR(width - 1 downto 0);
   done       : out STD_LOGIC;
   mout_row   : out integer;
   mout_col   : out integer
 );
```

```
end matrix_addcol;

architecture Behavioral of matrix_addcol is
----------------------------------------------------------------------------------------------
-- SEÑALES PARA AGREGAR COLUMNA

 -- Matriz de resultado
 type matrizr is array (0 to mrow - 1) of std_logic_vector(0 to ((mcol + 1) * width) - 1);
 signal mr  : matrizr;
 signal full : std_logic;
-- State machine declaration
 type ADD_SM is (ADD, INC);
 signal ADD_COL             : ADD_SM;
 attribute INIT             : string;
 attribute INIT of ADD_COL : signal is "ADD";

begin
----------------------------------------------------------------------------------------------
-- SECCION PARA AGREGAR COLUMNA
-- Proceso que llena la matriz 'A' con  los datos del arreglo 'data_vector'
MatrixA: process (cclk, clr, minput)
variable i, j : integer:= 0;
 begin
    if clr = '1' then
        full <= '0';
        done <= '0';
        i := 0;
        j := 0;
        mout_row <= 0;
        mout_col <= 0;
        mr <= (others => "0");
        ADD_COL <= ADD;
    elsif (rising_edge(cclk)) then
      case (ADD_COL) is

        when ADD =>
            if (full = '0' and ready = '1') then
                if (j = 0) then
                    mr(i)(0 to width - 1) <= "00111111100000000000000000000000";
                end if;
                mr(i)((j + 1) * width to ((j + 1) * width) + width - 1) <= minput(width - 1 downto 0);
                ADD_COL <= INC;
            else
                ADD_COL <= ADD;
            end if;

        when INC =>
```

```
            if (j < mcol-1) then
                j := j + 1;
                ADD_COL <= ADD;
            elsif (i < mrow-1) then
                i := i + 1;
                j := 0;
                ADD_COL <= ADD;
            else
                full <= '1';
            end if;
            done <= full;
            mout_row <= i;
            mout_col <= j;

        when others => ADD_COL <= ADD;      --Just in case.
      end case;
    end if;
 end process;
-- Termina sección de llenado de las matriz
----------------------------------------------------------------------------------------
-- SELECCION DE DIRECCION PARA LA SALIDA
-- Proceso que coloca en el vector de salida la dirección de la matriz resultante que se solicita
SelectorM1 : process (mraddr_row, mraddr_col, full)
 begin
     if (full = '1') then
         mrout(31 downto 0) <= mr(mraddr_row)(mraddr_col * width to (mraddr_col * width) + width - 1); -- Muestra la dirección
a la que se apunta
     end if;
 end process;
-- Termina sección de dirección de la salida
end Behavioral;
```

Código del modulo “mult_mat.vhd” para la multiplicación de matrices en FPGA.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity mult_mat is
    Generic (
    width: integer;           -- 32 bits
    m1row: integer;
    m1col: integer;
    m2row: integer;
    m2col: integer
);
```

```
Port (
    cclk    : in STD_LOGIC;
    clr     : in STD_LOGIC;
    ready   : in STD_LOGIC;     -- Entrada para habilitar la multiplicacion
    m1input   : in STD_LOGIC_VECTOR(31 downto 0);  -- Entrada de datos para la primera matriz
    m2input   : in STD_LOGIC_VECTOR(31 downto 0);  -- Entrada de datos para la segunda matriz
    mraddr_row : in integer;       -- Entrada para dirección en filas de la matriz resultante
    mraddr_col : in integer;       -- Entrada para dirección en columnas de la matriz resultante
    done    : out STD_LOGIC;    -- Salida para indicador de operacion realizada
    mrout : out STD_LOGIC_VECTOR(31 downto 0); -- Vector de salida para elemento de la matriz resultante seleccionado
    kout    : out integer;      -- Salidas para los contadores
    jout    : out integer;      --
    iout    : out integer       --
);
end mult_mat;

architecture Behavioral of mult_mat is
----------------------------------------------------------------------------------
-- SEÑALES PARA MULTIPLICACION DE MATRICES

 -- Matriz de resultado
 type matrizr is array (0 to m1row - 1) of std_logic_vector(0 to (m2col * width) - 1);
 signal mr  : matrizr;

 -- Señales para la multiplicacion en la maquina de estados
 signal mult_done     : std_logic;
 signal mult_overflow : std_logic;
 signal mult_res      : std_logic_vector (width - 1 downto 0);
 constant BIAS : unsigned(8 downto 0) := to_unsigned(127, 9);  --exponent bias of 127
 signal full_mantissa : std_logic_vector(47 downto 0) := (others => '0');
 signal full_exp      : std_logic_vector(8 downto 0) := (others => '0');  --extra msb for unsigned exponent addition
 signal R_sign        : std_logic;
 signal Aop, Bop, Cop : std_logic_vector(31 downto 0) := (others => '0');  --latched operands
 alias A_sign         : std_logic is Aop(31);           --operand segments
 alias A_exp          : std_logic_vector(7 downto 0) is Aop(30 downto 23);
 alias A_man          : std_logic_vector(22 downto 0) is Aop(22 downto 0);
 alias B_sign : std_logic is Bop(31);
 alias B_exp  : std_logic_vector(7 downto 0) is Bop(30 downto 23);
 alias B_man  : std_logic_vector(22 downto 0) is Bop(22 downto 0);

 -- Señales para las suma en la maquina de estados
 signal A_mantissa, B_mantissa : std_logic_vector (24 downto 0);
 signal A_expo, B_expo         : std_logic_vector (8 downto 0);
 signal A_sgn, B_sgn           : std_logic;
 --output signals
```

```
 signal sum                   : std_logic_vector (31 downto 0);
 --signal F_exp: std_logic_vector (7 downto 0);
 signal mantissa_sum          : std_logic_vector (24 downto 0);

 -- State machine declaration
 type MULT_SM is (WAITM, MAN_EXP, CHECK, NORMALIZE, PAUSE, WAITC, ALIGN, ADDC, NORMC, PAUSEC, INC);
 signal MULTIPLY            : MULT_SM;
 attribute INIT             : string;
 attribute INIT of MULTIPLY : signal is "WAITM";

begin
-------------------------------------------------------------------------------------------
-- MULTIPLICACION DE LAS MATRICES'A' Y 'B'
--Proceso para realizar la multiplicación de matrices, realizando la operacion A * B + C
process (MULTIPLY, cclk, ready, clr) is
variable diff : signed(8 downto 0);
variable i, j, k    : integer := 0;
begin
  if (clr = '1') then
    full_mantissa <= (others => '0');
    full_exp      <= (others => '0');
    sum(31 downto 0)      <= (others => '0');
    mult_res(31 downto 0) <= (others => '0');
    Cop(31 downto 0)      <= (others => '0');
    k := 0;
    j := 0;
    i := 0;
    kout <= 0;
    jout <= 0;
    iout <= 0;
    mult_done     <= '0';
    MULTIPLY      <= WAITM;
  elsif (rising_edge(cclk)) then
    case (MULTIPLY) is
      -- En este estado comienza a realizar la operacion correspondiente A * B
      when WAITM =>
        mult_overflow <= '0';
        mult_done     <= '0';
        if (ready = '1') then
          Aop      <= m1input;
          Bop      <= m2input;
          MULTIPLY <= MAN_EXP;
        else
          MULTIPLY <= WAITM;
```

```
  end if;

when MAN_EXP =>  --compute a sign, exponent and matissa for product
  R_sign        <= A_sign xor B_sign;
  full_mantissa <= std_logic_vector(unsigned('1' & A_man) * unsigned('1' & B_man));
  --full_exp <= std_logic_vector( (unsigned(A_exp)-BIAS) + (unsigned(B_exp)-BIAS)+ BIAS );
  full_exp      <= std_logic_vector((unsigned(A_exp)- BIAS) + unsigned(B_exp));
                                --MULTIPLY <= CHECK;
  MULTIPLY      <= NORMALIZE;
when CHECK =>                   --Check for exponent overflow
  if (unsigned(full_exp) > 255) then
    mult_overflow <= '1';
    mult_res   <= (31 => R_sign, others => '1');
    mult_done  <= '1';
    MULTIPLY <= PAUSE;
  else
    MULTIPLY <= NORMALIZE;
  end if;

when NORMALIZE =>
  if full_mantissa(47) = '1' then
    full_mantissa <= '0' & full_mantissa(47 downto 1);
    full_exp      <= std_logic_vector(unsigned(full_exp) + 1);
  else
    mult_res   <= R_sign & full_exp(7 downto 0) & full_mantissa(45 downto 23);
    MULTIPLY <= WAITC;
  end if;

when PAUSE =>                   -- wait for acknowledgement
  if (ready = '0') then
    mult_done     <= '0';
    MULTIPLY <= WAITM;
  end if;

-- A partir de este estado comienza a realizarse la suma de mult_res(A * B) + Cop
when WAITC =>
  A_sgn       <= mult_res(31);
  B_sgn       <= Cop(31);
  A_expo      <= '0' & mult_res(30 downto 23);
  B_expo      <= '0' & Cop(30 downto 23);
  A_mantissa  <= "01" & mult_res(22 downto 0);
  B_mantissa  <= "01" & Cop(22 downto 0);
  MULTIPLY    <= ALIGN;

when ALIGN =>  --exponent alignment. Always makes A_exp be final exponent
```

```
    --Below method is like a barrel shift, but is big space hog------
    --note that if either num is greater by 2**24, we skip the addition.
  if unsigned(A_expo) = unsigned(B_expo) then
    MULTIPLY <= ADDC;
  elsif unsigned(A_expo) > unsigned(B_expo) then
    diff := signed(A_expo) - signed(B_expo);  --B needs downshifting
    if diff > 23 then
      mantissa_sum <= A_mantissa;  --B insignificant relative to A
      sum(31)      <= A_sgn;
      MULTIPLY      <= PAUSEC;   --go latch A as output
    else       --downshift B to equilabrate B_exp to A_exp
      B_mantissa(24-TO_INTEGER(diff) downto 0)  <= B_mantissa(24 downto TO_INTEGER(diff));
      B_mantissa(24 downto 25-TO_INTEGER(diff)) <= (others => '0');
      MULTIPLY                                  <= ADDC;
    end if;
  else                          --A_exp < B_exp. A needs downshifting
    diff := signed(B_expo) - signed(A_expo);
    if diff > 23 then
      mantissa_sum <= B_mantissa;  --A insignificant relative to B
      sum(31)      <= B_sgn;
      A_expo        <= B_expo;  --this is just a hack since A_exp is used for final result
      MULTIPLY      <= PAUSEC;   --go latch B as output
    else       --downshift A to equilabrate A_exp to B_exp
      A_expo                                    <= B_expo;
      A_mantissa(24-TO_INTEGER(diff) downto 0)  <= A_mantissa(24 downto TO_INTEGER(diff));
      A_mantissa(24 downto 25-TO_INTEGER(diff)) <= (others => '0');
      MULTIPLY                                  <= ADDC;
    end if;
  end if;

when ADDC =>                    --Mantissa addition
    MULTIPLY <= NORMC;
    if (A_sgn xor B_sgn) = '0' then  --signs are the same. Just add 'em
      mantissa_sum <= std_logic_vector((unsigned(A_mantissa) + unsigned(B_mantissa)));
      sum(31)      <= A_sgn;      --both nums have same sign
    --otherwise subtract smaller from larger and use sign of larger
    elsif unsigned(A_mantissa) >= unsigned(B_mantissa) then
      mantissa_sum <= std_logic_vector((unsigned(A_mantissa) - unsigned(B_mantissa)));
      sum(31)      <= A_sgn;
    else
      mantissa_sum <= std_logic_vector((unsigned(B_mantissa) - unsigned(A_mantissa)));
      sum(31)      <= B_sgn;
    end if;
```

```
when NORMC =>  --post normalization. A_exp is the exponent of the unormalized sum
    if unsigned(mantissa_sum) = TO_UNSIGNED(0, 25) then
      mantissa_sum <= (others => '0');  --break out if a mantissa of 0
      A_expo        <= (others => '0');
      MULTIPLY      <= PAUSEC;     --
    elsif(mantissa_sum(24) = '1') then  --if sum overflowed we downshift and are done.
      mantissa_sum <= '0' & mantissa_sum(24 downto 1);  --shift the 1 down
      A_expo        <= std_logic_vector((unsigned(A_expo)+ 1));
      MULTIPLY     <= PAUSEC;
    elsif(mantissa_sum(23) = '0') then  --in this case we need to upshift
      --Below takes big resources to determine the normalization upshift,
      --  but does it one step.
      for i in 22 downto 1 loop   --find position of the leading 1
        if mantissa_sum(i) = '1' then
          mantissa_sum(24 downto 23-i) <= mantissa_sum(i+1 downto 0);
          mantissa_sum(22-i downto 0)  <= (others => '0');  --size of shift= 23-i
          A_expo <= std_logic_vector(unsigned(A_expo)- 23 + i);
          exit;
        end if;
      end loop;
      MULTIPLY <= PAUSEC;          --go latch output, wait for acknowledge
    else
      MULTIPLY <= PAUSEC;  --leading 1 already there. Latch output, wait for acknowledge
    end if;

when PAUSEC =>
    sum(22 downto 0)  <= mantissa_sum(22 downto 0);
    sum(30 downto 23) <= A_expo(7 downto 0);
    mr(i)(j * width to (j * width) + width - 1) <= Cop(31 downto 0);
    MULTIPLY <= INC;

when INC =>
    if (k < m1col) then
      Cop(31 downto 0) <= sum(31 downto 0);
      k := k + 1;
      MULTIPLY <= WAITM;
    elsif (j < m2col ) then
      Cop(31 downto 0) <= (others => '0');
      j := j + 1;
      k := 0;
      MULTIPLY <= WAITM;
    elsif (i < m1row ) then
      Cop(31 downto 0) <= (others => '0');
```

```
            i := i + 1;
            j := 0;
            k := 0;
            MULTIPLY <= WAITM;
          else
            mult_done <= '1';
          end if;
          if (ready = '0') then            --pause till request ends
            MULTIPLY <= WAITC;
          end if;
          kout <= k;
          jout <= j;
          iout <= i;
      when others => MULTIPLY <= WAITM;      --Just in case.
      end case;
  end if;
end process;

-------------------------------------------------------------------------------------------
-- SELECCION DE DIRECCION PARA LA SALIDA

done <= mult_done;  -- Salida que indica cuando la multiplicación de matrices ha sido completada

SelectorM1 : process (mult_done, mraddr_row, mraddr_col)
begin
    if (mult_done = '1') then
        mrout(31 downto 0) <= mr(mraddr_row)(mraddr_col * width to (mraddr_col * width) + width - 1); -- Muestra la dirección
a la que se apunta
    end if;
end process;

-------------------------------------------------------------------------------------------
end Behavioral;
```

Bibliografía

[1] Lara Rosano F. "Fundamentos de Redes Neuronales Artificiales". Instrumentación y Desarrollo, Vol 3, No 2, (1992), pp 82-91

[2] S. Russell y P. Norvig, *Inteligencia artificial: Un enfoque moderno*, Edición: 2. Madrid: ALHAMBRA, 2004.

[3] T. M. Mitchell, *Machine Learning*, 1 edition. New York: McGraw-Hill Education, 1997.

[4] S. S. Haykin y S. S. Haykin, *Neural networks and learning machines*, 3rd ed. New York: Prentice Hall, 2009.

[5] J. G. Cruz, “NEURONAS Y NEUROTRANSMISORES”, p. 15.[En línea]. Disponible en:http://depa.fquim.unam.mx/amyd/archivero/NEURONASYNEUROTRANSMISORES_1118.pdf [Consultado: 13-mar.2019]

[6] “Fundamentos de las redes neuronales”. [En línea]. Disponible en: https://thales.cica.es/rd/Recursos/rd98/TecInfo/07/capitulo2.html. [Consultado: 18-mar-2019].

[7].Carolus, “Hipótesis: El legado de Hebb para la psicología científica”, *Hipótesis*, 03-feb-2008. .

[8] E. M. Petriu, “Neural Networks: Modeling Applications”, p. 83.

[9] C. Pérez, “Sistemas Embebidos (ES)”, p. 19.

[10] “Microcontroladores ARM Advanced RISC Machine”. .

[11].S. D. Brown y Z. G. Vranesic, *Fundamentals of digital logic with VHDL design*, 3rd ed. New York, NY: McGraw-Hill, 2009.

[12] L. H. Crockett, R. a Elliot, y M. a Enderwitz, *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Glasgow: Strathclyde Academic Media, 2014.

[13] “Zybo Z7: Zynq-7000 ARM/FPGA SoC Development Board”, *Digilent*. [En línea]. Disponible en: https://store.digilentinc.com/zybo-z7-zynq-7000-arm-fpga-soc-development-board/. [Consultado: 04-abr-2019].

[14] B. J. McCaffrey y 07/22/2013, “Neural Network Data Normalization and Encoding -”, *Visual Studio Magazine*. [En línea]. Disponible en: https://visualstudiomagazine.com/articles/2013/07/01/neural-network-data-normalization-and-encoding.aspx. [Consultado: 11-abr-2019].

[15]L. H. Crockett, R. A. Elliot, y M. A. Enderwitz, *The Zynq Book Tutorials for Zybo and ZedBoard*. Glasgow: Strathclyde Academic Media, 2015.

[16] D. M. Muñoz, “ARITMÉTICA EN PUNTO FLOTANTE”, p. 18.

[17] xesscorp, *VHDL for basic floating-point operations. Contribute to xesscorp/Floating_Point_Library-JHU development by creating an account on GitHub*. 2019.