



GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

Acelerador de red neuronal (NN) en FPGA
para reconocimiento de dígitos (OCR), diseño
de sistema de captura y estudio comparativo

Autor: Patricia Martín González

Director: Fermín Zabalegui

Madrid, Julio 2026

Declaración de originalidad

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título **Acelerador de red neuronal (NN) en FPGA para reconocimiento de dígitos (OCR), diseño de sistema de captura y estudio comparativo** en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2025/2026 es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Uso de Inteligencia Artificial¹

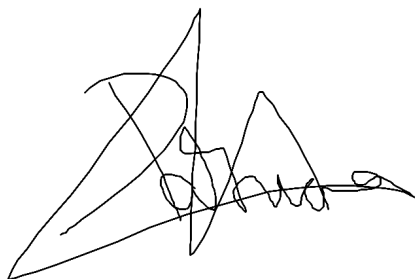
Declaro bajo mi responsabilidad que (indicar la opción correcta):

- ☐ No he utilizado Inteligencia Artificial en la elaboración del presente documento.
- ☒ He utilizado Inteligencia Artificial en la elaboración del presente documento y/o del Anexo B siempre en las condiciones permitidas por la Universidad Pontificia Comillas, es decir, aplicando el Nivel 2 de la [Escala de Evaluación de Perkins et al. \(2024\)](#): *“La IA puede utilizarse para actividades previas a la tarea, como la lluvia de ideas, la descripción y la investigación inicial. Este nivel se centra en el uso de la IA para la planificación, las síntesis y la generación de ideas, pero las evaluaciones deben hacer hincapié en la capacidad de desarrollar y refinar estas ideas de forma independiente”*. En concreto, la Inteligencia Artificial ha sido empleada para:

¹Esta declaración se refiere al uso de la Inteligencia Artificial generativa para realizar los documentos del Proyecto (Anexo B y Memoria). No aplica a Proyectos donde, por su naturaleza, deban emplear inteligencia artificial como parte de los mismos (aplicación de técnicas de aprendizaje automático, redes neuronales, análisis de datos...).

La inteligencia artificial se ha empleado para:

- Búsqueda de información
- Reescritura de párrafos con el fin de facilitar la lectura
- Generación de ideas iniciales



Fdo: Patricia Martín González

Fecha: 10/07/2026

Autorización para la entrega del Proyecto

El Director del Proyecto	El co-Director del Proyecto (si aplica)
ZABALEGUI SANZ 2026.07.13 FERMIN - 50202045Z 11:53:33 +02'00' (firma aquí)	(firma aquí)
Fdo: Fermín Zabalegui	Fdo:
Fecha:	



GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

Acelerador de red neuronal (NN) en FPGA
para reconocimiento de dígitos (OCR), diseño
de sistema de captura y estudio comparativo

Autor: Patricia Martín González

Director: Fermín Zabalegui

Madrid, Julio 2026

Agradecimientos

A mis padres por darme esta oportunidad, a mi familia por el apoyo incondicional y a mis amigos por estar ahí siempre.

ACELERADOR DE RED NEURONAL (NN) EN FPGA PARA RECONOCIMIENTO DE DÍGITOS (OCR), DISEÑO DE SISTEMA DE CAPTURA Y ESTUDIO COMPARATIVO

Autor: Patricia Martín González

Director: Fermín Zabalegui

Entidad Colaboradora: ICAI – Universidad Pontificia Comillas.

Resumen

Este Trabajo de Fin de Grado desarrolla un acelerador hardware para una red neuronal multicapa de reconocimiento de dígitos manuscritos, implementado en VHDL sobre una FPGA Basys 3. La solución se evalúa comparando su rendimiento, exactitud, latencia y consumo energético frente a una implementación software y en hardwares de uso general.

Palabras clave: FPGA; Redes neuronales; OCR; MNIST; VHDL; Cuantificación; Sistemas embebidos.

Resumen ejecutivo

1 Introducción

La inteligencia artificial, y en particular las redes neuronales, ha experimentado un gran crecimiento en los últimos años, impulsando aplicaciones con elevadas exigencias computacionales. En este contexto, las FPGA se presentan como una solución eficiente para la inferencia en sistemas embebidos, gracias a su capacidad de paralelismo, baja latencia y reducido consumo energético [1].

2 Objetivos

Entre los objetivos principales de este Trabajo de Fin de Grado se encuentran la implementación del diseño en la FPGA y la reducción del consumo energético en comparación con los procesadores

de uso general.

3 Descripción del modelo/sistema/herramienta

La solución desarrollada abarca todo el proceso de implementación de una red neuronal sobre una FPGA. En primer lugar, se entrena una red neuronal multicapa con el conjunto de datos MNIST y se selecciona una arquitectura que ofrece un equilibrio entre precisión y complejidad hardware. Posteriormente, los parámetros del modelo se cuantifican a representación entera de 8 bits y se implementan en VHDL sobre una FPGA Basys 3.

La arquitectura hardware, de carácter modular, se basa en una única unidad MAC reutilizable controlada por una máquina de estados, utilizando memorias BRAM para almacenar los parámetros y aritmética en punto fijo para optimizar el uso de recursos. Como complemento, se desarrolla un sistema de captura de imágenes mediante una cámara OV7670, orientado a futuras aplicaciones de reconocimiento en tiempo real.

La Figura 1 muestra el flujo general seguido, desde el entrenamiento del modelo hasta su implementación final sobre la FPGA.

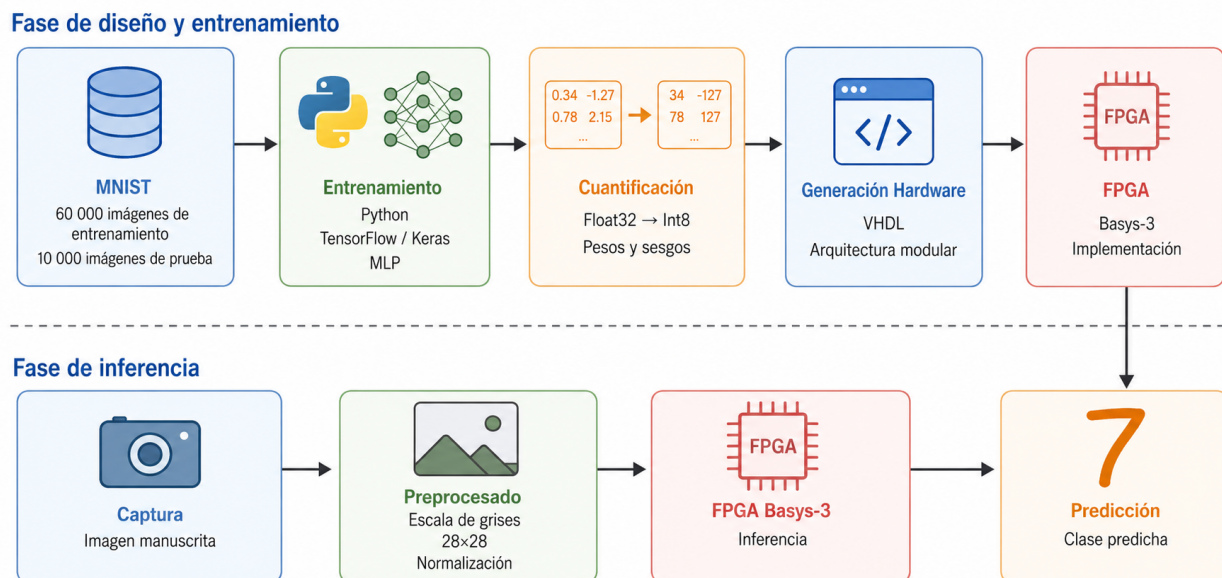


Figura 1: Flujo general seguido para el desarrollo del acelerador de red neuronal, desde el entrenamiento del modelo hasta su implementación en la FPGA.

4 Resultados

A continuación se muestran los recursos empleados de la FPGA, destacando el 100 % de uso de la memoria BRAM.

Recurso	Utilización	Disponible	Uso (%)
LUT	14716	20800	70,75 %
LUTRAM	456	9600	4,75 %
Flip-Flops	2961	41600	7,12 %
BRAM	50	50	100,00 %
IO	14	106	13,21 %

Tabla 1: Uso de recursos en FPGA

Se implementó el modelo en una Raspberry Pi y se realizó un análisis comparativo con respecto a la implementación en la FPGA. Los valores obtenidos se muestran a continuación:

Parámetro	Raspberry Pi	FPGA
Exactitud	97,6 %	97,6 %
Tiempo por inferencia	1–2 ms	17,48 ms
Inferencias por segundo	500–1000	57
Consumo	6,7 W	2,4 W
Arquitectura	CPU ARM	Hardware dedicado
Latencia	Variable	Determinista

Tabla 2: Comparación entre la implementación software en Raspberry Pi y la implementación hardware en FPGA.

Se observa que la FPGA es más lenta pero consume menos potencia que la Raspberry Pi.

5 Conclusiones

En este trabajo se ha desarrollado e implementado una red neuronal para el reconocimiento de dígitos manuscritos sobre una FPGA Basys 3. Los resultados obtenidos demuestran la viabilidad de esta solución, destacando su bajo consumo energético y su idoneidad para aplicaciones embebidas. Además, el diseño constituye una base para futuros desarrollos de sistemas OCR más complejos.

6 Referencias

- [1] A. Shawahna, S. M. Sait y A. El-Maleh, «FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review», *IEEE Access*, vol. 7, págs. 7823-7859, 2019. DOI: 10.1109/ACCESS.2018.2890150

FPGA-BASED NEURAL NETWORK (NN) ACCELERATOR FOR DIGIT RECOGNITION (OCR): SYSTEM CAPTURE DESIGN AND COMPARATIVE STUDY

Author: Patricia Martín González

Director: Fermín Zabalegui

Colaborating Entity: ICAI – Universidad Pontificia Comillas.

Abstract

This Bachelor's Thesis presents the development of a hardware accelerator for a multilayer neural network for handwritten digit recognition, implemented in VHDL on a Basys 3 FPGA. The proposed solution is evaluated by comparing its performance, accuracy, latency, and energy consumption against a software implementation and general-purpose hardware platforms.

Keywords: FPGA; Neural Networks; OCR; MNIST; VHDL; Quantization; Embedded Systems.

Executive Summary

1 Introduction

Artificial intelligence, and particularly neural networks, has experienced significant growth in recent years, driving applications with increasingly demanding computational requirements. In this context, FPGAs provide an efficient solution for inference in embedded systems due to their inherent parallelism, low latency, and reduced power consumption [1].

2 Objectives

The main objectives of this Bachelor's Thesis are to implement a neural network accelerator on an FPGA and to reduce energy consumption compared with general-purpose processors.

3 Description of the Model/System/Tool

The proposed solution covers the complete implementation flow of a neural network on an FPGA. First, a multilayer perceptron (MLP) is trained using the MNIST dataset, and an architecture that balances accuracy and hardware complexity is selected. The model parameters are then quantized to 8-bit signed integer representation and implemented in VHDL on a Basys 3 FPGA.

The hardware architecture follows a modular design based on a single reusable MAC unit controlled by a finite-state machine. BRAM memories are used to store the network parameters, while fixed-point arithmetic minimizes hardware resource utilization. In addition, an image acquisition system based on an OV7670 camera is developed, providing a foundation for future real-time digit recognition applications.

Figure 1 illustrates the overall development flow, from model training and parameter quantization to the final FPGA implementation.

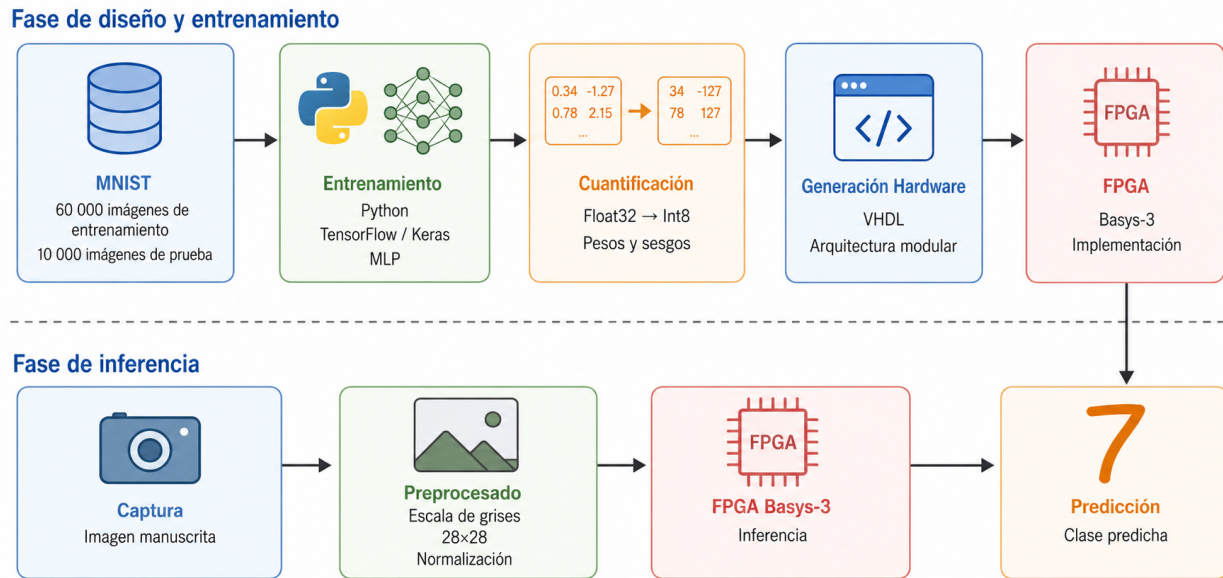


Figure 1: Overall development flow of the neural network accelerator, from model training to its implementation on the FPGA.

4 Resultados

The FPGA resource utilization is presented below, highlighting the complete utilization of the available BRAM memory.

Resource	Utilization	Available	Usage (%)
LUT	14716	20800	70,75%
LUTRAM	456	9600	4,75%
Flip-Flops	2961	41600	7,12%
BRAM	50	50	100,00%
IO	14	106	13,21%

Table 1: FPGA resource utilization.

The proposed model was also implemented on a Raspberry Pi, and a comparative analysis was carried out with respect to the FPGA implementation. The obtained results are summarized in the following table.

Parameter	Raspberry Pi	FPGA
Accuracy	97.6%	97.6%
Inference time	1–2 ms	17.48 ms
Inferences per second	500–1000	57
Power consumption	6,7 W	2,4 W
Architecture	ARM CPU	Dedicated hardware
Latency	Variable	Deterministic

Table 2: Comparison between the software implementation on the Raspberry Pi and the hardware implementation on the FPGA.

The results show that the FPGA implementation is slower than the Raspberry Pi implementation but requires significantly lower power consumption.

5 Conclusions

In this work, a neural network for handwritten digit recognition was successfully designed and implemented on a Basys 3 FPGA. The obtained results demonstrate the feasibility of this approach, highlighting its low power consumption and its suitability for embedded applications. Furthermore, the proposed design provides a solid foundation for the future development of more advanced OCR systems.

6 References

- [1] A. Shawahna, S. M. Sait, and A. El-Maleh, «Fpga-based accelerators of deep learning networks for learning and classification: A review», *IEEE Access*, vol. 7, pp. 7823–7859, 2019. DOI: 10.1109/ACCESS.2018.2890150

Índice

1	Introducción	3
1.1	Motivación	4
1.2	Objetivos	5
1.3	Estructura del trabajo	6
2	Estado del arte	7
2.1	Fundamentos de Redes Neuronales Artificiales (ANN)	7
2.1.1	La neurona artificial y el perceptrón	7
2.1.2	Funciones de activación	8
2.1.3	Arquitectura de redes multicapa (Feedforward)	9
2.1.4	El proceso de entrenamiento y optimización	9
2.2	Reconocimiento Óptico de Caracteres (OCR) y el Dataset MNIST	12
2.3	Paradigmas de Cómputo para Inteligencia Artificial: CPUs, GPUs y FPGAs	12
2.3.1	Arquitecturas basadas en Software (CPU y GPU)	12
2.3.2	Computación en el Borde (Edge Computing)	13
2.3.3	FPGAs como aceleradores de hardware	13
2.4	Optimización de Redes Neuronales para Hardware	13
2.4.1	Cuantificación y aritmética de punto fijo	13
2.4.2	El compromiso entre precisión y recursos	14
2.5	Trabajos relacionados con el proyecto y soluciones afines	14
2.5.1	Implementaciones clásicas de MNIST en FPGA	14
2.5.2	Soluciones comerciales vs. Soluciones a medida	16
3	Metodología	18
3.1	Elección de la red neuronal y justificación	18
3.2	Base de datos y preprocesado	21
3.3	Arquitectura de la red neuronal y entrenamiento	23
3.4	Arquitectura general del sistema	24
3.5	Arquitectura hardware	25
3.5.1	Modelo temporal de la arquitectura	29
3.6	Funcionamiento del sistema	30
3.7	Máquina de estados de la Unidad de Control	31

3.7.1	Control de iteraciones y punteros	32
3.7.2	Transición entre estados	32
3.7.3	Generación de direcciones	33
3.7.4	Señales de salida	33
3.8	Diseño y Arquitectura Real de la Unidad MAC Unificada y Reutilizable . . .	33
3.8.1	Camino de Datos (Datapath) y Prevención de Desbordamiento	35
3.8.2	Flujo de Operación por Capas (Multiplexación Temporal)	35
3.8.3	Bloque de Post-procesamiento: Bias, Saturación y ReLU Hardware .	36
3.9	Implementación en FPGA	37
3.9.1	Placa de desarrollo Basys3	37
3.9.2	Entorno de desarrollo Vivado	38
3.9.3	Proceso de síntesis e implementación	39
3.9.4	Archivo de restricciones (XDC)	40
3.9.5	Programación de la FPGA	41
3.9.6	Verificación funcional	41
3.10	Caracterización energética	42
3.11	Sistema de captura	44
4	Resultados	45
4.1	Análisis del modelo en Python	45
4.1.1	Visualización de activaciones neuronales	48
4.2	Comprobación del modelo en VHDL y comparación	49
4.3	Impacto de la cuantificación	51
4.4	Simulaciones y análisis temporal	51
4.4.1	Reevaluación temporal de la arquitectura implementada	53
4.5	Consumo energético	56
4.6	Estudio comparativo entre FPGA y Raspberry Pi	58
5	Conclusiones y trabajo futuro	60
5.1	Conclusiones generales	60
5.2	Cumplimiento de los objetivos	60
5.2.1	Diseño y entrenamiento del modelo de reconocimiento	60
5.2.2	Optimización aritmética para su implementación en hardware	60
5.2.3	Desarrollo del acelerador hardware en VHDL	61

5.2.4	Diseño del sistema de captura y preprocesado	61
5.2.5	Validación e implementación física del sistema	61
5.2.6	Análisis comparativo del rendimiento	61
5.3	Análisis crítico de los resultados	62
5.4	Limitaciones del trabajo	63
5.4.1	Limitaciones hardware	63
5.4.2	Limitaciones temporales	63
5.4.3	Limitaciones de la comparación experimental	63
5.5	Trabajo futuro	64
5.5.1	Implementación de un sistema OCR completo	64
5.5.2	Incremento de la capacidad de la red neuronal	64
5.5.3	Mayor paralelismo hardware	64
5.5.4	Ampliación del estudio comparativo	65
Bibliografía		66
A	Alineación con los ODS	71
B	Esquema de bloques del sistema	73
C	Código VHDL Unidad de Control	74
D	Código VHDL bloque MAC	78

Índice de figuras

1	Esquema de conexiones entre capas en una red neuronal feedforward multicapa.	9
2	Diagrama de flujo de los pasos seguidos en el desarrollo del proyecto.	18
3	Precisión del modelo en función del número de neuronas en la primera capa oculta.	19
4	Costes de entrenamiento y prueba en función del número de neuronas en la primera capa oculta.	20
5	Imágenes de la base de datos MNIST	22
6	Diagrama de flujo del proceso de clasificación en dos filas.	25
7	Diagrama de bloques del diseño en VHDL	26
8	Funcionamiento de los displays	28
9	Funcionamiento de las señales de activación de los displays	28
10	Diagrama de flujo del sistema en dos filas.	30
11	Diagrama de estados de la Unidad de Control	31
12	Diagrama de estados del bloque MAC	34
13	Placa de desarrollo Basys3 utilizada para la implementación del sistema. . .	38
14	Circuito de medición de potencia para la FPGA	42
15	Camara empleada como sistema de captura	44
16	Matriz de confusión del modelo en Python.	45
17	Errores del modelo en Python.	46
18	Errores del modelo en Python.	46
19	Título general para el grupo de 10 imágenes.	47
20	Visualización de activaciones de distintas neuronas ante la clase “7”	48
21	Grupo de las primeras 5 imágenes analizadas.	50
22	Simulación del diseño en VHDL	53
23	Esquema general del diseño en VHDL	73

Índice de tablas

1	Comparación de trabajos relacionados con implementaciones de redes neuronales en FPGA.	16
2	Arquitectura de la red neuronal implementada.	21
3	Hiperparámetros empleados durante el entrenamiento.	24
4	Estimación teórica del tiempo de inferencia según el grado de paralelismo . .	30
5	Ejemplo de asignación de señales mediante el archivo XDC.	41
6	Comparación entre el número real de la subfigura y la predicción del modelo.	47
7	Comparativa de predicciones entre el modelo en Python y la implementación en VHDL.	50
8	Impacto de la cuantificación en la precisión del modelo	51
9	Estimaciones temporales para una frecuencia de reloj de 25 MHz.	55
10	Desglose del consumo de potencia estimado por Vivado (solo chip de la FPGA).	56
11	Resultados de la caracterización energética del sistema completo.	57
12	Comparación de las distintas implementaciones software sobre Raspberry Pi.	58
13	Comparación entre la implementación software en Raspberry Pi y la implementación hardware en FPGA.	59

Capítulo 1: Introducción

La evolución de la tecnología ha impulsado la necesidad de digitalizar y automatizar procesos tradicionalmente manuales para mejorar la eficiencia y la productividad. En este contexto, las técnicas de aprendizaje automático y, en particular, las redes neuronales, han adquirido un papel destacado en el procesamiento de información y en la resolución de tareas complejas en múltiples dominios [1], [2].

Las redes neuronales son modelos computacionales inspirados en principios biológicos que permiten aprender a partir de los datos y abordar tareas como el reconocimiento de patrones, la clasificación o la predicción. Entre las arquitecturas más relevantes se encuentran las redes neuronales artificiales clásicas (ANN), las redes neuronales convolucionales (CNN) y las redes neuronales recurrentes (RNN), cada una orientada a distintos tipos de problemas. En particular, las CNN han mostrado un rendimiento especialmente alto en tareas de reconocimiento de imágenes y de dígitos manuscritos [1], [3], [4].

El reconocimiento de dígitos manuscritos es una tarea de gran interés en la digitalización documental y está estrechamente relacionado con la tecnología OCR (Optical Character Recognition), cuyo objetivo es transformar caracteres escritos en formatos legibles por máquina. Sus aplicaciones incluyen la automatización de procesos administrativos, la digitalización de documentos y distintos sistemas de lectura automática [5].

Tradicionalmente, la ejecución de redes neuronales se ha apoyado en plataformas de alto rendimiento como las GPU, debido a su elevada capacidad de cómputo paralelo. No obstante, en escenarios con restricciones de latencia, consumo energético y recursos disponibles, como los sistemas embebidos y los dispositivos móviles, las FPGA se presentan como una alternativa muy atractiva. Su capacidad de reconfiguración y de explotación del paralelismo permite diseñar aceleradores específicos con mejoras potenciales en rendimiento y eficiencia energética [6], [7], [8].

En los últimos años, además, se ha producido una transición desde infraestructuras centralizadas hacia paradigmas de Edge Computing, en los que el procesamiento se desplaza hacia nodos cercanos a la fuente de datos. Este cambio responde a necesidades de menor latencia, menor coste de transmisión y operación en dispositivos con recursos limitados. En este contexto, las FPGA destacan como una solución adecuada para ejecutar inferencia de redes neuronales de forma eficiente en el borde [9], [10], [11].

La elección de una FPGA, y en particular de una plataforma como BASYS-3, frente a alternativas basadas únicamente en procesadores de propósito general o microcontroladores, puede justificarse por su personalización hardware, su paralelismo intrínseco y su potencial para reducir la latencia en aplicaciones de tiempo real. Estas características hacen de las FPGA una opción adecuada para el desarrollo de aceleradores específicos de inferencia de redes neuronales [7], [8].

De este modo, este proyecto se centra en el desarrollo de un acelerador de red neuronal sobre FPGA para el reconocimiento de dígitos, abarcando tanto el diseño del sistema de captura en Python como la implementación del algoritmo de reconocimiento en hardware. De este modo, se plantea un estudio comparativo con otras arquitecturas de computación, evaluando aspectos como el rendimiento, el consumo energético y la capacidad de operación en tiempo real.

1.1. Motivación

En la actualidad, los algoritmos de aprendizaje profundo se implementan principalmente sobre arquitecturas tradicionales como CPU y GPU. Aunque estas plataformas ofrecen un elevado rendimiento computacional, también presentan limitaciones relevantes en determinados entornos, especialmente en términos de consumo energético, latencia y dependencia de infraestructuras externas. Además, cuando la inferencia depende del envío de datos a la nube, surgen problemas asociados al ancho de banda, la privacidad y la seguridad de la información transmitida.

Como alternativa a estas limitaciones, las FPGA permiten desplazar la inferencia al borde (Edge Computing), posibilitando el procesamiento local y en tiempo real. Su flexibilidad arquitectónica permite diseñar hardware específico y altamente paralelo, optimizando el rendimiento por vatio y reduciendo el consumo energético frente a soluciones convencionales.

Esta optimización puede reforzarse mediante técnicas de cuantificación, en particular a través del uso de representaciones en punto fijo de 8 bits. Dicho enfoque reduce el uso de recursos lógicos y memoria en FPGA, manteniendo al mismo tiempo un nivel de precisión adecuado para tareas de inferencia en redes neuronales.

Por tanto, abordar este problema contribuye al desarrollo de sistemas autónomos, seguros y de bajo consumo, especialmente valiosos en aplicaciones industriales y embebidas donde la

rapidez de respuesta y la eficiencia energética son factores críticos.

1.2. Objetivos

Con el fin de dar respuesta al problema planteado, este proyecto persigue los siguientes objetivos:

- **Diseño y entrenamiento del modelo de reconocimiento:** Desarrollar, entrenar y validar una arquitectura de red neuronal empleando Python y la base de datos MNIST, con el objetivo de alcanzar una elevada precisión en la clasificación de dígitos manuscritos.
- **Optimización aritmética para su implementación en hardware:** Estudiar y aplicar técnicas de cuantificación de parámetros, transformando pesos y sesgos desde coma flotante a una representación en punto fijo de 8 bits con signo, con el fin de optimizar el uso de recursos en FPGA.
- **Desarrollo del acelerador hardware en VHDL:** Diseñar e implementar la arquitectura modular de la red neuronal mediante VHDL, incluyendo capas de entrada, capas ocultas con activación ReLU y capa de salida, para explotar el paralelismo y optimizar las operaciones aritméticas.
- **Diseño del sistema de captura y preprocesado:** Desarrollar el sistema de captura, preprocesado y acondicionamiento de imágenes de entrada, garantizando su transferencia correcta hacia los bloques de procesamiento implementados en la FPGA.
- **Validación e implementación física del sistema:** Verificar el funcionamiento de la arquitectura mediante simulación y testbenches en Vivado, para posteriormente realizar la síntesis e implementación sobre la FPGA y comprobar su viabilidad en un entorno real.
- **Análisis comparativo del rendimiento:** Evaluar el acelerador en términos de latencia, consumo energético y ocupación de recursos, comparando los resultados con una implementación software basada en CPU o GPU y analizando el impacto de la cuantificación sobre la precisión final del modelo.

1.3. Estructura del trabajo

Este documento recoge toda la información relativa al desarrollo del proyecto y se encuentra dividido en los siguientes capítulos, cada uno de ellos orientado a describir los distintos aspectos abordados durante su realización:

- **Capítulo 1. Introducción:** Presenta el problema que se pretende resolver, justifica la elección del proyecto, expone los objetivos planteados y describe su alineación con los Objetivos de Desarrollo Sostenible (ODS).
- **Capítulo 2. Estado del arte:** Contextualiza la situación actual del problema y analiza los trabajos e investigaciones existentes relacionados con el reconocimiento de dígitos y la implementación de redes neuronales en FPGA. Además, se presentan los conceptos fundamentales, las técnicas empleadas y las limitaciones de las soluciones actuales.
- **Capítulo 3. Metodología:** Describe la metodología seguida para alcanzar los objetivos propuestos, detallando el diseño del sistema, el entrenamiento del modelo de red neuronal, el proceso de cuantificación y la implementación del acelerador hardware.
- **Capítulo 4. Resultados:** Presenta y analiza los resultados obtenidos durante la validación del sistema, así como el estudio comparativo realizado entre la implementación en FPGA y otras arquitecturas.
- **Capítulo 5. Conclusiones y trabajo futuro:** Recoge las principales conclusiones extraídas del proyecto, evalúa el grado de cumplimiento de los objetivos establecidos y propone posibles líneas de trabajo futuro orientadas a mejorar, ampliar o adaptar la solución desarrollada a nuevos escenarios de aplicación.

Capítulo 2: Estado del arte

En esta sección se presentan los fundamentos teóricos de las redes neuronales artificiales y los principios de entrenamiento que sustentan los algoritmos de aprendizaje profundo, con el fin de facilitar la comprensión del desarrollo del proyecto. También se contextualiza la implementación de redes neuronales sobre hardware reconfigurable, ya que las FPGA se han consolidado como una plataforma adecuada para acelerar inferencia y, de forma más limitada, entrenamiento de modelos neuronales, especialmente en entornos con restricciones de latencia, consumo y recursos [6], [12]. Por último, se consideran trabajos relacionados con la aceleración y despliegue de redes neuronales en FPGA, donde aparecen de forma recurrente estrategias como paralelismo, reutilización de datos, poda y cuantización [6], [12].

2.1. Fundamentos de Redes Neuronales Artificiales (ANN)

2.1.1. La neurona artificial y el perceptrón

El modelo matemático empleado por las redes neuronales artificiales combina entradas, pesos y un sesgo mediante una transformación afín antes de aplicar una no linealidad. En su forma más simple, cada neurona calcula una suma ponderada de las entradas procedentes de la capa anterior, lo que constituye la base del perceptrón y de las redes multicapa posteriores [13], [14]:

$$z_j = \sum_{i=1}^n w_{ji} \cdot x_i + b_j \quad (2.1)$$

Donde los componentes lógicos y matemáticos se definen a continuación:

- x_i : representa el valor de la i -ésima entrada proveniente de la capa anterior; en problemas de visión, estas entradas pueden corresponder a características o píxeles normalizados de la imagen.
 - w_{ji} : es el peso sináptico que conecta la entrada i con la neurona j , y determina la contribución relativa de dicha entrada a la salida de la neurona.
 - b_j : es el sesgo asociado a la neurona j , que desplaza la transformación afín y amplía la capacidad representacional del modelo.
-

- z_j : representa el potencial neto o suma ponderada antes de aplicar la función de activación.

2.1.2. Funciones de activación

Tras obtener la suma ponderada en cada neurona, es necesario aplicar una función de activación para introducir no linealidad; sin ella, incluso una red con múltiples capas se reduciría a una transformación lineal [15]. La literatura coincide en que la función de activación influye de manera directa en la capacidad de representación, la estabilidad del entrenamiento y el rendimiento final del modelo [15], [16].

En este modelo se han optado por las siguientes funciones de activación:

- **ReLU:**

$$\sigma_{\text{ReLU}}(z) = \max(0, z) = \begin{cases} z & \text{si } z \geq 0 \\ 0 & \text{si } z < 0 \end{cases} \quad (2.2)$$

ReLU es la activación más utilizada en redes profundas por su simplicidad computacional y su buen comportamiento práctico [16], [17]. Su definición a trozos hace que la salida sea nula para entradas negativas y lineal para entradas positivas, lo que simplifica también su derivada en implementaciones hardware [17], [18]. No obstante, la literatura también señala limitaciones como el problema de dying ReLU, que ha motivado numerosas variantes posteriores [15], [18].

- **Softmax:**

$$\sigma_{\text{Softmax}}(z_j) = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}} \quad (2.3)$$

donde K representa el número total de clases del problema. En clasificación multiclase, Softmax se emplea habitualmente como función de salida para transformar puntuaciones en valores comparables entre clases mutuamente excluyentes [17]. En este contexto, con $K = 10$, la neurona con mayor activación determina la predicción final del sistema OCR.

2.1.3. Arquitectura de redes multicapa (Feedforward)

Las redes multicapa de tipo feedforward organizan el procesamiento en capas sucesivas, donde la salida de una capa se convierte en la entrada de la siguiente sin realimentación [13], [19]. Este esquema constituye la base del perceptrón multicapa (MLP), que incluye una capa de entrada, una o varias capas ocultas y una capa de salida [19], [20].

La conexión entre las capas sigue, por tanto, un flujo secuencial de propagación hacia delante: cada neurona combina las activaciones de la capa previa, aplica su función de activación y transmite su salida a la siguiente capa [13], [19]. Este patrón también se conserva en aceleradores hardware, donde la inferencia se implementa como una sucesión de operaciones capa a capa adaptadas a la arquitectura disponible [6], [12]. La Figura 1 muestra el esquema de conexiones entre las distintas capas de una red neuronal feedforward.

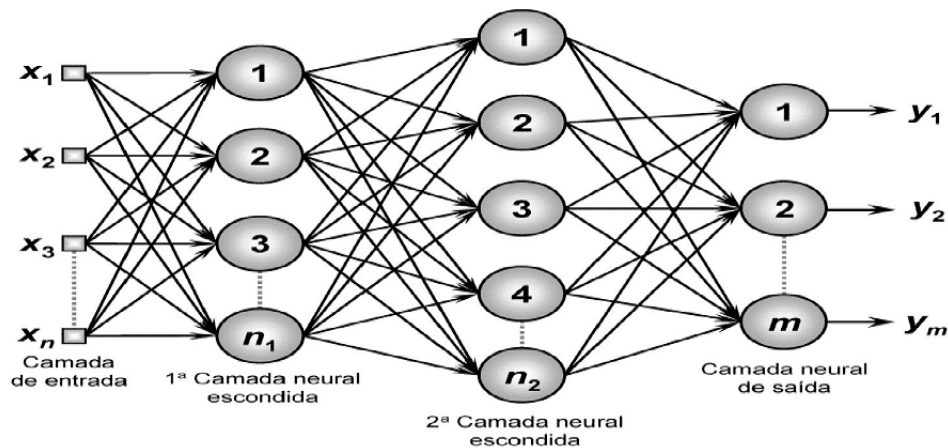


Figura 1: Esquema de conexiones entre capas en una red neuronal feedforward multicapa.

2.1.4. El proceso de entrenamiento y optimización

El entrenamiento de una red neuronal consiste en ajustar los pesos y sesgos para minimizar una función de error definida sobre las predicciones del modelo [14], [21]. En redes feedforward, este proceso se apoya de forma clásica en dos elementos: el descenso por gradiente para actualizar parámetros y la retropropagación para calcular de manera eficiente los gradientes capa a capa [13], [22]. Aunque existen variantes y alternativas, la retropropagación convencional sigue siendo el enfoque más utilizado por su estructura matemática clara y su

facilidad de implementación [22], [23].

El Gradiente Descendente (Gradient Descent)

El objetivo de este algoritmo es minimizar la función de error E , que depende del conjunto de pesos y sesgos del modelo. La actualización iterativa de cada peso se realiza en la dirección opuesta al gradiente:

$$w_{ij}^{(\text{nuevo})} = w_{ij}^{(\text{actual})} - \alpha \frac{\partial E}{\partial w_{ij}} \quad (2.4)$$

De forma análoga, para un sesgo b_i :

$$b_i^{(\text{nuevo})} = b_i^{(\text{actual})} - \alpha \frac{\partial E}{\partial b_i} \quad (2.5)$$

- $\frac{\partial E}{\partial w_{ij}}$: representa el gradiente de la función de error respecto a un peso concreto y determina el sentido e intensidad de la actualización.
- α : es la tasa de aprendizaje, un hiperparámetro que controla el tamaño del paso; su valor condiciona la velocidad de convergencia y la estabilidad del entrenamiento [14].

El descenso por gradiente clásico presenta limitaciones prácticas, entre ellas convergencia lenta y sensibilidad a la elección de hiperparámetros [21], [23]. Por ello se han desarrollado extensiones ampliamente utilizadas, como Momentum, RMSProp o Adam, que adaptan o estabilizan el proceso de optimización [22], [24], [25]. Además, también se han propuesto enfoques alternativos basados en metaheurísticas para mitigar problemas como el atrapamiento en mínimos locales en MLP [20].

La Retropropagación (Backpropagation) y la Regla de la Cadena

Para aplicar el descenso por gradiente es necesario calcular las derivadas parciales del error respecto a todos los parámetros de la red. La retropropagación proporciona un procedimiento eficiente para ello, propagando hacia atrás las señales de error desde la salida hasta las capas anteriores mediante la regla de la cadena [13], [23]. Este esquema de paso hacia delante y paso hacia atrás constituye el núcleo del entrenamiento supervisado en perceptrones multicapa [14], [22].

Asumiendo una función de error como el Error Cuadrático Medio (MSE) para una muestra:

$$E = \frac{1}{2} \sum_k (y_k - a_k)^2 \quad (2.6)$$

donde y_k es el valor objetivo y a_k es la activación de la neurona de salida correspondiente.

Para una neurona cualquiera, su activación se define como $a_j = \sigma(z_j)$, con $z_j = \sum_i w_{ji}a_i + b_j$, es decir, como una transformación afín seguida de una no linealidad. Aplicando la regla de la cadena, la derivada del error respecto a un peso puede descomponerse como:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial a_j} \cdot \frac{\partial a_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ji}} \quad (2.7)$$

1. $\frac{\partial E}{\partial a_j}$ mide la sensibilidad del error frente a cambios en la activación de la neurona; en la capa de salida depende directamente de la diferencia entre predicción y objetivo.
2. $\frac{\partial a_j}{\partial z_j}$ corresponde a la derivada de la función de activación, y por ello el comportamiento de dicha activación afecta de forma directa al flujo del gradiente durante el entrenamiento [15].
3. $\frac{\partial z_j}{\partial w_{ji}}$ expresa la influencia del peso sobre la suma ponderada, y coincide con la activación a_i de la neurona precedente.

Agrupando los dos primeros términos en una señal de error local δ_j , se obtiene:

$$\delta_j = \frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial a_j} \cdot \sigma'(z_j) \quad (2.8)$$

y, en consecuencia, la expresión compacta del gradiente respecto a cada peso queda:

$$\frac{\partial E}{\partial w_{ji}} = \delta_j \cdot a_i \quad (2.9)$$

Esta formulación explica por qué la retropropagación resulta escalable: convierte el cálculo de un gran número de derivadas en una propagación estructurada de señales de error a través de la red. Aun así, su implementación eficiente en hardware ha sido tradicionalmente difícil por los requisitos de almacenamiento intermedio y transferencia de datos, aunque trabajos recientes ya han demostrado variantes de retropropagación in situ que actualizan capas secuencialmente en hardware y reproducen un comportamiento de aprendizaje comparable al software. En FPGA, la mayor parte de la literatura sigue concentrándose en inferencia acelerada, con mejoras apoyadas en paralelismo, reducción de precisión, reutilización de datos, poda, cuantización y mapeo eficiente a operaciones de multiplicación-acumulación.

2.2. Reconocimiento Óptico de Caracteres (OCR) y el Dataset MNIST

El Reconocimiento Óptico de Caracteres (OCR) ha evolucionado desde enfoques basados en características diseñadas manualmente hacia métodos de aprendizaje profundo capaces de aprender representaciones directamente a partir de imágenes con preprocesado mínimo [26], [27]. En reconocimiento de escritura manuscrita, este cambio ha mejorado la capacidad de generalización frente a la variabilidad de estilos de escritura, y las redes convolucionales se han consolidado como una de las arquitecturas más eficaces para este tipo de tareas [5], [26].

La base de datos empleada en este proyecto es MNIST, uno de los conjuntos de referencia más utilizados para reconocimiento de dígitos manuscritos [4]. Este dataset contiene 60 000 imágenes de entrenamiento y 10 000 de prueba, con imágenes normalizadas en escala de grises y resolución de 28×28 píxeles [5]. Su carácter estandarizado ha favorecido su uso para validar modelos de clasificación y comparar arquitecturas de redes neuronales y aceleradores hardware [4]. Además, las etapas de preprocesado, como la normalización y el formateo adecuado de las entradas, siguen siendo relevantes para optimizar el comportamiento del sistema de reconocimiento [5].

2.3. Paradigmas de Cómputo para Inteligencia Artificial: CPUs, GPUs y FPGAs

2.3.1. Arquitecturas basadas en Software (CPU y GPU)

La ejecución de redes neuronales profundas se ha apoyado tradicionalmente en CPU y, sobre todo, en GPU, debido a la elevada demanda computacional y al paralelismo requerido por estos modelos [6], [12]. Las CPU ofrecen flexibilidad generalista, pero suelen resultar menos adecuadas cuando la inferencia exige una gran capacidad de cómputo paralelo, mientras que las GPU se han convertido en la plataforma dominante por su alta capacidad de procesamiento [6]. No obstante, la complejidad computacional y de almacenamiento de las redes neuronales sigue dificultando su despliegue eficiente en plataformas convencionales, especialmente en entornos con restricciones de energía y memoria [6], [12].

2.3.2. Computación en el Borde (Edge Computing)

Para afrontar estas limitaciones, la computación en el borde propone ejecutar la inferencia cerca de la fuente de datos en lugar de depender continuamente de infraestructuras en la nube [28]. Este desplazamiento reduce la latencia y favorece la toma de decisiones en tiempo real, además de disminuir la transmisión de datos y mejorar aspectos de privacidad y seguridad [28], [29]. Sin embargo, los dispositivos edge operan con recursos limitados de memoria, potencia y capacidad de cálculo, por lo que requieren modelos compactos y estrategias de optimización específicas [12], [30].

2.3.3. FPGAs como aceleradores de hardware

Las FPGA se han consolidado como una plataforma intermedia entre flexibilidad y eficiencia para desplegar redes neuronales en hardware embebido [6], [31]. Su principal ventaja es la posibilidad de definir arquitecturas específicas para cada aplicación, explotando paralelismo y reduciendo transferencias de datos innecesarias [12], [29]. Además, permiten aprovechar de forma diferenciada recursos como LUTs, DSPs y memoria interna, lo que resulta especialmente útil en operaciones de inferencia basadas en multiplicación-acumulación y almacenamiento local de parámetros [32]. En comparación con plataformas puramente software, las FPGA suelen ofrecer mejor eficiencia energética y menor latencia cuando el diseño se adapta al modelo y a las restricciones del dispositivo [6], [31].

2.4. Optimización de Redes Neuronales para Hardware

2.4.1. Cuantificación y aritmética de punto fijo

Dado que el objetivo final de la red neuronal es su despliegue en hardware embebido, la optimización numérica del modelo es un paso necesario. La cuantificación se ha convertido en una técnica central porque transforma redes en coma flotante en representaciones de baja precisión, reduciendo el coste de memoria, latencia y consumo energético [30], [33]. En FPGA, las representaciones enteras o de punto fijo son especialmente atractivas porque permiten reemplazar operaciones en coma flotante por cálculos más simples y eficientes [34]. La cuantificación a 8 bits es una elección frecuente por su buen equilibrio entre simplicidad hardware y precisión, mientras que enfoques más recientes de precisión mixta asignan distintos

anchos de palabra según la capa o la distribución de pesos para mejorar el aprovechamiento de recursos [32], [33], [35].

2.4.2. El compromiso entre precisión y recursos

La reducción de precisión introduce un compromiso directo entre exactitud y coste hardware [33]. En general, disminuir el número de bits reduce el uso de memoria y de recursos de implementación, pero una cuantificación excesiva puede degradar la precisión del clasificador [28]. Por ello, la literatura reciente se orienta hacia metodologías hardware-aware, donde la selección de precisión no se basa solo en el tamaño del modelo, sino también en latencia, energía y utilización efectiva de recursos [28], [32]. En este contexto, la evaluación del diseño mediante relaciones entre precisión, consumo y coste hardware resulta especialmente adecuada para sistemas de inferencia sobre FPGA [35].

2.5. Trabajos relacionados con el proyecto y soluciones afines

El reconocimiento de dígitos manuscritos sobre hardware embebido es un campo ampliamente estudiado, y MNIST sigue utilizándose como banco de pruebas estándar para comparar modelos y aceleradores [4], [5]. En software, las CNN continúan dominando por precisión en esta tarea [26], [36]. En hardware, en cambio, se exploran arquitecturas diversas, como MLP, CNN, redes cuantizadas, binarias e incluso de espigas, dependiendo de las restricciones de consumo, latencia y recursos [29], [31]. Por ello, los estudios comparativos entre plataformas y precisiones son especialmente útiles para contextualizar los resultados no solo en términos de accuracy, sino también de throughput, eficiencia energética y utilización de recursos hardware [31], [32].

2.5.1. Implementaciones clásicas de MNIST en FPGA

El equilibrio entre precisión numérica, consumo de recursos y rendimiento constituye uno de los problemas centrales en el diseño de aceleradores neuronales sobre FPGA [31], [37]. En este espacio de diseño coexisten soluciones que conservan aritmética en coma flotante, propuestas en coma fija de baja precisión y modelos bioinspirados como las redes neuronales de espigas, cada una con compromisos distintos en exactitud, coste hardware y eficiencia

energética [29], [38], [39].

Una primera línea consiste en mantener una representación cercana a la empleada durante el entrenamiento. En este contexto, Huynh propone un acelerador para redes profundas basado en aritmética FP16, con una única capa física de cómputo parametrizable que procesa secuencialmente la red completa y almacena los pesos preentrenados en memoria interna de la FPGA [37]. Sobre MNIST, y para una topología 784-126-126-10 implementada en una Xilinx ZynQ-7000 XC7Z045, esta arquitectura alcanza una tasa de reconocimiento del 98.16 % y un rendimiento pico de 15.90 kFPS, lo que muestra que una precisión próxima a la de coma flotante puede mantenerse en FPGA a costa de una aritmética más compleja [37].

Frente a ello, la reducción de precisión mediante coma fija suele ser más adecuada en entornos embebidos con recursos limitados. Si y Harris describen la implementación de un MLP sobre una FPGA Cyclone IV E con representación de 8 bits, obteniendo una precisión del 89 % y un tiempo de ejecución comparable al de una solución software en 32 bits, pero funcionando a una frecuencia de reloj 144 veces menor, lo que favorece la reducción del consumo dinámico [38]. Además, al reducir la precisión de 8 a 4 bits, la precisión desciende del 89 % al 78 %, mientras que el área ocupada se reduce en un 41 %, ilustrando de forma clara el compromiso entre exactitud y coste hardware [38].

Una tercera vía consiste en recurrir a modelos bioinspirados para evitar parte del coste asociado a multiplicadores y operaciones densas. Han et al. presentan una implementación de una SNN (red neuronal artificial que imita el comportamiento del cerebro mediante picos de actividad llamados "espagas") sobre FPGA de Xilinx en la que la información se procesa mediante impulsos discretos en el tiempo, lo que simplifica el cómputo y favorece un consumo reducido [39]. Su arquitectura soporta hasta 16384 neuronas y 16.8 millones de sinapsis, consume 0.477 W y alcanza una precisión del 97.06 % sobre MNIST con un rendimiento de 161 FPS [39]. Trabajos más recientes refuerzan esta dirección al mostrar que los aceleradores SNN sobre FPGA resultan especialmente adecuados para inferencia eficiente en el borde y pueden lograr latencias de cientos de microsegundos con consumos del orden de cientos de miliwatts [29].

Trabajo	Arquitectura	Exactitud	Precisión numérica	FPGA
[37]	MLP	98,16 %	FP16	Zynq-7000
[38]	MLP	89 %	Int8	Cyclone IV
[39]	SNN	97,06 %	Spikes	Xilinx
Este trabajo	MLP	97,7 %	Int8	Basys-3

Tabla 1: Comparación de trabajos relacionados con implementaciones de redes neuronales en FPGA.

Como se observa en la Tabla 1, la mayoría de los trabajos persiguen un compromiso entre precisión, consumo de recursos y velocidad de inferencia [37], [38], [39]. En esta línea, una implementación basada en 8 bits sobre una FPGA de recursos limitados resulta coherente con la práctica habitual de priorizar eficiencia sin abandonar una precisión competitiva [31], [38].

2.5.2. Soluciones comerciales vs. Soluciones a medida

El despliegue de modelos de aprendizaje profundo en sistemas embebidos ha impulsado el desarrollo de marcos de inferencia orientados a dispositivos con restricciones estrictas de latencia, memoria y energía [40], [41]. En este contexto, la computación en el borde no solo busca ejecutar inferencia local, sino también adaptar el modelo y el sistema de ejecución a las capacidades reales del hardware disponible [42], [43].

En microcontroladores, el paradigma TinyML ha adquirido especial relevancia porque traslada redes neuronales a dispositivos con recursos extremadamente limitados. La evidencia reciente muestra que estos sistemas trabajan con memorias muy reducidas; por ejemplo, un STM32F746 dispone de 320 KB de SRAM y 1 MB de flash, y en general la memoria de los microcontroladores es entre dos y tres órdenes de magnitud menor que la de los teléfonos móviles [40], [42]. Estas restricciones obligan a rediseñar tanto el modelo como el motor de inferencia para que la ejecución local sea viable [40], [41].

En este escenario, la cuantificación a 8 bits se ha consolidado como una técnica básica para reducir tamaño de modelo y coste computacional con pérdidas moderadas de precisión [42], [43]. También se exploran enfoques más agresivos, como redes binarias, que sustituyen

multiplicaciones por operaciones lógicas y reducen de forma drástica el consumo de hardware y memoria [31], [44]. A su vez, la cuantificación mixta permite asignar distintas precisiones a pesos y activaciones según la sensibilidad de cada capa, mejorando el compromiso entre memoria, latencia y exactitud [44].

La tendencia más reciente se orienta hacia soluciones a medida basadas en codiseño algoritmo-sistema, donde la arquitectura neuronal, la planificación de memoria y la precisión numérica se optimizan conjuntamente [40], [42]. MCUNet muestra este enfoque al combinar búsqueda de arquitecturas con un motor de inferencia eficiente en memoria, logrando reducir el uso pico de memoria y acelerar la inferencia frente a bibliotecas embebidas convencionales [42]. En conjunto, la evidencia sugiere que la inferencia eficiente en el borde depende menos de una única técnica aislada que de un equilibrio entre cuantificación, compresión, planificación de memoria y adaptación al hardware objetivo [29], [43], [44].

Capítulo 3: Metodología

En esta sección se describen los pasos seguidos para desarrollar el proyecto y cumplir con los objetivos descritos en la sección 3, incluyendo la elección de la red neuronal, la base de datos utilizada, el preprocesado de los datos, la arquitectura de la red neuronal y su entrenamiento, así como el diseño en VHDL y la implementación en FPGA. Además, se detalla la estructura del código y las diferentes comprobaciones realizadas para garantizar la reproducibilidad de los resultados.

En la Figura 2 se muestra un diagrama de flujo de los pasos seguidos:

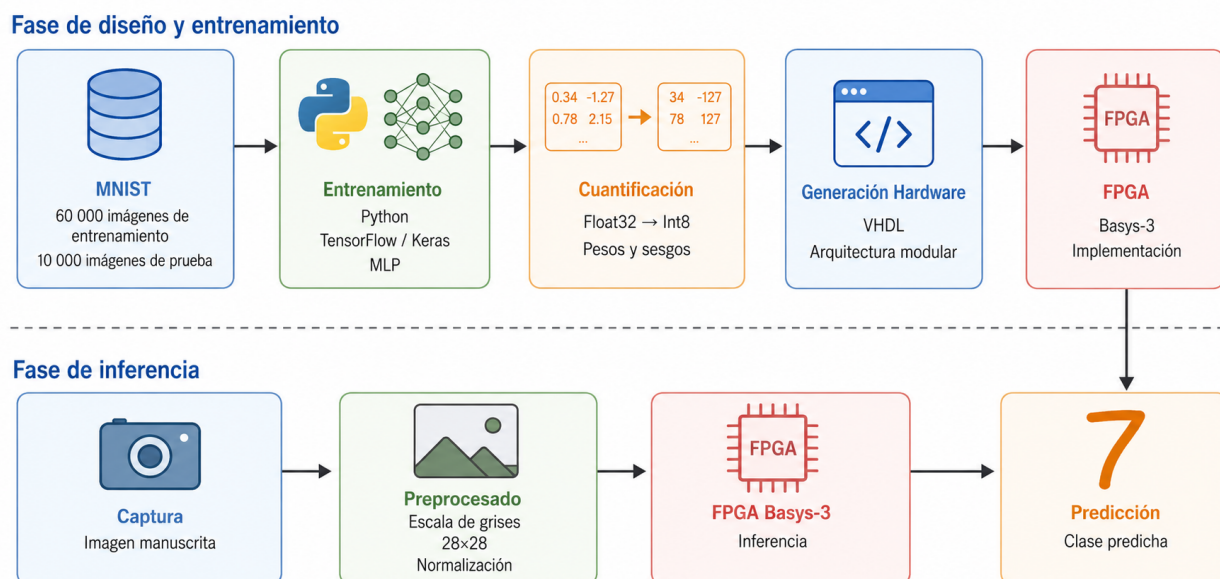


Figura 2: Diagrama de flujo de los pasos seguidos en el desarrollo del proyecto.

3.1. Elección de la red neuronal y justificación

Previo a la elección de la estructura de la red neuronal, se realizó un análisis de las diferentes arquitecturas disponibles, enfocando en el estudio de aquellas aplicadas a tareas similares de reconocimiento de dígitos. Tras dicho análisis, se seleccionó una arquitectura sencilla formada por una capa de entrada, dos capas ocultas y una capa de salida. El número de neuronas en las capas ocultas se determinó tras analizar la precisión del modelo como la

diferencia entre los costes de entrenamiento y prueba, en función de la cantidad de neuronas. Este estudio previo se enfocó en la primera capa oculta, ya que la segunda capa oculta no aportaba mejoras significativas en la precisión del modelo. De esta manera, se obtuvieron los siguientes resultados:

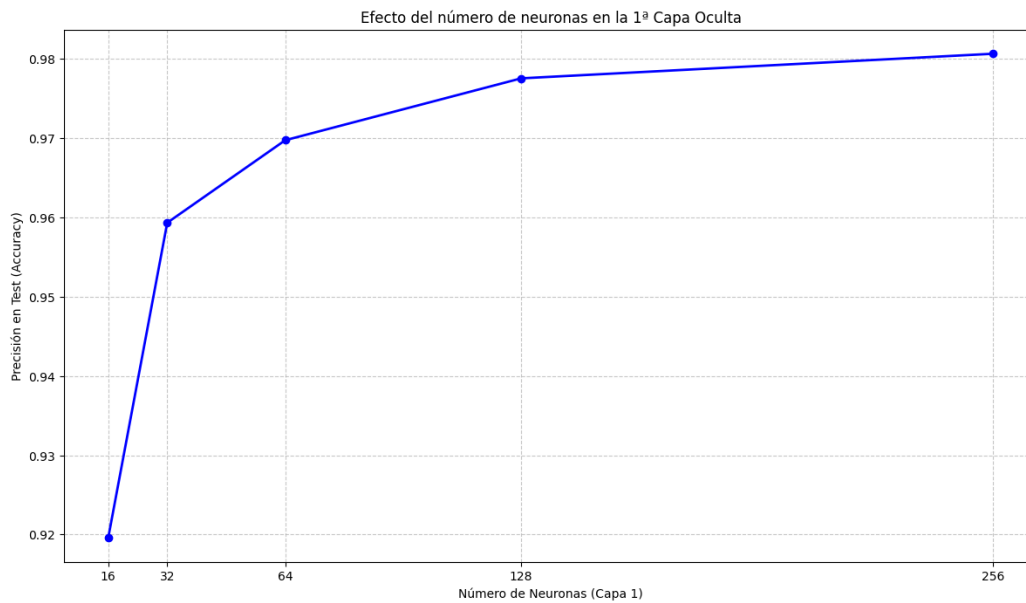


Figura 3: Precisión del modelo en función del número de neuronas en la primera capa oculta.

Se puede observar que el número de neuronas óptimo es de 128 debido a su equilibrio entre precisión y complejidad. Si se optará por un número mayor de neuronas, se obtendría una arquitectura más compleja, que posteriormente se traduciría en un mayor consumo de recursos en la FPGA, sin aportar mejoras significativas en la precisión del modelo. En el caso de optar por un número inferior al determinado como óptimo, se obtendría una arquitectura más sencilla pero menos fiable.

Además de un estudio de la precisión es necesario comprobar que no haya tanto overfitting como underfitting. Esto puede suceder porque las redes neuronales son modelos estadísticos que se entrenan con un conjunto de datos, de tal manera que pueda aprenderse este conjunto de datos de memoria y ser incapaz de predecir aquellos datos ajenos al conjunto o al contrario, ser tan simple que no es ni capaz de predecir los de la propia base de datos. De este modo, se realizó un estudio de los errores tanto en la fase de entrenamiento como en la de prueba

modificando de nuevo, únicamente el número de neuronas de la primera capa oculta.

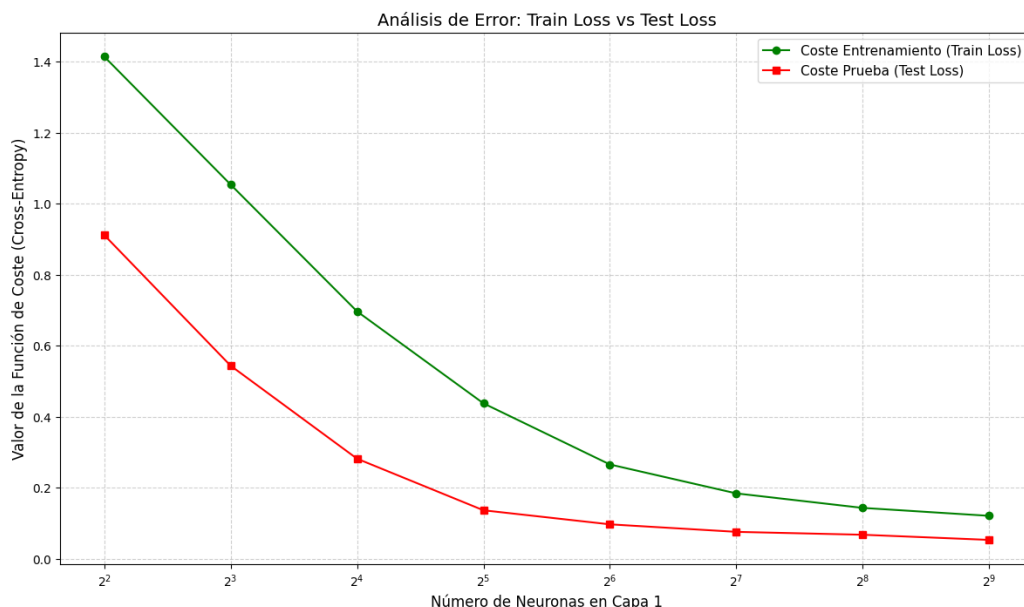


Figura 4: Costes de entrenamiento y prueba en función del número de neuronas en la primera capa oculta.

Se puede observar de nuevo que el punto de equilibrio entre reducir el error y aumentar la complejidad se encuentra en el número de neuronas correspondiente al 128. Es necesario comentar que en el entrenamiento el error es mayor debido a la configuración del modelo, que se comentará en las siguientes subsecciones. En este caso no se puede apreciar en que número de neuronas comienza el overfitting pero se observa claramente el underfitting en las opciones que se corresponden con una pequeña cantidad de nodos.

De esta manera, se opta por la siguiente arquitectura:

- Capa de entrada: 784 nodos, correspondiente al número de píxeles en una imagen de 28×28 .
- Capa oculta 1: 128 nodos, cuya función es identificar patrones y características relevantes en los dígitos: fin del trazo, líneas horizontales, verticales y diagonales, entre otros.

- Capa oculta 2: 64 nodos, que se encarga de combinar las características extraídas en la capa anterior para formar representaciones más abstractas de los dígitos.
- Capa de salida: 10 nodos, correspondientes a los dígitos del 0 al 9, que representan la probabilidad de que la imagen de entrada pertenezca a cada clase.

En la siguiente tabla se muestra la arquitectura escogida:

Capa	Entradas	Salidas	Activación
Entrada	784	-	-
Oculto 1	784	128	ReLU
Oculto 2	128	64	ReLU
Salida	64	10	Softmax

Tabla 2: Arquitectura de la red neuronal implementada.

Aunque las redes neuronales convolucionales (CNN) alcanzan habitualmente mayores tasas de clasificación sobre el conjunto de datos MNIST, también requieren un número significativamente superior de operaciones aritméticas, memoria y recursos hardware. Dado que el objetivo principal de este trabajo consiste en desarrollar un acelerador eficiente sobre una FPGA Basys-3, se optó por una arquitectura MLP, cuyo coste hardware resulta considerablemente menor manteniendo una precisión suficientemente elevada para demostrar la viabilidad de la propuesta.

3.2. Base de datos y preprocesado

Las redes neuronales son entrenadas mediante un conjunto de datos de los cuales deducen las reglas lógicas de clasificación de dichos datos. A lo largo del entrenamiento, se fijan los diferentes pesos y sesgos de las neuronas de las diferentes capas con el fin de recrear con la mayor precisión dichas deducciones lógicas. En la aplicación del reconocimiento de dígitos, la base de datos empleada para entrenar las redes neuronales es la MNIST (Modified National Institute of Standards and Technology). Este dataset consta de 60000 imágenes de entrenamiento y 10000 de prueba de dígitos manuscritos del 0 al 9. Cada imagen está en escala de grises y tiene un tamaño de 28×28 píxeles, normalizada y centrada en un cuadro

delimitador para facilitar su uso en algoritmos de clasificación. Dicha base de datos se creó combinando y remezclando los conjuntos de datos originales del NIST, que incluían dígitos escritos por estudiantes de secundaria y empleados de la Oficina del Censo de Estados Unidos.

Al tratarse de una base de datos estandarizada, existe una librería en python que contiene todas estas imágenes, tanto las de entrenamiento como las de prueba, lo cual facilita el entrenamiento del modelo en python.

Previo a entrenar la red neuronal es necesario normalizar dichas imágenes. La librería de python que contiene la base de datos se encuentra estructurada de la siguiente manera: por cada imagen se tienen dos variables, una que contiene los 28×28 píxeles en escala de grises del 0-255 y otra que tiene la etiqueta, es decir, el número representado en dicha imagen. En consecuencia, se va a trabajar con cuatro variables: dos destinadas al entrenamiento y otras dos a las pruebas para la evaluación de la precisión. Las variables que contienen los píxeles se han normalizado a valores entre 0-1, para así no provocar que la red neuronal trabaje con números tan elevados. Por otro lado, las variables que contienen las etiquetas se han de convertir a formato one-hot, por ejemplo, el 3 se transforma en $[0,0,0,1,0,0,0,0,0]$. Esto es necesario porque la red neuronal puede confundir las etiquetas por una escala de prioridad. Una vez normalizados los datos del dataset se procede a describir y entrenar la red neuronal en python.

A continuación se muestran algunas imágenes de la base de datos:

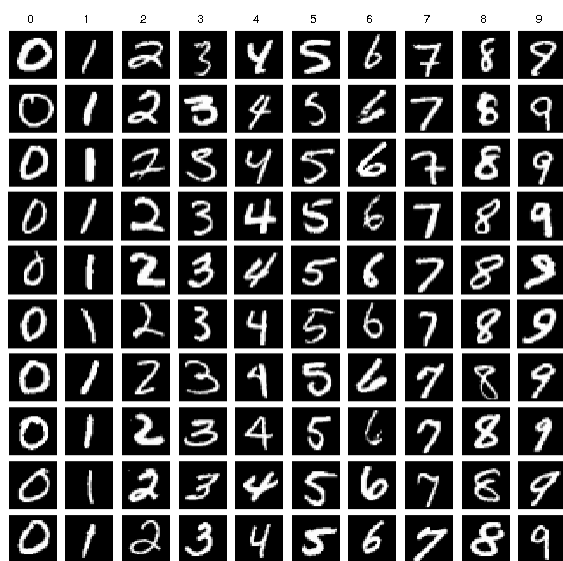


Figura 5: Imágenes de la base de datos MNIST

Se aprecia que las imágenes de esta base de datos tienen un fondo negro y trazo blanco, mientras que el papel que se emplea normalmente es blanco y los trazos suelen ser negros. Esta discrepancia entre la base de datos empleada y la realidad no supondrá un problema. El modelo será capaz de predecir las imágenes con la misma precisión, únicamente habrá que modificarlas antes de enviárselas al modelo. Como se ha mencionado previamente, se trabajará en escala de grises, lo cual facilitará esta modificación.

Tener una buena base de datos es permite que el modelo sea más preciso y como se puede apreciar en la Figura 4, la base de datos MNIST recoge una gran variedad de caligrafías.

3.3. Arquitectura de la red neuronal y entrenamiento

Una vez seleccionada la arquitectura de la red neuronal, se procede a definir el modelo en python, entrenarlo y evaluarlo. Para la definición se emplean las funciones de Sequential, Dense y Dropout de la librería Keras, enfocada en facilitar la construcción y entreno de modelos de aprendizaje profundo. La función Sequential permite crear un modelo secuencial, en el que las capas se añaden de forma lineal. La función Dense permite añadir capas densas a la red neuronal, mientras que la función Dropout permite añadir capas de regularización para evitar el sobreajuste. En la función Dense es necesario definir el tipo de función de activación que quiere que emplee el modelo. En este caso se ha optado por la activación ReLU (Rectified Linear Unit) para las capas ocultas, debido a que es la más utilizada debido a su simplicidad y buen rendimiento. Esta devuelve el propio valor de salida si es superior a cero y en caso contrario devuelve un cero. En el caso de la capa final, se ha seleccionado la función softmax, función matemática que transforma un vector de valores reales, conocidos como logits, en probabilidades que suman 1. Es la más empleada en la capa de salida en aquellas clasificaciones en las que las clases son mutuamente excluyentes. En cuanto a la función Dropout, para la primera capa oculta se ha optado por un dropout de 0.2, que consiste en apagar un 20 % de las neuronas mientras que en la segunda capa oculta se ha optado por un dropout de 0.1 debido a su menor cantidad de neuronas. Se apaga un porcentaje de las neuronas con el fin de evitar sobreajustes, lo cual se ha visto reflejado en la Figura 4, en la que se observa un mayor error en el test entrenamiento que en el de prueba.

Tras definir el modelo se procede a entrenarlo y evaluarlo. Para ello es necesario definir diferentes parámetros, como las épocas, el tamaño de los lotes y la división de validación. Las épocas son el número de veces que la red neuronal revisa el conjunto de datos y actualiza la red

neuronal, se ha optado por 10 épocas para evitar el sobreajuste. La red neuronal no procesa todas las imágenes a la vez, sino que lo hace en pequeños grupos, la cantidad de imágenes en esos grupos se determina con el tamaño del lote que se ha fijado a 32. Por último, la división de validación se ha fijado a 0.1, que indica a la función que reserve automáticamente el 10 % de los datos de entrenamiento para usarlos como datos de validación (datos externos que la red nunca usará para actualizar sus pesos), con el fin de realizar un entrenamiento más robusto. Los parámetros empleados en el entrenamiento se recogen en la siguiente tabla:

Parámetro	Valor
Optimizador	Adam
Función de pérdida	Categorical Crossentropy
Epochs	10
Batch size	32
Dropout capa 1	0.2
Dropout capa 2	0.1

Tabla 3: Hiperparámetros empleados durante el entrenamiento.

Finalmente, se evalúa la precisión del modelo entrenado. Para ello se emplean los datos del dataset guardados para las pruebas y se obtiene una exactitud del 97,70 %. Esta precisión se considera alta debido a la simplicidad del modelo de red neuronal empleado. Para la comprobación de dicha precisión se han clasificado correctamente mediante la red neuronal varias imágenes de números manuscritos por diferentes personas.

3.4. Arquitectura general del sistema

En la Figura 2 se ha mostrado un esquema general de los pasos a seguir tanto para el diseño como para la inferencia. Los pasos más específicos que se han seguido para conseguir la correcta implementación del modelo en la FPGA se muestran en la Figura 6. En ella se aprecian pasos intermedios como la exportación de pesos o la síntesis en Vivado, los cuales no aparecían en el esquema global.

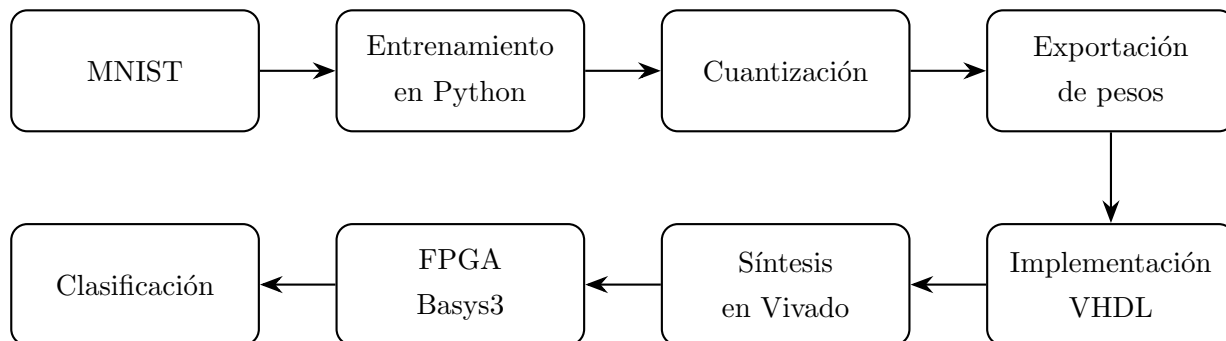


Figura 6: Diagrama de flujo del proceso de clasificación en dos filas.

En las siguientes secciones se procede a desarrollar como se han llevado a cabo los pasos desde la cuantización hasta la clasificación.

3.5. Arquitectura hardware

Para el diseño en VHDL se ha optado por representar los datos en punto fijo con 8 bits. Utilizar coma fija frente a coma flotante permite que se realicen un mayor número de operaciones simultáneamente debido al ahorro de espacio. Aunque emplear coma fija supone una pérdida de precisión debido al truncamiento y redondeo de los decimales, se ha comprobado que el modelo diseñado no se ve afectado drásticamente por dicha pérdida. La representación en 8 bits tanto de las entradas, como de los pesos y sesgos de las neuronas se debe a que este número de bits permite representar toda la información sin ocupar una gran cantidad de memoria. En consecuencia, se han extraído del modelo definido en Python los pesos y los sesgos de las neuronas y se han definido como signed de 8 bits para poder trabajar así con ellos en Vivado.

Por otro lado, en las operaciones de multiplicación y acumulación se ha optado por ampliar a 32 bits para así no tener pérdidas de información significativas. En consecuencia, la salida de cada neurona está definida por 32 bits. Para evitar trabajar con números de mayor magnitud, se seleccionan los 8 bits más significativos de cada neurona, siendo este número el dato de entrada de las neuronas de la siguiente capa. Se ha comprobado que este truncamiento no afecta de manera significativa a la precisión del modelo.

Tras elegir la representación de los diferentes datos, se ha procedido a diseñar la estructura que tendrá el modelo en VHDL, mostrando no solo las variables que entran y salen de cada

bloque, sino también la conexión entre ellos y el número de bits de cada señal. Todo esto se muestra en la siguiente figura:

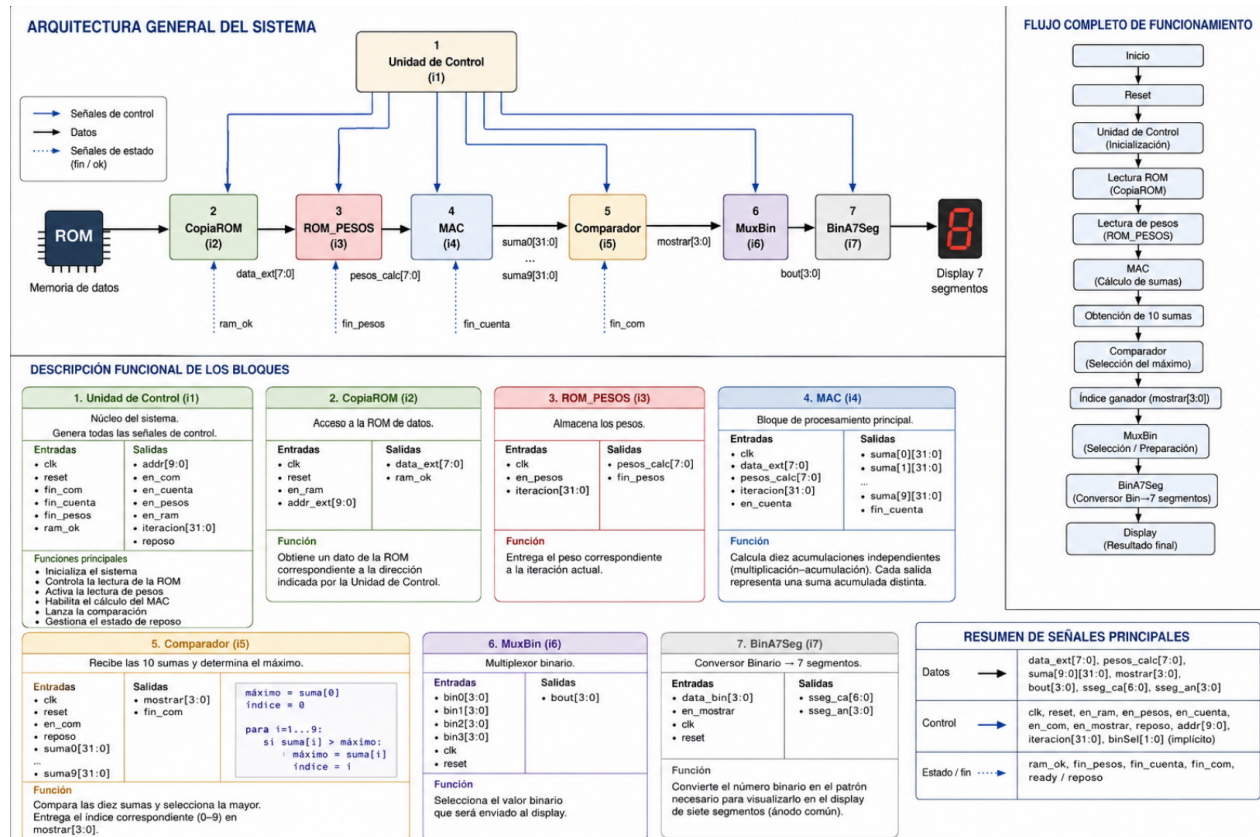


Figura 7: Diagrama de bloques del diseño en VHDL

En el Anexo B se muestra la figura con más detalle.

A continuación se describe la función de cada uno de los bloques mostrados en la Figura 7, así como la lógica de la arquitectura escogida.

- **Unidad de Control:** Este bloque se encarga de gestionar la secuencia de operaciones del sistema y de coordinar el flujo de datos entre los distintos módulos. Se basa en una máquina de estados compuesta por los siguientes estados: Reposo, CopiarRom, CargarDatos, Operacion, Incrementar, Comparar y Mostrar.

El sistema permanece en estado de Reposo hasta que se recibe una imagen. En ese momento, se procede a la carga de los datos necesarios para realizar las operaciones.

La lógica de cálculo implementada no hace uso de los bloques DSP disponibles en la FPGA. En su lugar, se emplea una única unidad MAC que opera de forma secuencial, reutilizando el hardware para procesar los distintos píxeles y pesos de la red neuronal. En cada iteración se realiza una operación de multiplicación y acumulación parcial implementada mediante lógica en LUTs, repitiéndose este proceso hasta completar la suma ponderada de cada neurona.

Este enfoque implica que no existe paralelismo a nivel de unidades de cálculo, sino reutilización de recursos, lo que reduce el área ocupada a costa de aumentar el tiempo de ejecución.

El estado de Incrementar se corresponde con el número de iteración del sistema, el cual se ha dividido en 4 fases. Al finalizar cada una de ellas, se avisa a la unidad de control y se continúa con la siguiente. Una vez completadas todas las iteraciones, se comparan los resultados obtenidos en la última capa y se selecciona el valor máximo, que corresponde al número predicho. Finalmente, este valor se muestra en los displays de la FPGA.

- **Copia ROM:** Este bloque se encarga de trasladar los datos desde una memoria ROM a una RAM, permitiendo su lectura y reutilización durante la ejecución. A diferencia de la ROM, la RAM permite la reescritura de datos.
- **ROM PESOS:** En esta memoria se almacenan los pesos de las neuronas de las distintas capas de la red neuronal. Estos valores se extraen según la lógica de control para realizar las operaciones correspondientes.
- **MAC:** Este bloque realiza las operaciones de multiplicación, acumulación y truncamiento. Se reutiliza de forma secuencial para todas las operaciones de la red neuronal, no existiendo múltiples unidades MAC en paralelo. Las multiplicaciones se implementan mediante lógica en LUTs en lugar de bloques DSP dedicados. Para evitar desbordamientos, la acumulación se realiza en 32 bits. Posteriormente, se seleccionan los 8 bits más significativos. Si el valor supera 261120, se satura al valor FF.
- **Comparador:** Este bloque recibe los valores finales de la última capa y selecciona el mayor de ellos, determinando la salida del sistema.
- **MuxBin:** Este bloque, junto con el siguiente, se encarga de mostrar el número predicho

en los displays. Actúa como multiplexor, seleccionando el dígito activo mediante la señal AN y generando la salida correspondiente.

- **BinA7Seg:** Convierte un número binario de 4 bits en su representación en 7 segmentos para poder ser mostrado en los displays.

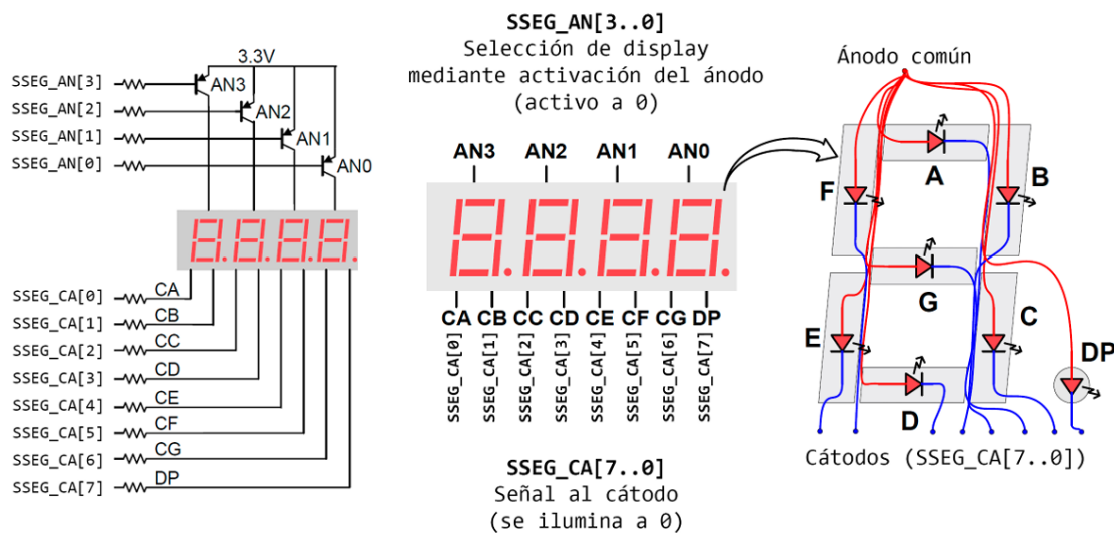


Figura 8: Funcionamiento de los displays

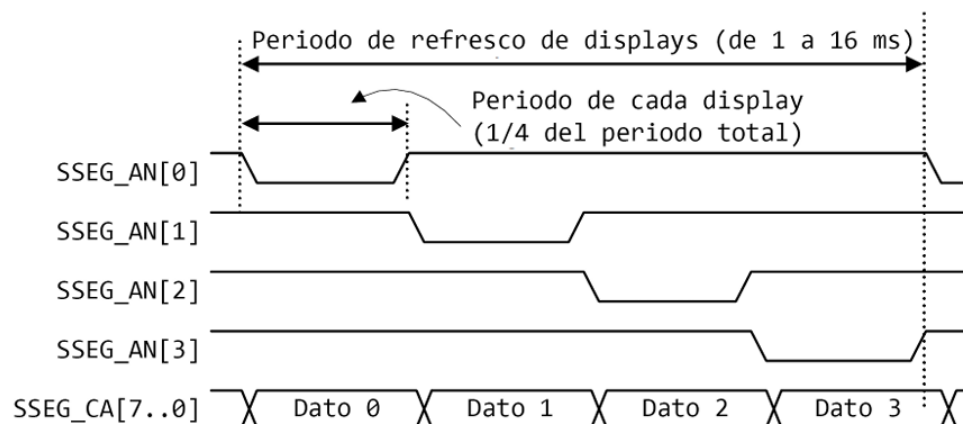


Figura 9: Funcionamiento de las señales de activación de los displays

Se observa que cada dígito se habilita individualmente con señales de selección AN0–AN3. El sistema activa los dígitos de forma secuencial a alta velocidad, de manera que el ojo humano percibe que todos están encendidos simultáneamente.

3.5.1. Modelo temporal de la arquitectura

Con el fin de analizar el comportamiento temporal del sistema, se ha desarrollado un modelo que estima el tiempo necesario para realizar una inferencia completa.

El número total de operaciones viene dado por:

$$N_{ops} = N_{in} \cdot N_1 + N_1 \cdot N_2 + N_2 \cdot N_{out} \quad (3.1)$$

El diseño implementado sigue un enfoque secuencial basado en la reutilización de una única unidad MAC, por lo que el número de ciclos depende directamente del número total de operaciones.

Dado que cada operación requiere varios ciclos de reloj (en concreto 4), se obtiene:

$$N_{ciclos} \approx 4 \cdot N_{ops} \quad (3.2)$$

y por tanto:

$$T \approx 4 \cdot \frac{N_{ops}}{f_{clk}} \quad (3.3)$$

Este modelo refleja el comportamiento real del sistema, en el que el paralelismo es limitado y viene condicionado por la reutilización de recursos.

Arquitectura	Paralelismo (P)	Modelo temporal	Tiempo estimado
Secuencial (implementada)	1	$T \approx 4 \cdot \frac{N_{ops}}{f_{clk}}$	$\sim 1,1 \text{ ms}$
Paralela ideal (teórica)	90	$T \approx \frac{N_{ops}}{P \cdot f_{clk}}$	$\sim 12 \text{ } \mu\text{s}$
Paralela con overhead (teórica)	90	$T \approx 4 \cdot \frac{N_{ops}}{f_{clk}}$	$\sim 4 \text{ ms}$
Totalmente paralela (teórica)	N_{in}	$T \approx \frac{N_{capas}}{f_{clk}}$	$\sim 0,1 \text{ } \mu\text{s}$

Tabla 4: Estimación teórica del tiempo de inferencia según el grado de paralelismo

Cabe destacar que las arquitecturas paralelas mostradas en la tabla representan escenarios teóricos y no se corresponden con la implementación realizada en este trabajo, que sigue un enfoque secuencial basado en la reutilización de recursos.

Como línea de mejora futura, se podría plantear una arquitectura con múltiples unidades MAC implementadas mediante bloques DSP trabajando en paralelo, lo que permitiría reducir significativamente el tiempo de inferencia a costa de un mayor consumo de recursos hardware.

3.6. Funcionamiento del sistema

Tras explicar la función de cada uno de los bloques empleados en el diseño de la red neuronal en VHDL, se procede a mostrar cual sería el recorrido de una imagen.

El funcionamiento del sistema comienza copiando la imagen de entrada a la RAM. A continuación se cargan los pesos, se ejecuta la inferencia, se comparan los resultados obtenidos y, finalmente, se muestra el dígito predicho.

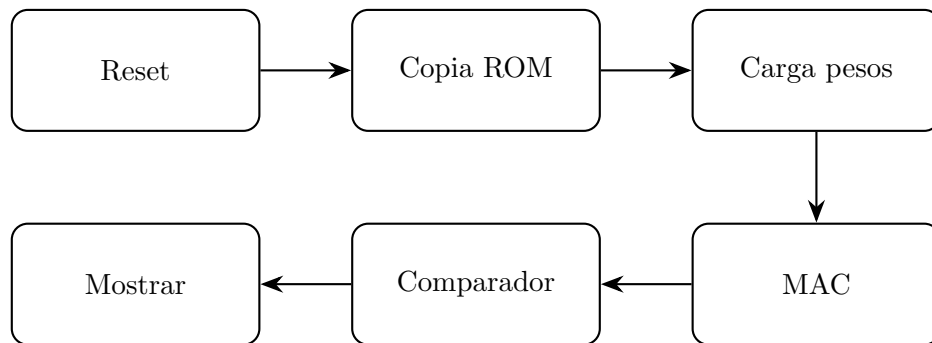


Figura 10: Diagrama de flujo del sistema en dos filas.

El diagrama resume el flujo general del sistema. En las siguientes secciones se describe con detalle el funcionamiento de cada bloque.

Se explicarán con más detalle el bloque de la Unidad de Control, que es el encargado de gestionar que todo el flujo de trabajo se lleve a cabo correctamente y el bloque MAC, encargado de realizar todas las operaciones. Los códigos de ambos bloques se muestran en el Anexo C y D, respectivamente, y se explicará su funcionamiento de forma general.

3.7. Máquina de estados de la Unidad de Control

A continuación se muestra el diagrama de estados del bloque de la Unidad de Control, que es el encargado de comunicarle a cada bloque cuando tiene que actuar:

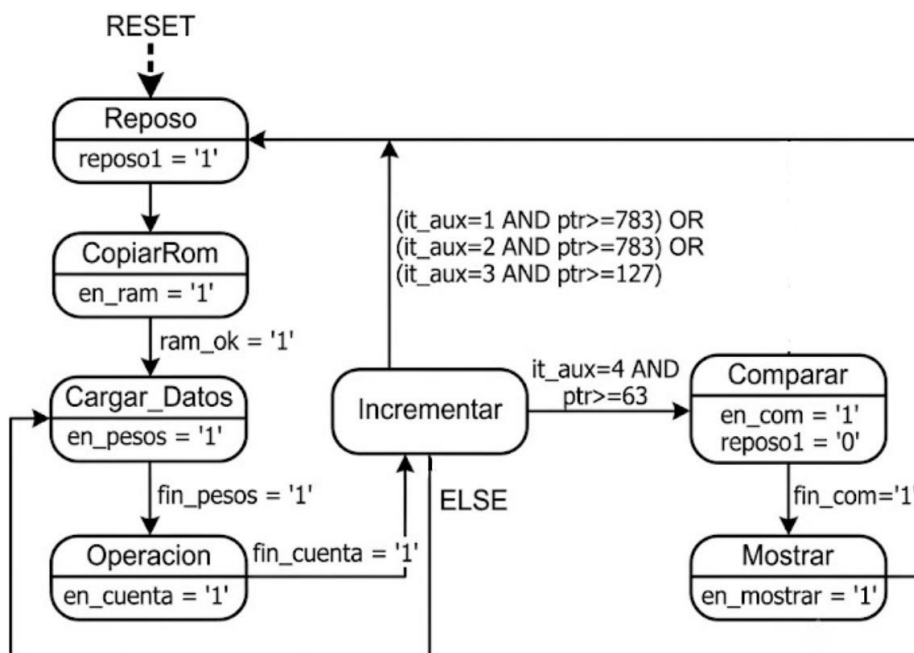


Figura 11: Diagrama de estados de la Unidad de Control

La Figura 11, muestra los siete estados que forman la máquina de estados de la Unidad de Control.

- **Reposo:** estado inicial o de espera del sistema.
- **CopiarRom:** copia de datos desde ROM a RAM.

- **Cargar_Datos:** carga de pesos y datos de entrada.
- **Operacion:** ejecución del bloque de cómputo (MAC).
- **Incrementar:** actualización de punteros e iteraciones.
- **Comparar:** evaluación de resultados finales.
- **Mostrar:** visualización de la salida.

3.7.1. Control de iteraciones y punteros

La unidad utiliza dos registros internos fundamentales:

- **it_aux:** indica la capa o iteración actual del sistema (de 1 a 4).
- **ptr:** actúa como puntero de acceso a memoria.

Se tienen 4 iteraciones porque la primera capa se divide en dos, como se explicará en las secciones correspondientes al desarrollo del bloque MAC. El número de iteración permite a este bloque saber en que capa y neuronas se encuentra.

En cuanto al puntero, dependiendo de la capa procesada, este recorre diferentes rangos de memoria, correspondientes a las entradas de cada una de las capas:

- 784 posiciones para la primera capa oculta.
- 128 posiciones para la segunda capa oculta.
- 64 posiciones para la última capa.

Cuando se alcanza el final de una capa, el puntero se reinicia a cero y se incrementa el valor de **it_aux**.

3.7.2. Transición entre estados

La transición entre estados depende de señales de control externas como **fin_cuenta**, **fin_pesos**, **ram_ok** y **fin_com**.

Por ejemplo, en el estado Operacion, el sistema permanece activo hasta recibir la señal `fin_cuenta`, indicando la finalización del cálculo. Posteriormente, se transita al estado Incrementar, donde se actualizan los punteros de memoria.

3.7.3. Generación de direcciones

La dirección de memoria se genera a partir del puntero interno:

```
addr <= to_unsigned(ptr, 10);
```

Esto permite direccionar hasta 1024 posiciones de memoria, suficientes para poder acceder a las 784 entradas que tiene la RAM.

3.7.4. Señales de salida

La unidad de control genera distintas señales de habilitación según el estado activo:

- `en_ram`: habilita acceso a memoria RAM.
- `en_pesos`: activa la carga de pesos.
- `en_cuenta`: habilita el bloque de cómputo.
- `en_com`: activa la etapa de comparación.
- `en_mostrar`: habilita la visualización de resultados.
- `reposo1`: indica estado de reposo del sistema.

En conjunto, esta unidad de control permite la correcta sincronización de todas las etapas del sistema, garantizando el flujo secuencial de datos y operaciones.

3.8. Diseño y Arquitectura Real de la Unidad MAC Unificada y Reutilizable

Todas las operaciones que se realizan en las neuronas de una red neuronal se han consolidado en un único bloque de Multiplicación y Acumulación (MAC). Las arquitecturas

tradicionales emplean varios bloques MAC, cada uno destinado a realizar las operaciones de cada una de las capas. En este proyecto se ha optado por un único bloque del cual se obtiene los resultados finales de la última capa.

La máquina de estados empleada consta de 10 estados lógicos. El comportamiento de estos estados depende de las variables descritas en la sección anterior: **ptr** y **iteracion** (valores de 1 a 4).

La Figura 12 muestra el diagrama de estados implementado en el módulo.

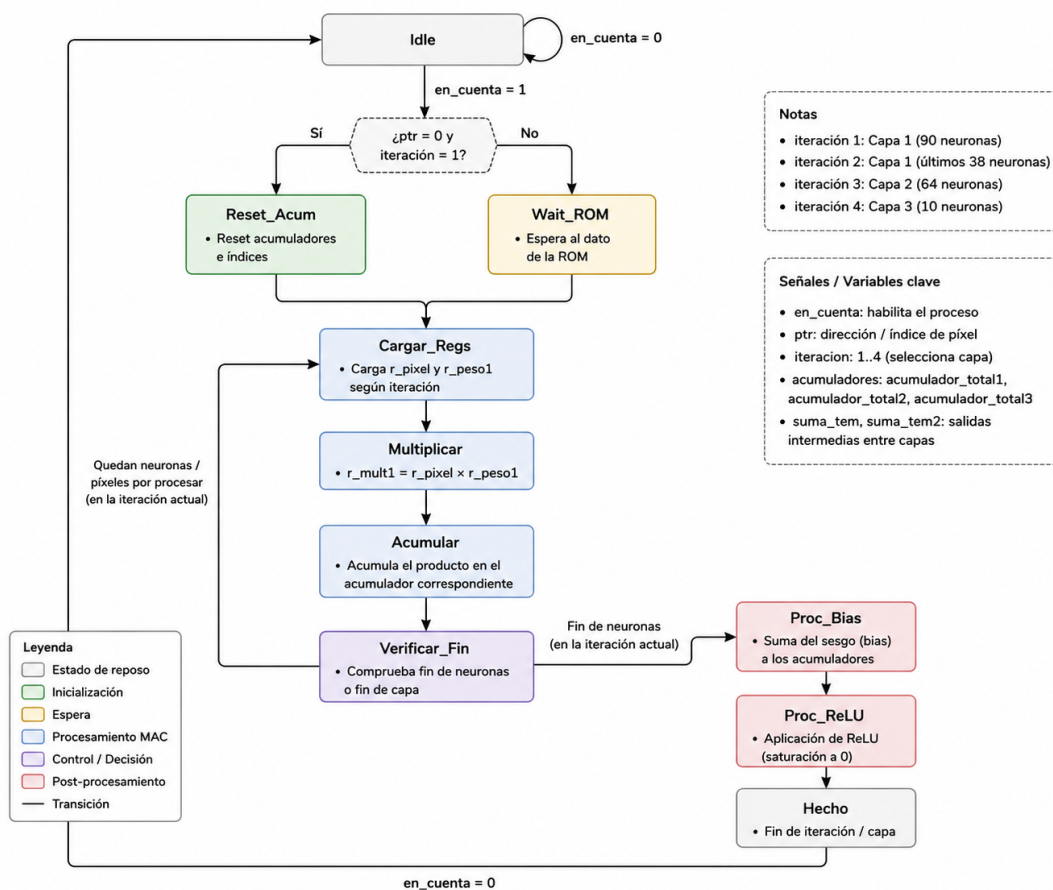


Figura 12: Diagrama de estados del bloque MAC

Este diagrama se ha simplificado con el fin de asegurar un entendimiento del funcionamiento del bloque MAC. En consecuencia, solo se muestran las variables de mayor importancia. Para un análisis más detallado, el código se encuentra en el Anexo D.

3.8.1. Camino de Datos (Datapath) y Prevención de Desbordamiento

El flujo aritmético de la unidad MAC está diseñado bajo una estricta gestión de anchos de palabra para conservar la precisión numérica tras la cuantización:

Para evitar desbordamientos o pérdidas de información se ha realizado un estricto control de anchos de palabra. El modo en el que se ha gestionado este control se explica a continuación:

- **Entradas:** Los píxeles o activaciones previas (`data_ext`) entran como valores sin signo de 8 bits y se extienden a un formato con signo de 9 bits (`signed(8 downto 0)`). Los pesos codificados en las memorias se leen en formato con signo de 8 bits (`signed(7 downto 0)`).
- **Multipliación:** El producto de ambos registros (`r_pixel × r_peso1`) genera un resultado parcial (`r_mult1`) con una resolución de **17 bits** (9 bits + 8 bits).
- **Acumulación de Alta Precisión:** Para acumular de forma segura la suma iterativa de cada neurona de cada uno de los productos realizados se ha extendido el ancho de palabra de 17 bits a 32 bits de tipo `signed` a través de la función `resize()`. En consecuencia, se asegura no caer en errores de desbordamiento. A continuación se explicará como se reduce al ancho de palabra cuando se finaliza una capa para evitar que la siguiente capa tenga que trabajar con anchos de palabra de 32bits en los productos. El sistema gestiona tres vectores independientes de acumulación según la capa activa (`acumulador_total1`, `acumulador_total2` y `acumulador_total3`).

3.8.2. Flujo de Operación por Capas (Multiplexación Temporal)

Como se ha mencionado previamente, las operaciones se dividen en 4 iteraciones, que se explican a continuación:

1. **iteracion = 1 (Capa 1 - Parte A):** El hardware procesa las primeras 90 neuronas de la primera capa oculta para los 784 píxeles de entrada. La primera capa se divide en dos iteraciones (90 + 38 neuronas) para completar las 128 neuronas del modelo, reutilizando la misma unidad MAC durante todo el proceso mediante multiplexación temporal.

2. **iteracion = 2 (Capa 1 - Parte B y Post-procesado):** se modifica el índice de la dirección (`idx_neurona + 90`) para procesar las 38 neuronas restantes necesarias para completar las 128 que conforman la primera capa oculta. Al finalizar el ciclo 783 de píxeles, la máquina de estados salta a los estados `Proc_Bias` y `Proc_ReLU`, encargados de sumar el bias y de aplicar la función de activación, respectivamente.
3. **iteracion = 3 (Capa Oculta 2):** Utiliza como entradas los resultados de la capa anterior almacenados temporalmente en el registro `suma_tem`. Lleva a cabo las operaciones de las 64 neuronas de esta segunda capa intermedia, acumulando los resultados y aplicando sus respectivos sesgos y la función de activación ReLU.
4. **iteracion = 4 (Capa de Salida / Clasificación):** Se encarga de procesar las últimas 10 neuronas (correspondientes a los dígitos del 0 al 9 del problema MNIST). Las entradas se extraen del registro temporal de activaciones `suma_tem2` de 64 posiciones. Al finalizar esta etapa, el sistema aplica la función de rectificación ReLU, limita el post-procesado al truncamiento y copia el resultado final directamente al puerto de salida `suma` en el estado `Hecho`.

3.8.3. Bloque de Post-procesamiento: Bias, Saturación y ReLU Hardware

Una vez finalizado el bucle de multiplicaciones de una neurona, el estado `Proc_Bias` añade el sesgo correspondiente (`BIAS_CAPA1`, `BIAS_CAPA2` o `BIAS_CAPA3`) sumándolo directamente sobre las variables que contienen la acumulación de cada capa, las cuales tienen un ancho de palabra de 32 bits.

Inmediatamente después, el estado `Proc_ReLU` ejecuta en un único ciclo de reloj la función de activación no lineal y la redimensión de la palabra a 8 bits sin signo (`unsigned`) mediante una lógica de saturación:

$$f(x) = \begin{cases} \text{x"00"} & \text{si } ACC < 0 \\ \text{x"FF"} & \text{si } ACC > 261120 \\ ACC(17 \text{ downto } 10) & \text{en otro caso} \end{cases} \quad (3.4)$$

Si el acumulador arroja un valor negativo, se activa la condición de la compuerta ReLU forzando la salida a cero estricto (`x"00"`). Si el valor supera la cota superior establecida en

punto fijo (261120), el circuito entra en saturación alta fijando el valor máximo (x_{FF}). En el rango intermedio de operación lineal, se extraen los bits que se encuentran en la ventana intermedia (17 down to 10) para devolver una activación limpia de 8 bits con signo corregido que alimenta a la etapa sucesiva de la red, para evitar así ir acumulando números cada vez más elevados y que ocupan más espacio y consumen más energía.

3.9. Implementación en FPGA

Una vez diseñado el modelo en VHDL y tras haber comprobado su funcionamiento mediante simulaciones que se muestran en la sección de resultados se ha procedido a la implementación en un dispositivo hardware. Para ello se ha utilizado una FPGA de la familia Artix-7 junto con el entorno de desarrollo Vivado Design Suite, siguiendo el proceso de diseño habitual para sistemas digitales descritos mediante VHDL.

El proceso de implementación comprende varias etapas: creación del proyecto, incorporación de los archivos fuente, síntesis del diseño, implementación física, definición de las restricciones hardware, generación del archivo de configuración y, finalmente, la programación y validación funcional sobre la placa.

3.9.1. Placa de desarrollo Basys3

La implementación del sistema se ha realizado sobre la placa de desarrollo **Basys3**, fabricada por Digilent y basada en la FPGA **Xilinx Artix-7 XC7A35T-1CPG236C**. Esta plataforma constituye una solución ampliamente utilizada en el ámbito académico debido a su facilidad de uso, disponibilidad de periféricos integrados y compatibilidad con las herramientas de desarrollo de Xilinx.

La FPGA Artix-7 consta de una gran cantidad de recursos lógicos facilitando la implementación de sistemas digitales de complejidad media. Estos recursos ofrecen un equilibrio entre prestaciones, consumo energético y coste. Además, la incorporación de elementos de entrada y salida permiten validar el funcionamiento del diseño sin necesidad de emplear hardware adicional.

Entre los principales recursos disponibles destacan:

- Oscilador de reloj de 25 MHz.

- 16 interruptores deslizantes.
- 16 LEDs de usuario.
- 5 pulsadores.
- Display de cuatro dígitos de siete segmentos.
- Conector USB-JTAG para programación y depuración.
- Memoria SRAM y memoria Flash.
- Conectores de expansión PMOD.

La Figura 13 muestra la placa Basys3 empleada durante el desarrollo experimental del proyecto y la cual se ha descrito previamente.

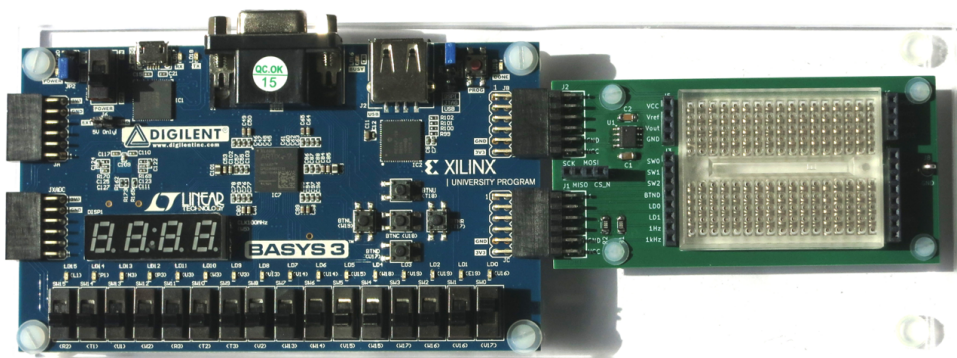


Figura 13: Placa de desarrollo Basys3 utilizada para la implementación del sistema.

3.9.2. Entorno de desarrollo Vivado

Para el desarrollo hardware se ha empleado el entorno Vivado Design Suite, desarrollado por Xilinx, que constituye la herramienta oficial para el diseño, síntesis e implementación de sistemas digitales sobre dispositivos FPGA de las familias Artix, Kintex y Virtex.

Dentro del proyecto de Vivado se han incorporado todos los módulos que se han descrito en la sección de Metodología, organizados siguiendo una jerarquía. Además, se han añadido los archivos de restricciones, que se explicarán en las siguientes páginas. La función de estos es adaptar el diseño a las características físicas de la placa empleada.

Vivado integra todas las herramientas necesarias para completar el flujo de diseño hardware, entre ellas se encuentran:

- Análisis sintáctico del código VHDL.
- Elaboración del diseño.
- Síntesis lógica.
- Implementación física.
- Análisis temporal.
- Generación del archivo de configuración (.bit).
- Programación del dispositivo FPGA.

La utilización de un único entorno de desarrollo simplifica considerablemente el proceso de implementación y permite detectar posibles errores en cada una de las etapas del flujo de diseño.

3.9.3. Proceso de síntesis e implementación

Una vez completada la descripción hardware del sistema, se ejecuta el proceso de síntesis mediante Vivado.

Durante esta etapa, la herramienta traduce la descripción realizada en VHDL a una representación lógica formada por puertas, registros, multiplexores y demás elementos digitales soportados por la arquitectura de la FPGA. De esta manera, se realizan diferentes optimizaciones destinadas a reducir el consumo de recursos y mejorar el rendimiento temporal del diseño.

Tras finalizar la síntesis, Vivado genera diversos informes que permiten conocer la utilización de los recursos disponibles en el dispositivo, tales como:

- Look-Up Tables (LUTs).
- Flip-Flops (FF).

- Bloques de memoria (BRAM).
- Bloques DSP.
- Entradas y salidas (I/O).

Estos informes se muestran y se analizan en la sección de resultados.

Posteriormente se realiza la fase de implementación, formada principalmente por dos procesos:

1. **Placement**: asignación de cada recurso lógico a una posición física dentro de la FPGA.
2. **Routing**: establecimiento de las conexiones físicas entre todos los elementos del diseño.

Una vez completada la implementación, Vivado realiza un análisis temporal (Timing Analysis) con el objetivo de comprobar que todas las señales cumplen las restricciones de tiempo establecidas para la frecuencia de funcionamiento del sistema.

La correcta superación de esta etapa garantiza que el circuito puede operar de forma segura sobre el dispositivo físico.

3.9.4. Archivo de restricciones (XDC)

Para que el diseño pueda ejecutarse correctamente sobre la placa Basys3 es necesario definir un archivo de restricciones Xilinx Design Constraints, (XDC), como se ha mencionado previamente.

Este archivo establece la correspondencia entre las señales definidas en el código VHDL y los pines físicos de la FPGA, permitiendo conectar correctamente los distintos periféricos integrados de la placa, como interruptores, pulsadores, LEDs o displays de siete segmentos.

Además, el archivo XDC especifica el estándar eléctrico utilizado por las entradas y salidas, que en el caso de la Basys3 corresponde a LVCMOS33.

De forma adicional, también se define la frecuencia del reloj principal mediante una restricción temporal, indicando a Vivado que la señal de reloj posee un periodo de 40 ns, equivalente a una frecuencia de 25 MHz.

La instrucción utilizada para definir el reloj es la siguiente:

```
create_clock -period 40 [get_ports clk]
```

El archivo de restricciones también incluye la asignación de los distintos pines físicos utilizados por el diseño. La Tabla 5 muestra un ejemplo simplificado de dicha asignación para este proyecto:

Señal	Pin FPGA	Función
clk	W5	Reloj de 25 MHz
rst	U18	Pulsador de reinicio
seg[6:0]	Varias	Display de siete segmentos

Tabla 5: Ejemplo de asignación de señales mediante el archivo XDC.

3.9.5. Programación de la FPGA

Tras completar con éxito las fases de síntesis e implementación, Vivado genera el archivo de configuración con extensión `.bit`, que contiene la información necesaria para configurar la FPGA.

La programación del dispositivo se realiza mediante la interfaz JTAG integrada en la placa Basys3 utilizando la herramienta Hardware Manager incluida en Vivado.

Durante este proceso, el archivo de configuración se transfiere desde el ordenador a la memoria de configuración de la FPGA, permitiendo que el circuito descrito en VHDL quede implementado físicamente sobre el dispositivo.

Una vez finalizada la programación, el sistema comienza a ejecutarse inmediatamente sin necesidad de realizar ninguna configuración adicional.

3.9.6. Verificación funcional

La última etapa del proceso consiste en la validación experimental del sistema implementado.

Para ello se realizaron diversas pruebas utilizando los periféricos disponibles en la placa Basys3, comprobando el correcto funcionamiento de las entradas y salidas definidas durante

el desarrollo.

En particular, se comprobó el comportamiento del botón de reinicio y de los displays, que es lo que se ha empleado en este proyecto.

3.10. Caracterización energética

Tras la definición, síntesis y comprobación funcional del modelo neuronal en VHDL, se procede a realizar la caracterización energética del sistema. Este paso es necesario para poder evaluar la eficiencia de la implementación en hardware en términos de consumo de potencia. Para poder obtener los valores relacionados con el consumo de potencia se implementará el circuito que se muestra en la Figura 14:

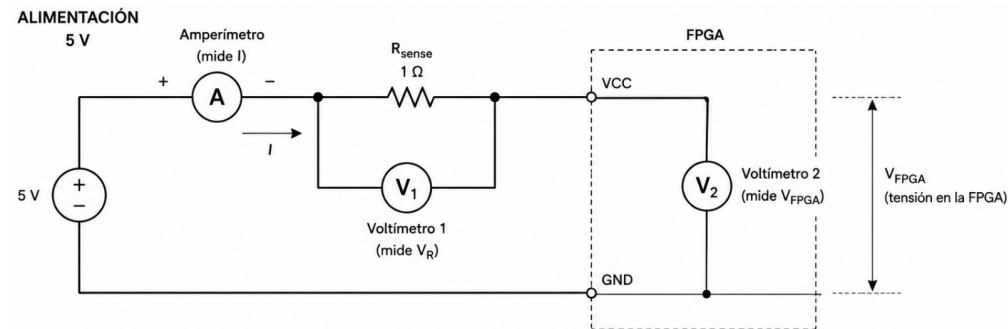


Figura 14: Circuito de medición de potencia para la FPGA

Para poder obtener el valor de la potencia que consume la FPGA, se mide la caída de tensión en el lado de alta (High-Side Current Sensing) mediante una resistencia de shunt (R_{shunt}). Esta resistencia se conecta en serie con la línea de alimentación y posee un valor nominal conocido y extremadamente bajo, en concreto, de 1. Al poseer este valor de resistencia tan bajo, se garantiza que la caída de tensión introducida en el lazo sea despreciable y no altere el correcto funcionamiento ni la estabilidad de los voltajes nominales de la FPGA.

El procedimiento de cálculo se divide en dos claras fases:

Cuando el circuito entra en operación y la FPGA ejecuta el algoritmo en VHDL, la demanda de corriente ($I_{circuito}$) atraviesa la resistencia de shunt, generando una diferencia de potencial diferencial a través de sus terminales (V_{shunt}). Aplicando la ley de Ohm, se puede obtener el valor de la intensidad que fluye hacia el dispositivo bajo prueba mediante

la ecuación 3.5:

$$I_{\text{circuito}} = \frac{V_{\text{shunt}}}{R_{\text{shunt}}} \quad (3.5)$$

A continuación, se mide la caída de tensión exacta entre los pines de alimentación específicos de la FPGA (V_{FPGA}). Finalmente, aplicando la definición de potencia eléctrica en corriente continua (Ecuación 3.6), se multiplica la intensidad calculada por dicha tensión en bornes:

$$P_{\text{FPGA}} = V_{\text{FPGA}} \cdot I_{\text{circuito}} \quad (3.6)$$

3.11. Sistema de captura

Una vez definido todo el sistema se procede al diseño del sistema de captura. Para ello, se ha escogido una cámara OV7670, cuya imagen se muestra a continuación:



Figura 15: Camara empleada como sistema de captura

Esta cámara captura imágenes en 320×340 píxeles. Para reducirlo a un tamaño de 28×28 píxeles, que es con el que se desea trabajar, se ha recortado la imagen a 224×224 píxeles. Esto implica, que solo se captura la información que se encuentra en el medio de la imagen inicial. Posteriormente, de esta imagen recortada, cada 8×8 píxeles se corresponden con un píxel de la imagen con tamaño 28×28 . Esta cámara envía dichos píxeles de forma secuencial, lo cual se tendrá en cuenta en el apartado de resultados, específicamente en la sección de análisis temporal.

Capítulo 4: Resultados

En este capítulo se presentan los resultados obtenidos tanto en el modelo de Python como en su implementación en la FPGA. Se realiza un análisis completo de la precisión del modelo, la eficiencia energética y el rendimiento temporal, comparando los resultados obtenidos en ambos entornos. Además, se analizan otros procesadores y se incluye un estudio comparativo entre estos y la implementación en FPGA.

Se realiza un análisis completo del consumo energético del sistema, además de un estudio del porcentaje de memoria utilizada en la FPGA y las implicaciones que tienen dichos resultados y como se podrían mejorar en un futuro.

4.1. Análisis del modelo en Python

En primer lugar, se analiza el modelo diseñado en Python. Se evalúa tanto la precisión global como la precisión de cada una de las clases. Para ello, se ha obtenido una matriz de confusión, que se muestra en la siguiente figura:

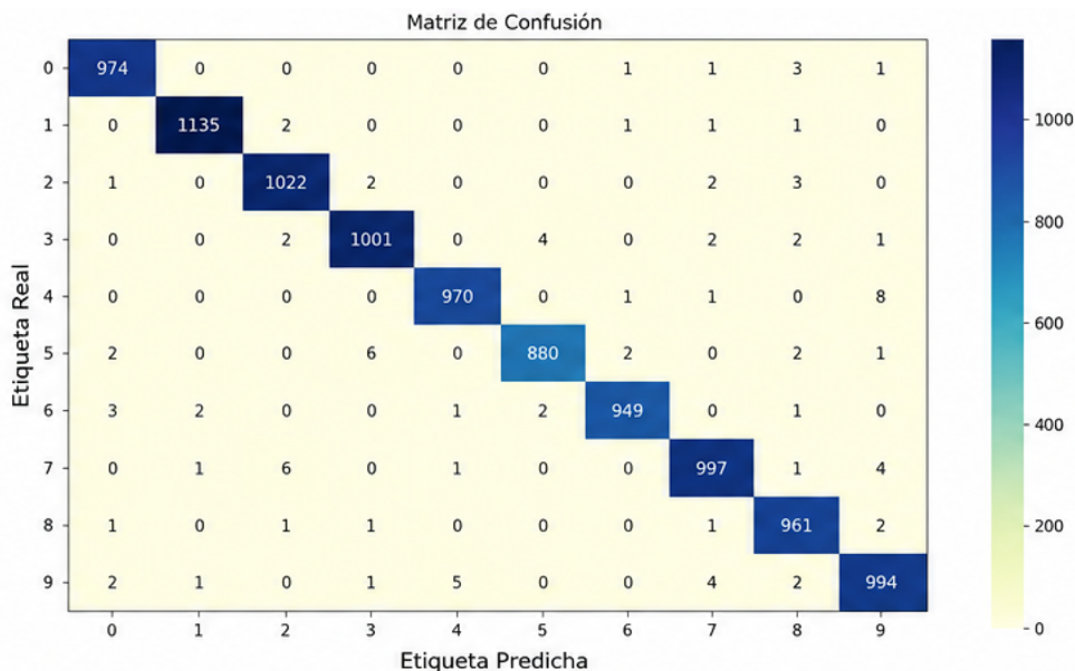


Figura 16: Matriz de confusión del modelo en Python.

Se observa que todos los números pertenecientes a la diagonal son elevados, esto implica que el modelo ha sido capaz de predecir con acierto un porcentaje muy alto en todas las clases. Fuera de la diagonal se observan números no tan elevados, implicando que el modelo no ha sido capaz de clasificar correctamente ciertos números. Los errores más comunes son entre números que se consideran similares, como por ejemplo, el 4 con el 9, o el 5 con el 3.

A continuación, se muestra una gráfica que representa la cantidad de errores que el modelo ha cometido por dígito:

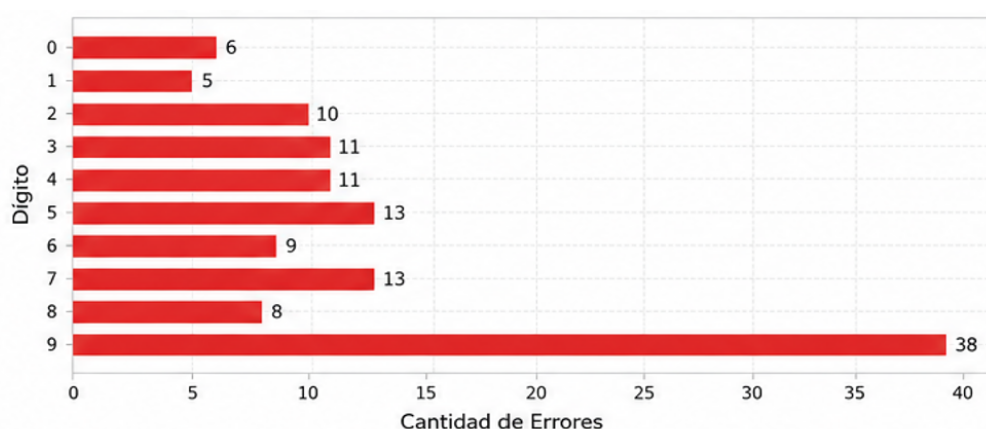


Figura 17: Errores del modelo en Python.

En la Figura 17 se observa que el número que tiene una elevada tasa de error en comparación con el resto es el 9.

Para poder comprender con mayor facilidad aquellos números que el modelo es incapaz de predecir correctamente, se ha escogido una imagen de cada clase que el modelo ha predicho de forma incorrecta. La siguiente figura muestra dichas imágenes:

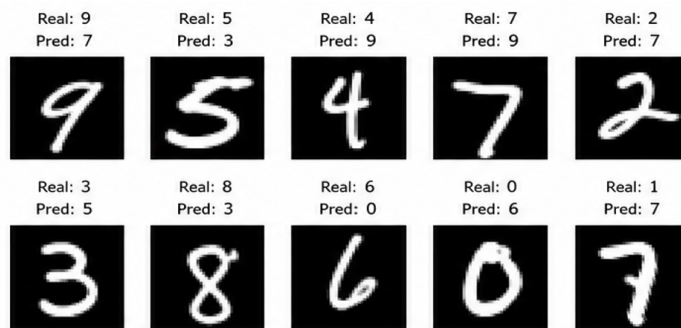


Figura 18: Errores del modelo en Python.

Una vez finalizado el diseño, entrenamiento y evaluación de dicho modelo se procedió a verificar su precisión mediante varias pruebas con diferentes imágenes.

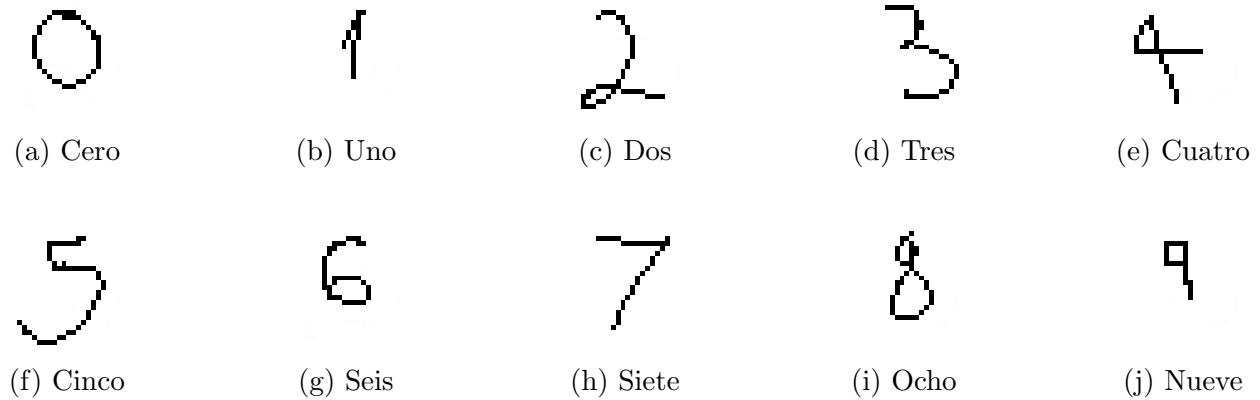


Figura 19: Título general para el grupo de 10 imágenes.

Los números representados en cada una de estas imágenes están escritos por diferentes personas, no son imágenes pertenecientes a la base de datos MNIST. Para dibujarlas y exportarlas con facilidad se ha empleado la herramienta GIMP.

La predicción del modelo ante cada una de estas imágenes se recoge en la siguiente tabla:

Subfigura	Número Real	Predicción del Modelo
(a) Cero	0	0
(b) Uno	1	1
(c) Dos	2	2
(d) Tres	3	3
(e) Cuatro	4	4
(f) Cinco	5	5
(g) Seis	6	5
(h) Siete	7	7
(i) Ocho	8	9
(j) Nueve	9	9

Tabla 6: Comparación entre el número real de la subfigura y la predicción del modelo.

Se aprecia que el modelo es capaz de predecir con una gran precisión, números escritos con diferentes caligrafías. Además, se puede apreciar la correlación entre estos resultados y la matriz de confusión mostrada en la Figura 16. El número 8 lo confunde con un 9, que es con el que lo suele confundir de manera más frecuente, lo mismo sucede con el 6, ya que lo predice como un 5.

4.1.1. Visualización de activaciones neuronales

En este apartado se analiza el comportamiento de distintas neuronas del modelo mediante la visualización de sus mapas de activación ante una misma entrada de la clase “7”. El objetivo de este análisis es entender que regiones de la imagen tienen más peso en cada una de las neuronas de la primera capa oculta y así poder tener una mejor comprensión acerca de los patrones visuales aprendidos por el modelo en el entrenamiento.

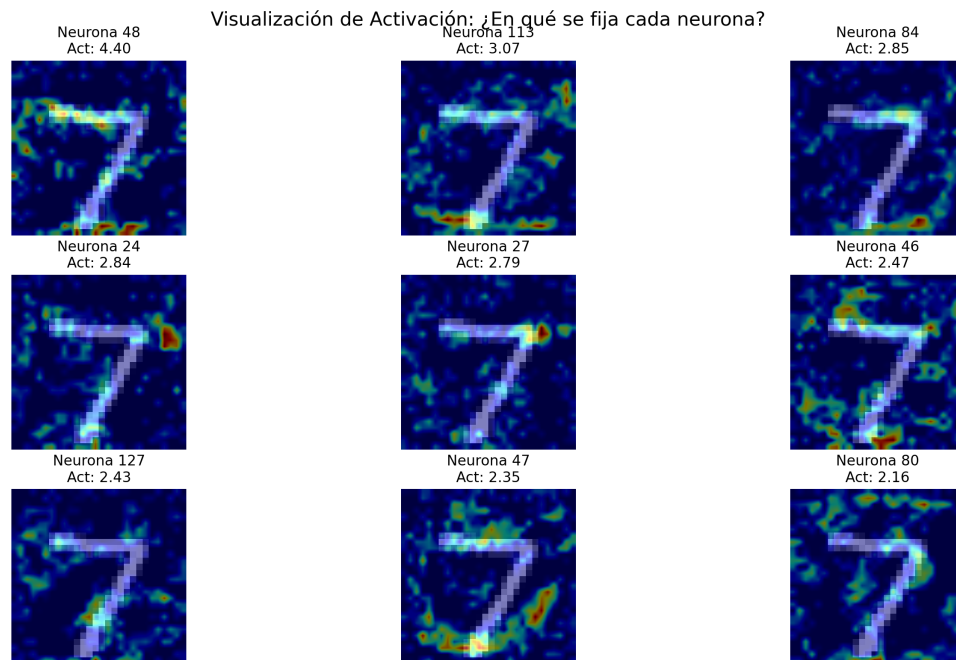


Figura 20: Visualización de activaciones de distintas neuronas ante la clase “7”

En la Figura 20 se muestran varias neuronas seleccionadas junto con sus correspondientes mapas de activación. Cada subfigura incluye el valor de activación medio (Act), lo que permite comparar la intensidad de respuesta entre neuronas. Las regiones en tonos cálidos (amarillo

y rojo) indican una mayor contribución a la activación, que se corresponden con un peso de mayor valor en el modelo, mientras que los tonos fríos (azul) representan zonas con menor importancia.

Se aprecia que cada neurona se fija en una parte distinta del dígito que se está analizando, aunque de forma global las neuronas se activan ante la presencia del trazo del “7”. Por un lado, algunas neuronas se fijan más en el trazo de la barra horizontal en la parte superior, mientras que por otro lado otras le dan mayor importancia a la diagonal principal o al final del trazo en la parte inferior. De esta manera, se entiende como el modelo no se fija en el conjunto global del trazo sino que en características específicas.

La diferencia en la intensidad de las activaciones en las neuronas se debe al nivel de selectividad que tienen cada una de estas. Las neuronas con valores de activación más altos tienden a enfocar su respuesta en zonas más concretas de la imagen mientras que aquellas que tienen unos valores de activación más bajos muestran patrones más difusos.

En consecuencia, estos resultados permiten entender los patrones visuales en los que se fija el modelo y que este no se ha aprendido los números de memoria, sino que se fija en ciertos trazos en específico para posteriormente unirlos y predecir el número con la mayor precisión posible.

4.2. Comprobación del modelo en VHDL y comparación

A continuación, se procedió a comprobar el diseño desarrollado en VHDL. Este proceso se realizó en los siguientes pasos:

1. Validación del número obtenido: Se envió la misma imagen a la red neuronal definida en Python y al modelo diseñado en VHDL con el fin de comparar dichos modelos y verificar el funcionamiento del segundo.
2. Comprobación de valores finales: Hubo varios casos en los que el diseño de VHDL predijo de forma errónea el número mientras que el modelo de Python lo hizo de forma correcta. Ante esta situación se revisaron los valores finales y los valores que se obtenían en la salida de cada capa comparándolos con aquellos obtenidos en el modelo de Python. Al analizar las diferencias entre estos valores se procedió a modificar ciertos aspectos del código de VHDL.

Una vez verificado el funcionamiento correcto del modelo definido en VHDL, se procedió a enviar las mismas imágenes a ambos modelos. Se enviaron las siguientes 5 imágenes y se compararon las predicciones obtenidas:

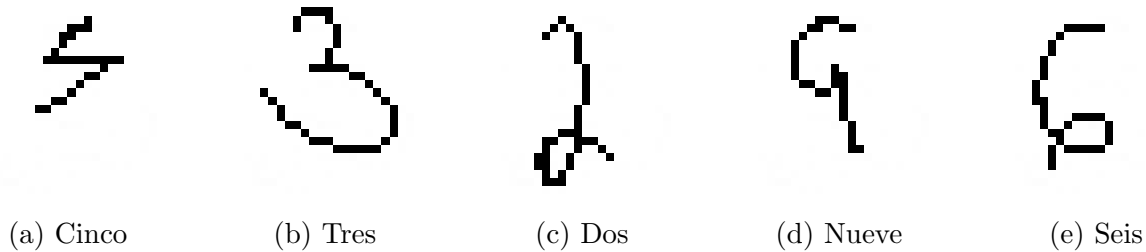


Figura 21: Grupo de las primeras 5 imágenes analizadas.

En la siguiente tabla se recogen tanto el número que se representa en cada una de las imágenes mostradas como las predicciones del modelo en Python y en VHDL.

Subfigura	Número en Foto	Predicción Python	Predicción VHDL
(a) Cinco	5	9	9
(b) Tres	3	3	3
(c) Dos	2	2	2
(d) Nueve	9	9	2
(e) Seis	6	6	6

Tabla 7: Comparativa de predicciones entre el modelo en Python y la implementación en VHDL.

Tras analizar diferentes imágenes en las que se muestran números escritos por personas distintas se ha llegado a la conclusión de que la precisión que se pierde al implementar el modelo en VHDL es insignificativa. En la tabla 2, se recogen los resultados de las predicciones y se observa que ambos modelos se equivocan en la misma imagen y solo hay una discrepancia en una imagen, en la que el modelo de Python la predice de forma correcta mientras que el modelo en VHDL no. Estas pruebas confirman lo que se lleva defendiendo a lo largo de este trabajo. La red neuronal no es 100 % precisa y la implementación en la FPGA no alcanza la misma precisión que el modelo en Python, pero la diferencia es muy pequeña en

comparación con el ahorro en el consumo que se obtiene. Los resultados que defienden este último argumento se mostrarán en las siguientes secciones.

4.3. Impacto de la cuantificación

A lo largo de este proyecto se ha afirmado que la cuantificación realizada para la implementación del modelo en la FPGA no afectaba de forma significativa a la precisión de este. En esta sección se confirmará dicha afirmación. Para ello se ha realizado una comparación entre la implementación en coma flotante (float32) y su equivalente en punto fijo de 8 bits (int8).

Representación	Precisión (%)	Pérdida
Coma flotante (float32)	97,70	-
Punto fijo (int8)	97,65	0,05

Tabla 8: Impacto de la cuantificación en la precisión del modelo

Los resultados obtenidos muestran que el modelo en coma flotante alcanza una precisión del 97,70 %, mientras que el modelo cuantificado presenta una precisión del 97.65 %. Esto supone una diferencia de tan solo 0.05 %. En consecuencia, se ha comprobado que la cuantificación de los datos no afecta de forma significativa a la precisión del modelo y facilita su implementación en la FPGA.

De esta manera, la cuantificación a 8 bits se presenta como una solución óptima para este tipo de aplicaciones, ya que equilibra de manera adecuada la eficiencia hardware y la precisión del modelo.

4.4. Simulaciones y análisis temporal

Previo a la implementación en la FPGA, es necesario hacer varias simulaciones del diseño, para comprobar que no tarda mucho en predecir el número. De esta manera, se empleó el siguiente testbench para la simulación:

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

```
3 use IEEE.NUMERIC_STD.ALL;
4 use work.variables_pkg.all;
5
6 entity tb_UnaNeurona is
7 end tb_UnaNeurona;
8
9 architecture Sim of tb_UnaNeurona is
10
11     component UnaNeurona
12     port(
13         clk_hardware : in std_logic;
14         reset        : in std_logic;
15         numero_final : out std_logic_vector(3 downto 0)
16     );
17 end component;
18
19 signal clk          : std_logic := '0';
20 signal reset        : std_logic := '0';
21 signal numero_final : std_logic_vector(3 downto 0);
22
23 constant clk_period : time := 10 ns;
24
25 begin
26
27     uut: UnaNeurona
28     port map (
29         clk_hardware => clk,
30         reset        => reset,
31         numero_final => numero_final
32     );
33
34     clk_process : process
35     begin
36         clk <= '0';
37         wait for clk_period/2;
38         clk <= '1';
39         wait for clk_period/2;
40     end process;
41
42     stim_proc: process
43     begin
44         reset <= '1';
45         wait for 100 ns;
46
47         reset <= '0';
48
49         report "Iniciando procesamiento de 784 entradas...";
50
51         wait for 100 ms;
52
53         assert false
```

```

54     report "Simulacion terminada por tiempo alcanzado."
55     severity failure;
56
57     wait;
58     end process;
59
60 end Sim;

```

Algoritmo 1: Código en VHDL del banco de pruebas (Testbench) purgado de errores.

Tras emplear este testbench se obtuvo la siguiente simulación:

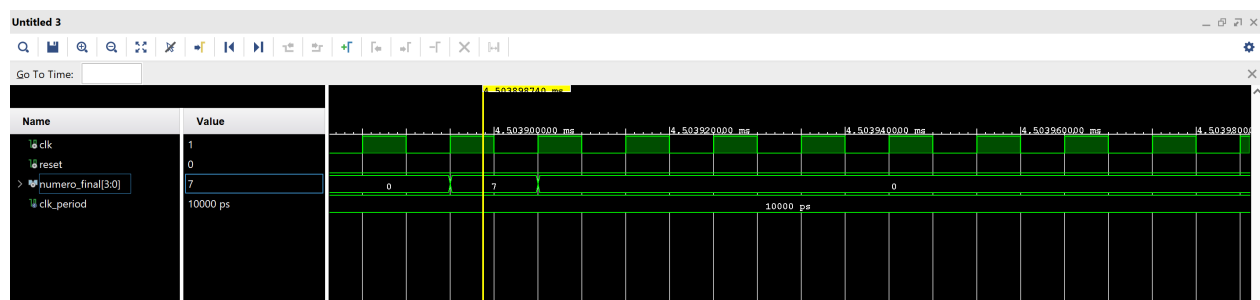


Figura 22: Simulación del diseño en VHDL

Se aprecia que las salidas no coinciden con las mostradas en el esquema del sistema que se encuentra detallado en el Anexo A. Esto se debe a que las salidas de este figura hacen referencia a los valores que van a recibir los controles de los displays en la FPGA, mientras que en la simulación el objetivo es observar si el número que se obtiene es el correcto además de obtener el tiempo que tarda en predecir dicho número. De esta manera, se ha omitido el estado de mostrar y se ha empleado como variable de salida aquella que sale del bloque de comparación.

Los resultados de la simulación realizada mostrados en la Figura 22, muestran un tiempo de ejecución de 4.5ms para predecir un dígito. Esto no es del todo cierto como se explicará en la siguiente sección.

4.4.1. Reevaluación temporal de la arquitectura implementada

Tras sintetizar e implementar el diseño sobre la FPGA, el análisis temporal realizado mediante Vivado puso de manifiesto la existencia de incumplimientos de timing para la frecuencia inicialmente prevista. Con el objetivo de garantizar el correcto funcionamiento

del sistema, se redujo la frecuencia de reloj hasta los 25 MHz. Como consecuencia de esta modificación, fue necesario recalcular las estimaciones temporales presentadas en la Sección de Metodología.

El número total de operaciones de multiplicación-acumulación (MAC) necesarias para procesar una inferencia completa viene dado por:

$$N_{ops} = N_{in} \cdot N_1 + N_1 \cdot N_2 + N_2 \cdot N_{out} \quad (4.1)$$

donde $N_{in} = 784$ corresponde al número de píxeles de entrada, $N_1 = 128$ y $N_2 = 64$ representan las neuronas de las capas ocultas, y $N_{out} = 10$ corresponde al número de clases de salida. Sustituyendo estos valores se obtiene:

$$N_{ops} = 784 \cdot 128 + 128 \cdot 64 + 64 \cdot 10 = 109184 \quad (4.2)$$

Considerando una frecuencia de reloj de 25 MHz, el periodo de reloj es:

$$T_{clk} = \frac{1}{25 \cdot 10^6} = 40 \text{ ns} \quad (4.3)$$

A partir de estos valores se recalcularon los tiempos teóricos para los distintos escenarios estudiados.

En una arquitectura secuencial ideal, donde cada operación MAC se ejecuta en un único ciclo de reloj, el tiempo de inferencia sería:

$$T_{sec} = \frac{109184}{25 \cdot 10^6} = 4,37 \text{ ms} \quad (4.4)$$

Si se aprovecharan simultáneamente los 90 bloques DSP disponibles en el dispositivo, el tiempo de inferencia se reduciría a:

$$T_{ideal} = \frac{109184}{90 \cdot 25 \cdot 10^6} = 48,6 \text{ } \mu s \quad (4.5)$$

Sin embargo, la arquitectura desarrollada reutiliza el bloque MAC mediante una máquina de estados, introduciendo ciclos adicionales asociados a la carga de datos, acumulación de

resultados y control de flujo. Considerando un coste aproximado de cuatro ciclos por operación MAC, el tiempo estimado de inferencia resulta:

$$T_{FPGA} = 4 \cdot \frac{109184}{25 \cdot 10^6} = 17,48 \text{ ms} \quad (4.6)$$

De esta manera, se analizaron dos escenarios extremos relacionados con el acceso a los datos de entrada. En el caso de una ROM completamente paralela, donde todos los datos se encuentran disponibles simultáneamente, el tiempo de acceso es prácticamente despreciable, siendo del orden de unos pocos ciclos de reloj:

$$T_{ROM} \approx 3 \cdot 40 \text{ ns} = 120 \text{ ns} \quad (4.7)$$

Por el contrario, si los píxeles se recibieran secuencialmente en flujo (streaming), el tiempo mínimo necesario para recibir una imagen completa sería:

$$T_{stream} = 784 \cdot 40 \text{ ns} = 31,36 \mu s \quad (4.8)$$

Los resultados obtenidos se resumen en la Tabla 9.

Arquitectura	Tiempo estimado
ROM paralela	120 ns
Entrada en streaming	31.36 μs
Secuencial (1 MAC/ciclo)	4.37 ms
Paralelismo ideal (90 DSP)	48.6 μs
Implementación FPGA desarrollada	17.48 ms

Tabla 9: Estimaciones temporales para una frecuencia de reloj de 25 MHz.

A partir del tiempo de inferencia obtenido para la implementación desarrollada, puede calcularse la capacidad de procesamiento del sistema, que representa el número de imágenes que la FPGA es capaz de clasificar en un segundo:

$$IPS = \frac{1}{T_{FPGA}} = \frac{1}{17,48 \times 10^{-3}} = 57,2 \quad (4.9)$$

Por tanto, la arquitectura implementada es capaz de procesar aproximadamente 57 inferencias por segundo.

4.5. Consumo energético

Una vez finalizada la fase de síntesis e implementación en Xilinx Vivado, se obtuvo el informe de consumo energético estimado mediante la herramienta Report Power. Este análisis teórico permite evaluar el consumo interno del dispositivo, es decir, exclusivamente el consumo del chip de la FPGA, sin considerar el resto de componentes de la placa. Los resultados del consumo del chip (On-Chip Power) se detallan en la Tabla 10.

Componente de Potencia	Potencia Absoluta (W)	Porcentaje (%)
Potencia Dinámica (Dynamic)	0.038	35 %
◦ Relojes (<i>Clocks</i>)	0.014	35 %
◦ Señales (<i>Signals</i>)	0.012	31 %
◦ Lógica (<i>Logic</i>)	0.009	24 %
◦ Memoria (<i>BRAM</i>)	0.004	10 %
◦ Entrada/Salida (<i>I/O</i>)	<0.001	0 %
Potencia Estática (Device Static)	0.070	65 %
Potencia Total Estimada	0.109	100 %

Tabla 10: Desglose del consumo de potencia estimado por Vivado (solo chip de la FPGA).

El sistema presenta un consumo estimado de 109 mW, valor que corresponde exclusivamente al consumo interno del chip de la FPGA. Este dato no incluye el consumo del resto de elementos de la placa, como reguladores de tensión, circuitería auxiliar o dispositivos de visualización.

Por este motivo, se ha realizado una medida experimental del consumo total del sistema completo, empleando el circuito de caracterización descrito en el Capítulo 3 basado en una resistencia de shunt. Esta medida permite obtener una estimación realista del consumo en condiciones de funcionamiento.

Al analizar el consumo del chip, se observa que la potencia estática (70 mW, 65 % del

total) constituye la base fija de disipación debida a corrientes de fuga. Por otro lado, la potencia dinámica se sitúa en 38 mW (35 %), asociada directamente a la actividad del sistema.

Dentro de la potencia dinámica, el árbol de distribución de reloj representa la mayor contribución (14 mW), seguido por las interconexiones y la lógica de cálculo. El consumo en memoria BRAM es reducido, lo cual es coherente con el hecho de que gran parte del almacenamiento se implementa mediante lógica distribuida (LUTs).

Parámetro	Valor Medido / Calculado
Tensión del sistema (V)	4,46 V
Resistencia de Shunt (R_{shunt})	1,0 Ω
Caída de tensión (V_{shunt})	0,54 V
Corriente total (I)	540 mA
Potencia total del sistema (P)	2,4 W

Tabla 11: Resultados de la caracterización energética del sistema completo.

A partir de la caída de tensión medida en la resistencia de shunt (0,54 V sobre 1 Ω), se obtiene una corriente total de 540 mA. Esto conduce a una potencia total del sistema de aproximadamente 2,4 W.

A diferencia del valor obtenido mediante Vivado, esta medida incluye el consumo de todos los elementos de la placa (FPGA, displays, reguladores y electrónica auxiliar), por lo que representa el consumo real del sistema en funcionamiento.

Para el cálculo energético se utiliza esta medida experimental, ya que es la más representativa del comportamiento real del sistema.

$$E = P \cdot T, \quad (4.10)$$

donde E representa la energía consumida por inferencia (J), P es la potencia media consumida (W) y T el tiempo de inferencia (s).

Sustituyendo los valores obtenidos, con $P = 2,4$ W y $T = 17,48$ ms:

$$E = 2,4 \cdot 17,48 \times 10^{-3} = 4,19 \times 10^{-2} \text{ J} \approx 41,9 \text{ mJ.} \quad (4.11)$$

Este resultado indica que el sistema consume aproximadamente **41.9 mJ** por inferencia completa.

4.6. Estudio comparativo entre FPGA y Raspberry Pi

Tras analizar en profundidad el rendimiento de la implementación del modelo en la FPGA, se procede a realizar un análisis del rendimiento de la misma red neuronal sobre una Raspberry Pi, con el fin de compararlos. Para ello, sobre la Raspberry Pi se han implementado diferentes versiones software: Python puro, empleando la librería Keras y OpenCV + NumPy / TensorFlow Lite.

En el Python puro se han realizado las diferentes operaciones que realiza una red neuronal, mientras que la librería Keras ejecuta operaciones optimizadas escritas en C/C++. Por otro lado, NumPy realiza operaciones vectoriales, OpenCV optimiza la lectura de imágenes y TensorFlow Lite está pensado para inferencia y elimina operaciones innecesarias del entrenamiento, por lo que la opción que implementa esta versión software es la más óptima.

La Tabla 12 muestra el tiempo de inferencia obtenido para las diferentes implementaciones software desarrolladas sobre la Raspberry Pi. Se observa que el uso de bibliotecas optimizadas, como OpenCV junto con NumPy o TensorFlow Lite, reduce significativamente el tiempo de procesamiento respecto a una implementación en Python puro, comprobando así lo explicado previamente.

Implementación	Tiempo por inferencia	Inferencias/s
Python puro	>10 ms	<100
Keras (<code>model.predict</code>)	3–6 ms	167–333
OpenCV + NumPy / TensorFlow Lite	1–2 ms	500–1000

Tabla 12: Comparación de las distintas implementaciones software sobre Raspberry Pi.

Una vez analizadas las diferentes implementaciones software, se procede a realizar un estudio comparativo entre la opción más eficiente y la implementación sobre la FPGA. Teniendo

en cuenta los resultados de inferencia y consumo de potencia obtenidos en los apartados anteriores con respecto a la implementación sobre la FPGA se ha desarrollado la siguiente tabla para poder realizar dicho análisis comparativo:

Parámetro	Raspberry Pi	FPGA
Exactitud	97,6 %	97,6 %
Tiempo por inferencia	1–2 ms	17,48 ms
Inferencias por segundo	500–1000	57
Consumo	6,7 W	2,4 W
Arquitectura	CPU ARM	Hardware dedicado
Latencia	Variable	Determinista

Tabla 13: Comparación entre la implementación software en Raspberry Pi y la implementación hardware en FPGA.

Los resultados muestran que la implementación optimizada sobre Raspberry Pi obtiene un menor tiempo de inferencia gracias a la elevada frecuencia de reloj del procesador ARM y al uso de bibliotecas altamente optimizadas para operaciones matriciales. Sin embargo, la implementación sobre FPGA presenta una latencia determinista y una arquitectura hardware específica para la aplicación, características especialmente relevantes en sistemas embebidos y aplicaciones de tiempo real.

En cuanto a consumo de potencia, la FPGA consume menos de la mitad que la Raspberry Pi, mostrando una de las grandes ventajas de emplear este tipo de hardware.

Capítulo 5: Conclusiones y trabajo futuro

5.1. Conclusiones generales

El objetivo principal de este Trabajo de Fin de Grado consistía en diseñar e implementar un sistema de reconocimiento de dígitos manuscritos basado en redes neuronales artificiales sobre una FPGA Basys 3, con una posterior evaluación frente a una implementación ejecutada sobre un procesador de uso genérico. Para ello se llevaron a cabo todas las fases del desarrollo, desde el diseño y entrenamiento del modelo hasta su implementación física y validación experimental.

5.2. Cumplimiento de los objetivos

A continuación, se analiza el grado de cumplimiento de los objetivos planteados al inicio del Trabajo de Fin de Grado.

5.2.1. Diseño y entrenamiento del modelo de reconocimiento

Este objetivo se ha cumplido de forma satisfactoria mediante el desarrollo y entrenamiento de una red neuronal multicapa utilizando Python y TensorFlow/Keras sobre la base de datos MNIST. El modelo obtenido alcanzó una elevada precisión en la clasificación de dígitos manuscritos, constituyendo una base fiable para su posterior implementación en hardware. Además, se verificó su correcto funcionamiento antes de proceder a la fase de cuantificación y exportación de parámetros.

5.2.2. Optimización aritmética para su implementación en hardware

Con el objetivo de adaptar la red neuronal a las limitaciones propias de una FPGA, se realizó la cuantificación de los pesos y sesgos desde representación en coma flotante a enteros con signo de 8 bits. Esta etapa permitió reducir considerablemente los recursos necesarios para la implementación hardware, manteniendo una precisión de clasificación adecuada para la aplicación desarrollada. De esta manera, se analizaron los efectos de dicha cuantificación

sobre el comportamiento de la red, comprobando que la pérdida de precisión resultaba insignificante.

5.2.3. Desarrollo del acelerador hardware en VHDL

Este objetivo se ha alcanzado mediante el diseño completo de la arquitectura hardware de la red neuronal empleando VHDL. Se implementaron de forma modular los diferentes bloques funcionales que componen el sistema, incluyendo las memorias de pesos y sesgos, las unidades de multiplicación-acumulación (MAC), las funciones de activación ReLU y la unidad de control encargada de coordinar el proceso de inferencia. La arquitectura desarrollada permitió ejecutar correctamente el modelo entrenado sobre la FPGA Basys 3.

5.2.4. Diseño del sistema de captura y preprocesado

También se ha cumplido el objetivo de desarrollar el sistema de captura y acondicionamiento de imágenes. Para ello se implementó la interfaz con la cámara OV7670 y el correspondiente sistema de preprocesado, adaptando las imágenes adquiridas al formato requerido por la red neuronal. De este modo, el sistema desarrollado integra tanto la adquisición de imágenes como el proceso completo de inferencia sobre la FPGA.

5.2.5. Validación e implementación física del sistema

La arquitectura diseñada fue validada mediante simulaciones funcionales y testbenches desarrollados en Vivado, comprobando el correcto funcionamiento de cada uno de los módulos implementados. Posteriormente, el diseño fue sintetizado e implementado físicamente sobre la FPGA Basys 3. Durante esta fase fue necesario reducir la frecuencia de funcionamiento hasta 25 MHz para satisfacer las restricciones temporales del diseño, obteniéndose finalmente una implementación completamente funcional y estable.

5.2.6. Análisis comparativo del rendimiento

Finalmente, se realizó un análisis del rendimiento de la implementación hardware considerando el tiempo de inferencia, el consumo de potencia, la energía consumida por inferencia y la utilización de recursos hardware. De esta manera, se comparó el comportamiento de la

FPGA con distintas implementaciones software desarrolladas sobre una Raspberry Pi, evaluando diferentes estrategias de inferencia mediante Python, Keras y OpenCV/TensorFlow Lite.

Aunque inicialmente se planteó extender esta comparación a otras plataformas de procesamiento, como GPU u otros aceleradores de inteligencia artificial, las limitaciones temporales del desarrollo del Trabajo de Fin de Grado impidieron completar dicho estudio. No obstante, la comparación realizada permite demostrar la viabilidad de la implementación sobre FPGA y manifiesta las ventajas que ofrece este tipo de dispositivos en cuanto a consumo energético y comportamiento temporal determinista.

En conjunto, puede afirmarse que los objetivos inicialmente planteados han sido alcanzados de forma satisfactoria. El trabajo desarrollado demuestra que es posible implementar una red neuronal para reconocimiento de dígitos sobre una FPGA de bajo coste, obteniendo un sistema funcional y eficiente que constituye una base sólida para futuras ampliaciones y mejoras.

5.3. Análisis crítico de los resultados

Los resultados obtenidos muestran que la implementación software optimizada sobre Raspberry Pi alcanza menores tiempos de inferencia gracias a la elevada frecuencia de reloj del procesador ARM y al uso de bibliotecas altamente optimizadas como OpenCV, NumPy o TensorFlow Lite.

Por el contrario, la implementación sobre FPGA presenta un mayor tiempo de inferencia debido a la reutilización del bloque MAC para el cálculo secuencial de todas las operaciones de la red neuronal. Esta decisión de diseño permitió reducir considerablemente la utilización de recursos hardware, aunque limitó el paralelismo efectivo de la arquitectura.

A pesar de ello, la implementación desarrollada demuestra la viabilidad de ejecutar redes neuronales sobre hardware reconfigurable utilizando una FPGA de gama educativa como la Basys 3. Además, el consumo de potencia obtenido resulta notablemente inferior al de la Raspberry Pi, confirmando que este tipo de arquitecturas constituyen una alternativa especialmente interesante para sistemas embebidos donde la eficiencia energética resulta prioritaria.

Por otra parte, el funcionamiento completamente determinista de la FPGA garantiza

tiempos de respuesta constantes, eliminando la variabilidad introducida por el sistema operativo y por la planificación de tareas presente en plataformas basadas en CPU.

5.4. Limitaciones del trabajo

5.4.1. Limitaciones hardware

La principal limitación encontrada durante el desarrollo ha sido la capacidad de almacenamiento disponible en la FPGA Basys 3. La memoria BRAM disponible restringe el tamaño de la red neuronal que puede implementarse, limitando tanto el número de capas como el número de neuronas por capa.

De esta manera, aunque el dispositivo dispone de múltiples bloques DSP48, la arquitectura implementada reutiliza un único bloque MAC para realizar todas las operaciones de multiplicación-acumulación. Esta decisión permitió reducir el consumo de recursos, pero incrementó el tiempo total de inferencia respecto a una implementación con mayor paralelismo.

5.4.2. Limitaciones temporales

Durante la implementación física del diseño fue necesario reducir la frecuencia de funcionamiento hasta 25 MHz para satisfacer las restricciones temporales impuestas por la herramienta de síntesis. Como consecuencia, fue necesario recalcular las estimaciones temporales desarrolladas durante la fase metodológica y adaptar el análisis de rendimiento a las condiciones reales de funcionamiento del sistema.

5.4.3. Limitaciones de la comparación experimental

Debido al tiempo disponible para el desarrollo del Trabajo de Fin de Grado, la comparación experimental únicamente pudo realizarse frente a distintas implementaciones software ejecutadas sobre Raspberry Pi.

Aunque esta comparación resulta suficiente para validar el funcionamiento del sistema desarrollado, habría sido interesante extender el estudio a otras plataformas de computación embebida, como dispositivos NVIDIA Jetson, FPGA de mayores prestaciones o aceleradores específicos para inteligencia artificial, permitiendo realizar un análisis más completo del

compromiso entre rendimiento, consumo energético y utilización de recursos.

5.5. Trabajo futuro

5.5.1. Implementación de un sistema OCR completo

La principal línea de trabajo futuro consiste en ampliar el sistema desarrollado hasta implementar un sistema OCR completo. Para ello sería necesario extender la red neuronal para reconocer no únicamente los diez dígitos del conjunto MNIST, sino también letras mayúsculas, minúsculas y otros símbolos.

Esta ampliación implicaría modificar tanto la arquitectura de la red neuronal como el sistema de captura y segmentación de imágenes, permitiendo reconocer documentos completos en lugar de caracteres individuales.

5.5.2. Incremento de la capacidad de la red neuronal

Otra mejora consistiría en aumentar el número de neuronas y capas ocultas de la red con el objetivo de mejorar la precisión de clasificación y permitir el tratamiento de problemas de mayor complejidad.

Esta ampliación requerirá optimizar la utilización de memoria y recursos hardware mediante nuevas estrategias de almacenamiento de pesos o técnicas adicionales de cuantificación.

5.5.3. Mayor paralelismo hardware

La arquitectura desarrollada en este trabajo se basa en la reutilización de una única unidad de multiplicación-acumulación (MAC), lo que implica un procesamiento secuencial de las operaciones de la red neuronal. Esta decisión de diseño ha permitido reducir significativamente la ocupación de recursos hardware, pero limita el paralelismo efectivo del sistema y, por tanto, el rendimiento en términos de tiempo de inferencia.

Como línea de mejora futura, resultaría especialmente interesante explotar el paralelismo intrínseco de las FPGA mediante la implementación de múltiples unidades MAC funcionando de forma simultánea. En particular, podría plantearse una arquitectura completamente

paralela en la que cada neurona, o incluso cada operación de multiplicación, disponga de su propio recurso hardware dedicado.

En este contexto, una posible evolución del diseño consistiría en utilizar los bloques DSP48 disponibles en la FPGA para implementar múltiples multiplicadores en paralelo. Por ejemplo, una arquitectura con decenas de unidades MAC operando simultáneamente (frente a la única unidad actual) permitiría reducir drásticamente el tiempo de inferencia, incrementando el número de imágenes procesadas por segundo.

No obstante, esta mejora implicaría un aumento significativo en la utilización de recursos, especialmente en términos de bloques DSP y consumo energético, por lo que sería necesario realizar un análisis detallado del compromiso entre rendimiento, consumo y ocupación de hardware.

En definitiva, la exploración de arquitecturas paralelas constituye una de las principales líneas de trabajo futuro, permitiendo aprovechar de forma más eficiente las capacidades de computación concurrente de las FPGA.

5.5.4. Ampliación del estudio comparativo

Finalmente, sería conveniente ampliar la comparación realizada incorporando otras plataformas de inferencia embebida, tanto basadas en CPU como en GPU o aceleradores específicos para inteligencia artificial.

Este estudio permitiría situar el rendimiento de la arquitectura desarrollada dentro del estado actual de las soluciones de Edge AI y analizar con mayor profundidad el compromiso entre consumo energético, rendimiento y coste hardware.

Bibliografía

- [1] J. Schmidhuber, «Deep Learning in Neural Networks: An Overview», *Neural Networks*, vol. 61, págs. 85-117, 2014. DOI: 10.1016/j.neunet.2014.09.003
- [2] L. Alzubaidi et al., «Review of Deep Learning: Concepts, CNN Architectures, Challenges, Applications, Future Directions», *Journal of Big Data*, vol. 8, n.º 1, pág. 53, 2021. DOI: 10.1186/s40537-021-00444-8
- [3] K. O'Shea y R. Nash, «An Introduction to Convolutional Neural Networks», *arXiv preprint arXiv:1511.08458*, 2015. dirección: <https://arxiv.org/abs/1511.08458>
- [4] A. Baldominos, Y. Sáez y P. Isasi, «A Survey of Handwritten Character Recognition with MNIST and EMNIST», *Applied Sciences*, vol. 9, n.º 15, pág. 3169, 2019. DOI: 10.3390/app9153169
- [5] S. Ahlawat, A. Choudhary, A. Nayyar, S. Singh y B. Yoon, «Improved Handwritten Digit Recognition Using Convolutional Neural Networks (CNN)», *Sensors*, vol. 20, n.º 12, pág. 3344, 2020. DOI: 10.3390/s20123344
- [6] K. Guo, S. Zeng, J. Yu, Y. Wang y H. Yang, «A Survey of FPGA-Based Neural Network Inference Accelerators», *ACM Transactions on Reconfigurable Technology and Systems*, vol. 12, n.º 1, págs. 1-26, 2019. DOI: 10.1145/3289185
- [7] A. Shawahna, S. M. Sait y A. El-Maleh, «FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review», *IEEE Access*, vol. 7, págs. 7823-7859, 2019. DOI: 10.1109/ACCESS.2018.2890150
- [8] A. Nechi, L. Groth, S. Mulhem, F. Merchant, R. Buchty y M. Berekovic, «FPGA-Based Deep Learning Inference Accelerators: Where Are We Standing?», *ACM Transactions on Reconfigurable Technology and Systems*, 2023. DOI: 10.1145/3583133 dirección: <https://dl.acm.org/doi/10.1145/3583133>
- [9] M. Shuvo, S. K. Islam, J. Cheng y B. I. Morshed, «Efficient Acceleration of Deep Learning Inference on Resource-Constrained Edge Devices: A Review», *Proceedings of the IEEE*, 2023. DOI: 10.1109/JPROC.2023.3274533
- [10] C. Xu et al., «The Case for FPGA-Based Edge Computing», *IEEE Transactions on Mobile Computing*, vol. 21, n.º 7, págs. 2492-2509, 2022. DOI: 10.1109/TMC.2020.3046877

-
- [11] I. Colbert, J. Daly, K. Kreutz-Delgado y S. Das, «A Competitive Edge: Can FPGAs Beat GPUs at DCNN Inference Acceleration in Resource-Limited Edge Computing Applications?», *arXiv preprint arXiv:2101.11591*, 2021. dirección: <https://arxiv.org/abs/2101.11591>
- [12] R. Wu, X. Guo, J. Du y J. Li, «Accelerating Neural Network Inference on FPGA-Based Platforms—A Survey», *Electronics*, vol. 10, n.º 9, pág. 1025, 2021. DOI: 10.3390/electronics10091025
- [13] K. Markov, «Multilayer Perceptron with Backpropagation, HDL Coder, and FPGA Technology: An Integrated Approach for Efficient Neural Network Implementation», *Problems of Engineering Cybernetics and Robotics*, 2023. dirección: <https://www.problems-cybernetics-and-robotics.org/>
- [14] H. Zhu, «Multi-layered Perceptron and Its Applications in Biotechnology», *Theoretical and Natural Science*, 2023. dirección: <https://www.ewadirect.com/proceedings/tns>
- [15] A. Ullah, M. Imran, M. A. Basit, M. Tahir y J. A. Younis, «AHerfReLU: A Novel Adaptive Activation Function Enhancing Deep Neural Network Performance», *Complexity*, 2025.
- [16] M. Varshney y P. Singh, «Optimizing Nonlinear Activation Function for Convolutional Neural Networks», *Signal, Image and Video Processing*, vol. 15, págs. 1041-1048, 2021. DOI: 10.1007/s11760-020-01843-3
- [17] A. F. Agarap, «Deep Learning Using Rectified Linear Units (ReLU)», *arXiv preprint arXiv:1803.08375*, 2018. dirección: <https://arxiv.org/abs/1803.08375>
- [18] Y. Bai, «RELU-Function and Derived Function Review», *SHS Web of Conferences*, vol. 148, pág. 01 020, 2022. DOI: 10.1051/shsconf/202214801020
- [19] Multilayer Perceptron, *Multilayer Perceptron*, Machine Learning: A Journey to Deep Learning, 2021. dirección: <https://www.machine-learning-a-journey.com/>
- [20] A. Cinar, «Training Feed-Forward Multi-Layer Perceptron Artificial Neural Networks with a Tree-Seed Algorithm», *Arabian Journal for Science and Engineering*, vol. 45, págs. 10 915-10 938, 2020. DOI: 10.1007/s13369-020-04857-9
- [21] R. Zaheer y H. Shaziya, «A Study of the Optimization Algorithms in Deep Learning», en *2019 Third International Conference on Inventive Systems and Control (ICISC)*, 2019, págs. 536-539. DOI: 10.1109/ICISC44355.2019.9036441
-

-
- [22] J. Zhang, «Gradient Descent Based Optimization Algorithms for Deep Learning Models Training», *arXiv preprint arXiv:1903.03614*, 2019. dirección: <https://arxiv.org/abs/1903.03614>
- [23] E. C. Menguc, A. Emlek y D. P. Mandic, «A Novel Backpropagation Algorithm Based on Negated Kurtosis Loss for Training Shallow, Convolutional, and Deep Neural Networks», *Neural Networks*, 2026.
- [24] F. Mehmood, S. Ahmad y T. K. Whangbo, «An Efficient Optimization Technique for Training Deep Neural Networks», *Mathematics*, vol. 11, n.º 6, pág. 1440, 2023. DOI: 10.3390/math11061440
- [25] R. Elshamy, O. Abu-Elnasr, M. Elhoseny y S. Elmougy, «Improving the Efficiency of RMSProp Optimizer by Utilizing Nesterov in Deep Learning», *Scientific Reports*, vol. 13, pág. 8951, 2023. DOI: 10.1038/s41598-023-36053-9
- [26] Y. LeCun, L. Bottou, Y. Bengio y P. Haffner, «Gradient-Based Learning Applied to Document Recognition», *Proceedings of the IEEE*, vol. 86, n.º 11, págs. 2278-2324, 1998. DOI: 10.1109/5.726791
- [27] Y. LeCun, P. Haffner, L. Bottou e Y. Bengio, «Object Recognition with Gradient-Based Learning», en *Shape, Contour and Grouping in Computer Vision*, Springer, 1999, págs. 319-345.
- [28] C. Coelho et al., «Automatic Heterogeneous Quantization of Deep Neural Networks for Low-Latency Inference on the Edge for Particle Detectors», *Nature Machine Intelligence*, vol. 3, págs. 675-686, 2021. DOI: 10.1038/s42256-021-00356-5
- [29] A. Carpegna, A. Savino y S. Di Carlo, «Spiker+: A Framework for the Generation of Efficient Spiking Neural Networks FPGA Accelerators for Inference at the Edge», *IEEE Transactions on Emerging Topics in Computing*, 2024. DOI: 10.1109/TETC.2024.3352340
- [30] L. Wei, Z. Ma, C. Yang y Q. Yao, «Advances in the Neural Network Quantization: A Comprehensive Review», *Applied Sciences*, vol. 14, 2024. DOI: 10.3390/app14146147
- [31] S. Liang, S. Yin, L. Liu, W. Luk y S. Wei, «FP-BNN: Binarized Neural Network on FPGA», *Neurocomputing*, vol. 275, págs. 1072-1086, 2018. DOI: 10.1016/j.neucom.2017.09.027
-

-
- [32] S.-E. Chang et al., «Mix and Match: A Novel FPGA-Centric Deep Neural Network Quantization Framework», en *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, págs. 208-220. DOI: 10.1109/HPCA51647.2021.00027
- [33] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen y T. Blankevoort, «A White Paper on Neural Network Quantization», *arXiv preprint arXiv:2106.08295*, 2021. dirección: <https://arxiv.org/abs/2106.08295>
- [34] B. Jacob et al., «Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference», en *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, págs. 2704-2713. DOI: 10.1109/CVPR.2018.00286
- [35] D. Lohar, C. Jeangoudoux, A. Volkova y E. Darulova, «Sound Mixed Fixed-Point Quantization of Neural Networks», *ACM Transactions on Embedded Computing Systems*, 2023. DOI: 10.1145/3584374
- [36] H. Shao, E. Ma, M. Zhu, X. Deng y S. Zhai, «MNIST Handwritten Digit Classification Based on Convolutional Neural Network with Hyperparameter Optimization», *Intelligent Automation & Soft Computing*, 2023. DOI: 10.32604/iasc.2023.032693
- [37] T. Huynh, «Deep Neural Network Accelerator Based on FPGA», en *2017 4th NAFOSTED Conference on Information and Computer Science*, 2017. DOI: 10.1109/NAFOSTED.2017.8108090
- [38] J. Si y S. Harris, «Handwritten Digit Recognition System on an FPGA», en *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, 2018. DOI: 10.1109/CCWC.2018.8301722
- [39] J. Han, Z. Li, W. Zheng e Y. Zhang, «Hardware Implementation of Spiking Neural Networks on FPGA», *Tsinghua Science and Technology*, vol. 25, n.º 4, págs. 479-486, 2020. DOI: 10.26599/TST.2019.9010051
- [40] J. Lin, L. Zhu, W.-M. Chen, W.-C. Wang y S. Han, «Tiny Machine Learning: Progress and Futures», *IEEE Circuits and Systems Magazine*, 2024. DOI: 10.1109/MCAS.2024.3381956
- [41] P. P. Ray, «A Review on TinyML: State-of-the-Art and Prospects», *Journal of King Saud University - Computer and Information Sciences*, vol. 34, n.º 6, págs. 1595-1623, 2022. DOI: 10.1016/j.jksuci.2021.11.019
-

- [42] J. Lin, W.-M. Chen, Y. Lin, J. Cohn, C. Gan y S. Han, «MCUNet: Tiny Deep Learning on IoT Devices», *arXiv preprint arXiv:2007.10319*, 2020. dirección: <https://arxiv.org/abs/2007.10319>
- [43] N. Alajlan y D. M. Ibrahim, «TinyML: Enabling of Inference Deep Learning Models on Ultra-Low-Power IoT Edge Devices for AI Applications», *Micromachines*, vol. 13, n.º 6, pág. 851, 2022. DOI: 10.3390/mi13060851
- [44] P.-E. Novac, G. B. Hacene, A. Pegatoquet, B. Miramond y V. Gripon, «Quantization and Deployment of Deep Neural Networks on Microcontrollers», *Sensors*, vol. 21, n.º 9, pág. 2984, 2021. DOI: 10.3390/s21092984

Alineación con los ODS

La Agenda 2030 para el Desarrollo Sostenible, aprobada por la ONU en 2015, constituye un marco global de referencia para la transición hacia la sostenibilidad y se articula en 17 Objetivos de Desarrollo Sostenible (ODS) y 169 metas. Este marco reconoce que retos como la educación, la desigualdad y la sostenibilidad ambiental están interrelacionados y deben abordarse de forma integrada, bajo el principio de no dejar a nadie atrás. Además, la Agenda subraya el papel de la ciencia, la tecnología y la innovación como palancas transversales para su implementación.

De esta manera, este proyecto se alinea con los siguientes Objetivos de Desarrollo Sostenible:

- **Educación de calidad (ODS 4):** El trabajo promueve la adquisición de competencias técnicas avanzadas en ingeniería electrónica y computación, en línea con la necesidad de fortalecer capacidades y competencias digitales para desenvolverse en entornos tecnológicos avanzados. Además, la literatura reciente destaca que garantizar un acceso inclusivo a tecnologías basadas en inteligencia artificial es una condición relevante para avanzar hacia una educación equitativa y de calidad.
- **Energía asequible y no contaminante (ODS 7):** Al centrarse en soluciones de bajo consumo energético, el proyecto contribuye al diseño de sistemas electrónicos más eficientes y sostenibles. La investigación en inferencia embebida muestra que los dispositivos edge están fuertemente condicionados por restricciones de energía, memoria y latencia, por lo que la optimización energética es un criterio central de diseño. Además, trabajos recientes muestran que arquitecturas y modelos optimizados pueden reducir de forma sustancial el consumo energético en nodos IoT y sistemas de inferencia en el extremo.
- **Industria, innovación e infraestructura (ODS 9):** El proyecto fomenta el desarrollo de tecnologías innovadoras y eficientes en el ámbito de los sistemas electrónicos y la inteligencia artificial. La convergencia entre sistemas embebidos, edge computing y aprendizaje automático está dando lugar a nuevos paradigmas de desarrollo tecnológico para resolver problemas antes no abordables. De esta manera, el edge AI desempeña un papel clave en la toma de decisiones en tiempo real en entornos con limitaciones

de potencia y espacio, lo que refuerza su relevancia para infraestructuras inteligentes e innovación industrial.

- **Reducción de las desigualdades (ODS 10):** El proyecto contribuye al desarrollo de tecnologías accesibles de reconocimiento de dígitos en sistemas embebidos de bajo consumo, lo que puede facilitar el acceso a herramientas digitales en contextos con recursos limitados. La evidencia indica que la brecha digital está asociada a desigualdades en acceso, competencias e infraestructura, y que reducirla exige tecnologías más inclusivas y accesibles. Además, distintos trabajos advierten que, sin intervenciones orientadas a la inclusión, la expansión de la inteligencia artificial puede ampliar las desigualdades existentes en comunidades desfavorecidas, mientras que estrategias de adopción accesible pueden ayudar a compartir sus beneficios de forma más equitativa.

Esquema de bloques del sistema

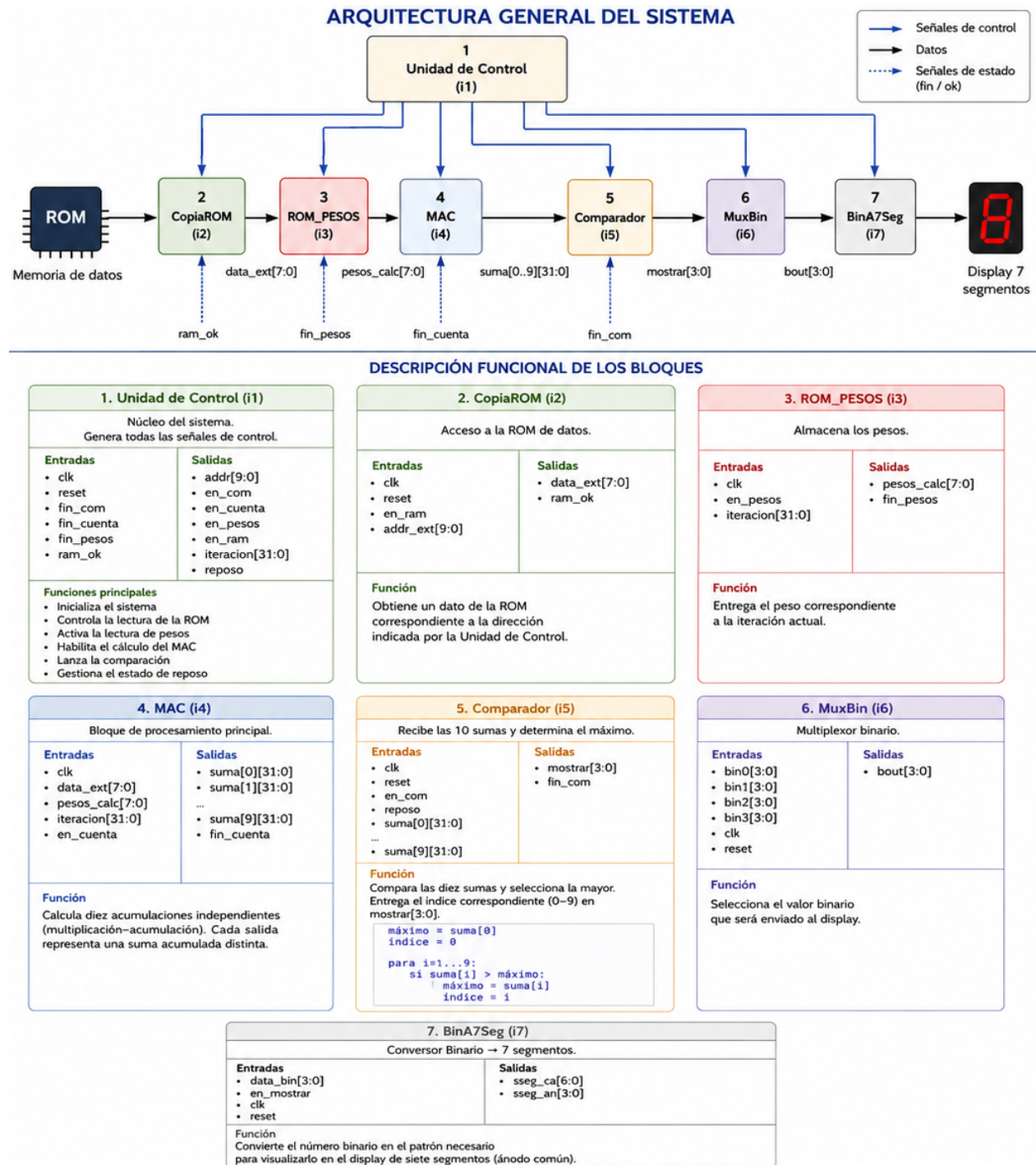


Figura 23: Esquema general del diseño en VHDL

Código VHDL Unidad de Control

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity UnidadControl is
6     port(
7         clk          : in  std_logic;
8         reset        : in  std_logic;
9         fin_cuenta   : in  std_logic;
10        en_cuenta    : out std_logic;
11        en_entrada   : out std_logic;
12        en_pesos     : out std_logic;
13        fin_entrada  : in  std_logic;
14        ram_ok       : in  std_logic;
15        en_ram       : out std_logic;
16        fin_pesos    : in  std_logic;
17        fin_com      : in  std_logic;
18        en_com       : out std_logic;
19        iteracion    : out integer;
20        reposol      : out std_logic;
21        en_mostrar   : out std_logic;
22        addr         : out unsigned(9 downto 0);
23        ptr_out      : out integer
24    );
25 end UnidadControl;
26
27 architecture Behavioral of UnidadControl is
28
29     type t_estados is (
30         Reposo, CopiarRom, Cargar_Datos,
31         Operacion, Incrementar, Comparar, Mostrar
32     );
33
34     signal estado_act, estado_sig : t_estados;
35     signal it_aux : integer range 1 to 5 := 1;
36     signal ptr : integer := 0;
37
38 begin
39
40     iteracion <= it_aux;
41     ptr_out <= ptr;
42
43     -- Registro de estado y control secuencial
44     Registro: process(clk, reset)
45     begin
46         if reset = '1' then
47             estado_act <= Reposo;
```

```
48         ptr          <= 0;
49         it_aux       <= 1;
50
51     elsif rising_edge(clk) then
52         estado_act <= estado_sig;
53
54         if estado_act = Incrementar then
55
56             if it_aux = 1 and ptr >= 783 then
57                 it_aux <= 2;
58                 ptr <= 0;
59
60             elsif it_aux = 2 and ptr >= 783 then
61                 it_aux <= 3;
62                 ptr <= 0;
63
64             elsif it_aux = 3 and ptr >= 127 then
65                 it_aux <= 4;
66                 ptr <= 0;
67
68             elsif it_aux = 4 and ptr >= 63 then
69                 ptr <= 0;
70
71             else
72                 ptr <= ptr + 1;
73
74             end if;
75
76         elsif estado_act = Reposo or estado_act = Comparar then
77             ptr <= 0;
78         end if;
79     end if;
80 end process;
81
82 -- Lógica de transición de estados
83 Transiciones: process(estado_act, fin_cuenta, ptr, fin_entrada, fin_pesos, ram_ok,
84                     it_aux)
85 begin
86     estado_sig <= estado_act;
87     addr       <= to_unsigned(ptr, 10);
88
89     case estado_act is
90
91         when Reposo =>
92             estado_sig <= CopiarRom;
93
94         when CopiarRom =>
95             if ram_ok = '1' then
96                 estado_sig <= Cargar_Datos;
97             end if;
```

```
98     when Cargar_Datos =>
99         if fin_pesos = '1' then
100             estado_sig <= Operacion;
101         end if;
102
103     when Operacion =>
104         if fin_cuenta = '1' then
105             estado_sig <= Incrementar;
106         end if;
107
108     when Incrementar =>
109         if it_aux = 1 and ptr >= 783 then
110             estado_sig <= Reposo;
111         elsif it_aux = 2 and ptr >= 783 then
112             estado_sig <= Reposo;
113         elsif it_aux = 3 and ptr >= 127 then
114             estado_sig <= Reposo;
115         elsif it_aux = 4 and ptr >= 63 then
116             estado_sig <= Comparar;
117         else
118             estado_sig <= Cargar_Datos;
119         end if;
120
121     when Comparar =>
122         if fin_com = '1' then
123             estado_sig <= Mostrar;
124         end if;
125
126     when Mostrar =>
127         estado_sig <= Reposo;
128
129     when others =>
130         estado_sig <= Reposo;
131
132     end case;
133 end process;
134
135 -- Lógica de salidas
136 Salidas: process(estado_act)
137 begin
138     en_cuenta <= '0';
139     en_entrada <= '0';
140     en_pesos <= '0';
141     en_com <= '0';
142     en_ram <= '0';
143     en_mostrar <= '0';
144     reposol <= '1';
145
146     case estado_act is
147
148     when Reposo =>
```

```
149         reposo1 <= '1';
150
151     when CopiarRom =>
152         en_ram <= '1';
153
154     when Cargar_Datos =>
155         en_pesos <= '1';
156
157     when Operacion =>
158         en_cuenta <= '1';
159
160     when Comparar =>
161         en_com <= '1';
162         reposo1 <= '0';
163
164     when Mostrar =>
165         en_mostrar <= '1';
166
167     when others => null;
168
169     end case;
170 end process;
171
172 end Behavioral;
```

Algoritmo 2: Unidad de control del acelerador de red neuronal en VHDL

Codigo VHDL bloque MAC

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4 use work.variables_pkg.all;
5
6 entity MAC is
7     port(
8         clk          : in  std_logic;
9         en_cuenta    : in  std_logic;
10        ptr          : in  integer;
11        data_ext      : in  unsigned(7 downto 0);
12        pesos_calc    : in  pesos_neurona;
13        pesos_calc2   : in  pesos_neur12;
14        pesos_calc3   : in  pesos_f;
15        fin_cuenta    : out std_logic;
16        iteracion     : in  integer;
17        suma          : out t_vector3
18    );
19 end MAC;
20
21 architecture Behavioral of MAC is
22
23     signal acumulador_total1 : t_vector := (others => (others => '0'));
24     signal acumulador_total2 : t_vector2 := (others => (others => '0'));
25     signal acumulador_total3 : t_vector3 := (others => (others => '0'));
26
27     signal suma_tem : suma_temp := (others => (others => '0'));
28     signal suma_tem2 : suma_temp2 := (others => (others => '0'));
29
30     signal r_pixel : signed(8 downto 0) := (others => '0');
31     signal r_peso1 : signed(7 downto 0) := (others => '0');
32     signal r_mult1 : signed(16 downto 0) := (others => '0');
33
34     signal idx_neurona : integer range 0 to 127 := 0;
35     signal idx_relu : integer range 0 to 127 := 0;
36     signal cuenta_pixeles : integer range 0 to 1023 := 0;
37
38     type t_estado_mac is (Idle, Reset_Acum, Wait_ROM, Cargar_Regs, Multiplicar, Acumular,
39         Verificar_Fin, Proc_Bias, Proc_ReLU, Hecho);
40     signal estado_int : t_estado_mac := Idle;
41
42 begin
43     process(clk)
44     begin
45         if rising_edge(clk) then
```

```
47     case estado_int is
48
49         when Idle =>
50             fin_cuenta <= '0';
51             if en_cuenta = '1' then
52                 if ptr = 0 then
53                     cuenta_pixeles <= 0;
54                 end if;
55
56                 if ptr = 0 and iteracion = 1 then
57                     idx_neurona <= 0;
58                     estado_int <= Reset_Acum;
59                 else
60                     idx_neurona <= 0;
61                     estado_int <= Wait_ROM;
62                 end if;
63             end if;
64
65         when Reset_Acum =>
66             if idx_neurona <= 9 then
67                 acumulador_total3(idx_neurona) <= (others => '0');
68             end if;
69             if idx_neurona <= 63 then
70                 acumulador_total2(idx_neurona) <= (others => '0');
71             end if;
72             acumulador_total1(idx_neurona) <= (others => '0');
73
74             if idx_neurona = 127 then
75                 idx_neurona <= 0;
76                 estado_int <= Cargar_Regs;
77             else
78                 idx_neurona <= idx_neurona + 1;
79             end if;
80
81         when Wait_ROM =>
82             estado_int <= Cargar_Regs;
83
84         when Cargar_Regs =>
85             case iteracion is
86                 when 1 =>
87                     r_pixel <= signed('0' & data_ext);
88                     r_pesos1 <= pesos_calc(idx_neurona);
89                 when 2 =>
90                     r_pixel <= signed('0' & data_ext);
91                     r_pesos1 <= pesos_calc(idx_neurona + 90);
92                 when 3 =>
93                     if cuenta_pixeles <= 127 then
94                         r_pixel <= signed('0' & suma_tem(cuenta_pixeles));
95                     else
96                         r_pixel <= (others => '0');
97                     end if;
```

```

98         r_pesol1 <= pesos_calc2(idx_neurona);
99     when 4 =>
100         if cuenta_pixeles <= 63 then
101             r_pixel <= signed('0' & suma_tem2(cuenta_pixeles));
102         else
103             r_pixel <= (others => '0');
104         end if;
105         r_pesol1 <= pesos_calc3(idx_neurona);
106     when others => null;
107 end case;
108 estado_int <= Multiplicar;
109
110 when Multiplicar =>
111     r_mult1 <= r_pixel * r_pesol1;
112     estado_int <= Acumular;
113
114 when Acumular =>
115     case iteracion is
116     when 1 => acumulador_total1(idx_neurona) <= acumulador_total1(
117         idx_neurona) + resize(r_mult1, 32);
118     when 2 => acumulador_total1(idx_neurona + 90) <= acumulador_total1(
119         idx_neurona + 90) + resize(r_mult1, 32);
120     when 3 => acumulador_total2(idx_neurona) <= acumulador_total2(
121         idx_neurona) + resize(r_mult1, 32);
122     when 4 => acumulador_total3(idx_neurona) <= acumulador_total3(
123         idx_neurona) + resize(r_mult1, 32);
124     when others => null;
125     end case;
126     estado_int <= Verificar_Fin;
127
128 when others =>
129     estado_int <= Idle;
130 end case;
131 end if;
132 end process;
133 end Behavioral;

```

Algoritmo 3: Implementación del módulo MAC en VHDL