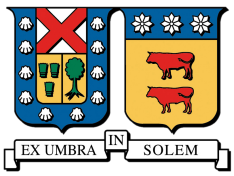


**UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA**

DEPARTAMENTO DE ELECTRÓNICA E INFORMÁTICA

# DISEÑO E IMPLEMENTACIÓN DEL ALGORITMO K-NEAREST NEIGHBORS EN FPGA PARA CLASIFICACIÓN BINARIA

Tesis de Grado presentada por

**Diego Hernan Hidalgo Contreras**

como requisito parcial para optar al título de

**Ingeniería de Ejecución en Control e Instrumentación Industrial**

Director de Tesis
Dr. Jorge Portilla Gómez.

Concepción, 2025.

TíTULO DE LA TESIS:

# DISEÑO E IMPLEMENTACIÓN DEL ALGORITMO K-NEAREST NEIGHBORS EN FPGA PARA CLASIFICACIÓN BINARIA

AUTOR:

**Diego Hernan Hidalgo Contreras**

TRABAJO DE TESIS, presentado en cumplimiento parcial de los requisitos para el título de **Ingeniería de Ejecución en Control e Instrumentación Industrial** de la Universidad Técnica Federico Santa María.

**Director de Tesis**
Dr. Jorge Portilla Gómez. _____

**Codirector de Tesis**
Dr. AAA _____

**Examinador Interno**
Dr. BBB _____

**Examinador Externo**
Dr. BBB _____

Concepción, 2025.

# AGRADECIMIENTOS

A mi abuelo.

# CONTENIDO

# ÍNDICE DE FIGURAS

# ÍNDICE DE TABLAS

# ÍNDICE DE CÓDIGOS

# GLOSARIO

**k-NN (k-Nearest Neighbors):** Algoritmo que clasifica un dato nuevo según las clases de sus $k$ vecinos más cercanos.

**FPGA (Field Programmable Gate Array):** Chip reconfigurable que permite implementar circuitos digitales personalizados.

**IoT (Internet of Things):** Conjunto de dispositivos conectados a internet que intercambian datos y permiten automatizar procesos.

**CPU (Central Processing Unit):** Unidad central de procesamiento que ejecuta instrucciones y controla el funcionamiento de un sistema.

**HDL (Hardware Description Language):** Lenguaje usado para describir y diseñar circuitos digitales.

**VHDL (VHSIC Hardware Description Language):** Lenguaje de descripción de hardware de circuitos integrados de muy alta velocidad.

**RTL (Register Transfer Level):** Nivel de descripción de hardware que muestra cómo los datos se mueven entre registros y qué operaciones lógicas se aplican.

**LUTs (Look-Up Tables):** Tablas de búsqueda usadas en FPGAs para implementar funciones lógicas de forma eficiente.

**FFs (Flip-Flops):** Celdas de almacenamiento en circuitos digitales que guardan un bit y se actualizan con la señal de reloj.

**BRAMs (Block RAMs):** Memorias internas de la FPGA organizadas en bloques, usadas para almacenar datos o parámetros.

**UART (Universal Asynchronous Receiver-Transmitter):** Módulo que permite comunicación serial asíncrona enviando y recibiendo datos.

**GPU (Graphics Processing Unit):** Procesador especializado en operaciones paralelas, usado principalmente para gráficos y tareas de alto rendimiento.

**ASIC (Application-Specific Integrated Circuit):** Circuito integrado diseñado para una aplicación o tarea específica.

**TPU (Tensor Processing Unit):** Procesador especializado creado para acelerar cálculos de redes neuronales y aprendizaje automático.

**DSP (Digital Signal Processor):** Procesador optimizado para realizar operaciones rápidas sobre señales digitales, como filtrado y transformadas.

**SoC (System on Chip):** Circuito integrado que incluye procesador, memoria y periféricos en un solo chip.

**CLBs (Configurable Logic Blocks):** Bloques lógicos programables en una FPGA que permiten implementar funciones digitales personalizadas.

**SLICEM (Slice with Memory):** Tipo de bloque dentro de una FPGA que, además de lógica, puede configurarse para funcionar como memoria distribuida o registros de desplazamiento.

**SLICEL (Slice Logic):** Tipo de bloque dentro de una FPGA que contiene LUTs y flip-flops para implementar lógica, pero sin funciones de memoria adicionales.

**IP (Intellectual Property Core):** Bloque o módulo reutilizable de hardware que implementa una función específica dentro de un diseño.

**FSM (Finite State Machine):** Modelo de control que cambia entre un conjunto finito de estados según entradas y condiciones.

**SPI (Serial Peripheral Interface):** Protocolo de comunicación serial síncrona usado para conectar microcontroladores y periféricos.

**PCIe (Peripheral Component Interconnect Express):** Interfaz de alta velocidad que conecta dispositivos periféricos al procesador o sistema principal.

**AXI (Advanced eXtensible Interface):** Interfaz de bus utilizada en sistemas SoC y FPGAs para transferir datos de forma eficiente entre módulos.

**FFT (Fast Fourier Transform):** Algoritmo que convierte una señal del dominio del tiempo al dominio de la frecuencia de forma rápida y eficiente.

**PRL (Power Ratio Low):** Porcentaje de energía espectral concentrada en la banda de baja frecuencia de la señal.

**PRH (Power Ratio High):** Porcentaje de energía espectral concentrada en la banda de alta frecuencia de la señal.

**BROM (Block Read-Only Memory):** Memoria interna de solo lectura en la FPGA, utilizada para almacenar datos fijos o constantes que no requieren modificación durante la operación.

**ILA (Integrated Logic Analyzer):** Herramienta interna de depuración en FPGAs que permite capturar y visualizar señales en tiempo real durante la ejecución del diseño.

**WNS (Worst Negative Slack):** Valor de slack negativo más crítico en el diseño; indica cuánto se supera el tiempo límite de llegada de una señal respecto al tiempo requerido.

**TNS (Total Negative Slack):** Suma acumulada de todos los valores de slack negativos en el diseño; refleja el impacto global del incumplimiento temporal.

**THS (Total Hold Slack):** Suma total del margen temporal en todas las comprobaciones de retención (*hold*); valores negativos indican posibles violaciones de retención.

**WHS (Worst Hold Slack):** Menor slack de retención (*hold*) detectado; si es negativo, indica una violación crítica de retención.

**WBSS (Worst Pulse Width Slack):** Margen mínimo disponible respecto al ancho de pulso requerido para una señal de reloj; valores negativos indican riesgo de fallos de temporización por pulsos estrechos.

**TPWS (Total Pulse Width Slack):** Suma acumulada de márgenes de ancho de pulso; valores negativos reflejan cuántas comprobaciones de ancho de pulso no cumplen con el requisito mínimo.

# RESUMEN

# ABSTRACT

Capítulo 1

# INTRODUCCIÓN

Este documento describe el diseño e implementación del algoritmo *k-Nearest Neighbors* (k-NN), optimizado para su ejecución en una matriz de puertas programables en campo (*Field-Programmable Gate Array*, FPGA), con un enfoque orientado a tareas de clasificación binaria en tiempo real y estructurado bajo una arquitectura *host–device*. En este esquema, una computadora de escritorio actúa como *host*, encargada de la configuración, el control operativo y la visualización de resultados, mientras que la FPGA cumple el rol de *device*, asumiendo el procesamiento intensivo y especializado. El sistema desarrollado se basa en una aplicación implementada en Python que realiza el preprocesamiento de los datos y la gestión de la comunicación con la FPGA, mientras que la lógica principal del clasificador fue modelada en SystemVerilog, adaptándose a las restricciones estructurales y temporales del dispositivo reconfigurable, con el objetivo de lograr una inferencia binaria eficiente, determinista y reproducible.

Este capítulo presenta una introducción general al trabajo, exponiendo la motivación que origina la investigación y los antecedentes técnicos que contextualizan el problema. A partir de este marco, se plantea la hipótesis de trabajo y se formulan los objetivos generales y específicos que orientan el desarrollo del proyecto. Se incluyen además las justificaciones que sustentan el enfoque propuesto, los alcances y limitaciones del sistema diseñado, y las principales contribuciones previstas. Finalmente, se describe la estructura del documento, proporcionando una visión global de los capítulos que conforman la tesis y la forma en que se abordan los aspectos teóricos, metodológicos, arquitectónicos y experimentales implicados.

## 1.1. Motivación y Contexto

La era digital contemporánea está marcada por un crecimiento exponencial en la generación de datos, impulsado por la proliferación de sensores, dispositivos inteligentes interconectados (*Internet of Things*, o IoT) y diversos sistemas de información distribuidos [1]. Esta expansión ha dado lugar a volúmenes de datos sin precedentes, cuya velocidad y magnitud demandan métodos de procesamiento cada vez más sofisticados, especialmente en contextos industriales, científicos y tecnológicos, donde se impone el reto de transformar grandes cantidades de información en conocimiento útil, accesible en tiempo real y capaz de respaldar decisiones bien fundamentadas [2].

Frente a esta realidad, las arquitecturas tradicionales basadas en el escalamiento progresivo de procesadores de propósito general, conocidos como CPUs (*Central Processing Units*), han comenzado a mostrar sus limitaciones [3]. Durante décadas, este modelo se sostuvo gracias a la vigencia de la Ley de Moore, que predecía un incremento sostenido en la densidad de transistores y en el rendimiento computacional. Sin embargo, esta tendencia se ha desacelerado debido a barreras físicas, térmicas y

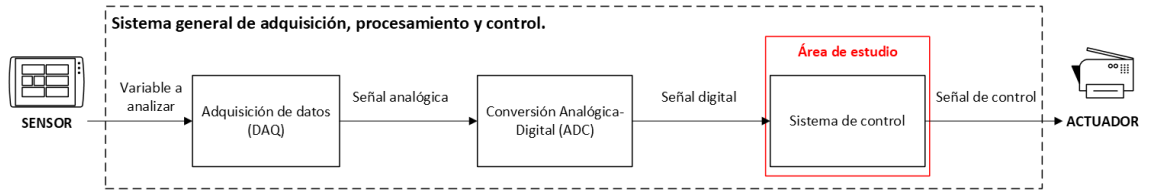


Fig. 1.1: Sistema general de adquisición, procesamiento y control. El área destacada en rojo corresponde a la etapa de procesamiento lógico y toma de decisiones, elaboración propia.

económicas que dificultan la miniaturización continua de los circuitos integrados [4]. En consecuencia, la simple actualización de CPUs ya no resulta suficiente para atender las crecientes demandas de rendimiento y eficiencia que plantea el entorno tecnológico actual.

En respuesta a este escenario, ha surgido un enfoque centrado en arquitecturas especializadas de hardware, concebidas para maximizar la eficiencia en tareas específicas mediante la adaptación estructural del sistema al problema que se desea resolver [3]. Este tipo de diseño contrasta con su contraparte de propósito general al priorizar la optimización del consumo energético, el uso eficiente de recursos y un comportamiento altamente predecible; cualidades esenciales en aplicaciones sensibles a la latencia o sujetas a restricciones operativas estrictas. Su desarrollo se orienta a cargas computacionales concretas, alcanzando niveles de rendimiento superiores a los que pueden ofrecer las plataformas convencionales.

Entre las distintas arquitecturas especializadas de hardware, las FPGAs se destacan por su capacidad para implementar secuencias de control precisas y reproducibles directamente a nivel de hardware. A diferencia de los procesadores convencionales, cuya ejecución depende de instrucciones almacenadas y ciclos de lectura-escritura, las FPGAs permiten diseñar circuitos con rutas de ejecución explícitas, lo que las hace especialmente adecuadas para aplicaciones deterministas como sistemas embebidos de tiempo real o lógica de decisión secuencial [5]. Además de su capacidad secuencial, sobresalen por su potencial de paralelismo, habilitando procesos concurrentes y algoritmos complejos con baja latencia [6]. Su lógica programable admite la reconfiguración mediante lenguajes de descripción de hardware (*hardware description language*, HDL) como Verilog o VHDL, permite integrar módulos especializados y administrar recursos internos como LUTs (*Look-Up Tables*), FFs (*Flip-Flops*), BRAMs (*Block RAMs*) y DSPs (*Digital Signal Processors*), garantizando sincronización precisa a través de redes de reloj dedicadas. Estas características convierten a las FPGAs en una plataforma robusta y versátil para el desarrollo de soluciones embebidas con requisitos estrictos de control y procesamiento paralelo.

En el ámbito de los sistemas de adquisición, procesamiento y control, estas capacidades se proyectan como un recurso valioso. En este tipo de sistemas, los datos se obtienen desde sensores, se procesan mediante bloques lógicos y se generan señales de salida que actúan sobre el entorno. La Fig. 1.1 ilustra este flujo y resalta, en el área sombreada en rojo, la etapa dedicada a interpretar los datos digitales y generar decisiones lógicas que retroalimenten el sistema. Su correcta implementación es clave para garantizar un funcionamiento confiable y adaptable en arquitecturas orientadas a control y procesamiento en tiempo real [7].

Sobre este esquema general, distintos trabajos han propuesto mecanismos para hacer más flexible la etapa de control. Una de las estrategias más destacadas es mantener la lógica de control independiente de los sensores o del entorno físico, de modo que pueda reutilizarse en distintos contextos sin modificar la arquitectura interna. Esta idea ha favorecido el desarrollo de soluciones con procesamiento embebido capaces de analizar los datos directamente en el dispositivo [8]. Además,

acercar la inferencia al punto donde se capturan los datos se considera una forma efectiva de reducir la latencia y mejorar el rendimiento, y las FPGAs, por su naturaleza reconfigurable, ofrecen una base sólida para implementar esta lógica sin depender de sistemas de apoyo externos [9].

Aun así, surgen nuevas exigencias: la toma de decisiones requiere mayor flexibilidad para responder a situaciones cambiantes. Los esquemas de control basados exclusivamente en reglas fijas pueden resultar limitados ante entornos dinámicos o inciertos. En este contexto, técnicas de inferencia y aprendizaje automático emergen como alternativas sólidas, al permitir que el sistema identifique patrones y aprenda de datos históricos y experiencias previas. Sin embargo, el hardware por sí solo no basta para alcanzar este nivel de adaptación; es necesario incorporar algoritmos que implementen dicha lógica inteligente. Entre estas técnicas, la integración de clasificadores binarios directamente en el dispositivo se perfila como una opción prometedora, particularmente cuando las decisiones se reducen a dos clases discretas, un enfoque de gran interés en entornos *safety-critical* [10].

En este marco, el algoritmo k-NN se presenta como una alternativa sencilla y eficaz para implementar clasificadores en FPGA, ya que compara cada muestra con un conjunto de ejemplos etiquetados e identifica las $k$ instancias más cercanas según una métrica de distancia y un algoritmo de ordenamiento [11]. Al no requerir fases de entrenamiento complejas y basarse en operaciones aritméticas simples, resulta especialmente adecuado para dispositivos reconfigurables, donde puede organizarse en etapas bien definidas que favorecen el diseño modular y la verificación funcional. Para llevar este tipo de algoritmos al hardware se emplea con frecuencia una arquitectura *host–device*, en la que el coprocesador reconfigurable asume el procesamiento intensivo, mientras que un sistema anfitrión se encarga únicamente del control y la configuración externa, como la carga de parámetros o la lectura de resultados. Este reparto de tareas permite concentrar la lógica computacional más pesada directamente en el chip, optimizando recursos y simplificando el diseño general del sistema [6].

Diversos estudios han demostrado la viabilidad de integrar el k-NN en este tipo de plataformas, destacando su potencial para sistemas embebidos. Por ejemplo, en [12] se describe una arquitectura orientada a reducir el consumo energético mediante un uso ajustado de elementos lógicos, mientras que en [13] se propone una variante que utiliza aritmética en línea para acelerar el cálculo de distancias. Estos trabajos muestran un interés constante por adaptar el algoritmo a entornos con restricciones de consumo y temporización; sin embargo, la mayoría aborda escenarios muy específicos, lo que limita su alcance general. De allí surge la necesidad de propuestas más versátiles y compactas que, además de mantener la precisión, aprovechen mejor las capacidades del dispositivo, conserven un comportamiento determinista y puedan integrarse con facilidad en arquitecturas de control embebido. Esta perspectiva abre la puerta a desarrollos con impacto directo en aplicaciones críticas de tiempo real y constituye el motor que impulsa la presente investigación.

## 1.2. Planteamiento del Problema

Considerando lo anterior, aunque se han logrado avances relevantes en la adaptación del algoritmo k-NN a FPGAs, su despliegue en sistemas embebidos sigue enfrentando retos significativos desde el punto de vista del diseño digital. La ejecución de operaciones de comparación, ordenamiento y decisión bajo restricciones temporales estrictas exige una arquitectura capaz de mantener la coherencia del sistema y su escalabilidad sin comprometer la integridad del flujo de datos.

Dentro de este proceso, las etapas de cálculo de distancias y selección de los $k$ elementos más cercanos constituyen núcleos críticos cuya implementación debe optimizarse para equilibrar el uso de recursos y asegurar una respuesta determinista. Adaptar esta lógica al dispositivo real implica atender con precisión los ciclos de reloj, las restricciones de temporización y los límites físicos propios

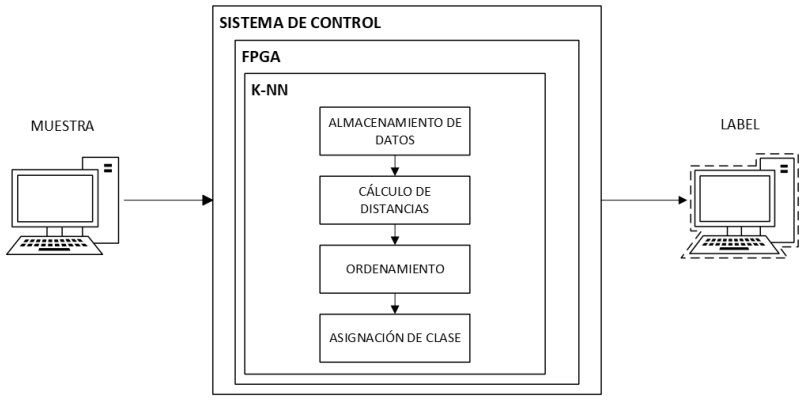


Fig. 1.2: Vista general del sistema k-NN implementado en FPGA y su entorno de control, elaboración propia.

de la plataforma.

A estas exigencias se suma la pronunciada curva de aprendizaje asociada al diseño digital, que requiere un conocimiento detallado de la arquitectura, la sincronización y la validación funcional. Traducir un modelo algorítmico a circuitos eficientes y verificables implica un esfuerzo considerable, lo que evidencia la importancia de contar con metodologías claras y soluciones reutilizables que simplifiquen el desarrollo y su posterior integración en sistemas reales.

Por último, cuando se adopta un esquema *host–device* para distribuir tareas entre un procesador de control y la lógica dedicada a la clasificación, es fundamental que la integración preserve un comportamiento determinista y reproducible frente a distintos escenarios de entrada, manteniendo al mismo tiempo una arquitectura modular, escalable y fácil de verificar.

## 1.3. Propuesta de Solución

Como respuesta a estas necesidades técnicas, se plantea el desarrollo de una implementación funcional del algoritmo k-NN para clasificación binaria, integrando sus etapas internas en una arquitectura digital organizada en módulos secuenciales. El diseño busca simplicidad lógica, precisión en la toma de decisiones y compatibilidad con plataformas embebidas reconfigurables, de manera que la solución resulte verificable, escalable y adaptable a futuros requerimientos. La Fig. 1.2 muestra el flujo general del sistema, desde la entrada de muestras hasta la emisión del resultado clasificado, destacando la relación entre la lógica implementada y las tareas de soporte externo.

La propuesta se apoya en una interacción directa entre la FPGA y una interfaz desarrollada en Python. Mientras la FPGA ejecuta las operaciones críticas de clasificación, la interfaz gestiona tareas externas como la configuración de parámetros, la visualización de resultados y la supervisión del flujo de datos. Esta división de funciones permite concentrar los recursos del dispositivo en el procesamiento especializado sin recurrir a procesadores embebidos adicionales.

El procesamiento de datos se realiza mediante un flujo secuencial que permite operar sobre cada muestra a medida que ingresa, sin necesidad de disponer de todo el conjunto desde el inicio. Para ello se aplican técnicas eficientes para el cálculo de distancias, la selección de los vecinos más cercanos y la emisión de la decisión final, privilegiando estructuras reutilizables y de baja latencia. Esta aproximación asegura el cumplimiento de los requisitos funcionales sin comprometer la escalabilidad ni la portabilidad del sistema.

## 1.4. Objetivos

El objetivo general de este trabajo es diseñar e implementar en FPGA una arquitectura modular para clasificación binaria basada en el algoritmo k-NN. El sistema está pensado para operar con preprocesamiento externo, de modo que la FPGA se encargue únicamente de ejecutar las operaciones críticas del clasificador mediante lógica descrita en HDL.

Los objetivos específicos se listan a continuación:

1. Desarrollar un módulo en FPGA para el cálculo de distancias entre la muestra de entrada y los vectores de entrenamiento, empleando la métrica Manhattan como base para una implementación aritmética optimizada.
2. Implementar un bloque funcional de ordenamiento con baja complejidad computacional, capaz de identificar los $k$ vecinos más cercanos sin afectar la latencia del sistema.
3. Diseñar una interfaz de comunicación externa mediante Python que permita la transferencia dinámica de datos y la configuración de parámetros del sistema, bajo una estructura tipo *host–device*.
4. Validar la arquitectura propuesta mediante pruebas experimentales con conjuntos de datos etiquetados, evaluando métricas como latencia, uso de recursos lógicos, precisión de clasificación y rendimiento general bajo condiciones operativas representativas.

## 1.5. Contribuciones

Este trabajo presenta y valida experimentalmente un clasificador binario implementado íntegramente en FPGA, aportando una base sólida para la etapa de control dentro de un sistema de adquisición. Su desempeño es evaluado en un entorno real y comparado con una implementación equivalente en software, lo que permite identificar ventajas y limitaciones en términos de latencia, utilización de recursos lógicos y operación continua bajo un flujo secuencial de datos. Este resultado se proyecta como un punto de partida útil para futuros desarrollos orientados a optimizar y ampliar sistemas de control embebidos en plataformas reconfigurables.

En el ámbito del diseño digital, se desarrolla un bloque de ordenamiento optimizado con baja complejidad computacional, elemento clave para la identificación de los $k$ vecinos más cercanos en menor tiempo. Esta optimización es especialmente relevante en plataformas con recursos limitados, ya que impacta directamente en el rendimiento global. La estrategia empleada preserva la precisión de la clasificación sin comprometer la velocidad de inferencia, integrando un flujo secuencial que evita la necesidad de disponer del conjunto completo de datos antes de iniciar el proceso.

Desde el punto de vista metodológico, se lleva a cabo una caracterización exhaustiva del sistema bajo distintas configuraciones. Se analiza cómo influyen variables como el tamaño del conjunto de entrenamiento y el valor de $k$ en métricas clave como el consumo de LUTs, FFs y BRAMs. Este estudio permite identificar umbrales críticos y establecer lineamientos prácticos para dimensionar eficientemente el sistema, generando conocimiento aplicable a otros diseños de clasificación con restricciones similares.

Finalmente, se propone una arquitectura modular y escalable que puede reconfigurarse o ampliarse sin alterar el funcionamiento general. Su compatibilidad con herramientas externas, como interfaces gráficas desarrolladas en Python y flujos de carga automática de memoria, la convierte

en una plataforma sólida para futuros avances en clasificación embebida, validación de algoritmos y aplicaciones educativas en hardware reconfigurable.

## 1.6. Alcances y Limitaciones

Este estudio se desarrolla bajo un conjunto de restricciones que condicionan tanto el diseño como su aplicabilidad en escenarios reales. Aunque la arquitectura propuesta es parametrizable y adaptable, no incluye procesos de adquisición de señales ni etapas de preprocesamiento, las cuales se identifican como posibles líneas de trabajo futuro para una solución más integral.

La implementación se realiza sobre la tarjeta Nexys A7-100T, que integra una FPGA Xilinx Artix-7 (modelo XC7A100T-1CSG324C). Las decisiones de diseño responden a las capacidades y limitaciones de esta plataforma, y el sistema se centra exclusivamente en problemas de clasificación binaria, dejando fuera escenarios multiclase y técnicas avanzadas de reducción de dimensionalidad.

Para la aplicación desarrollada en Python, se establece un intercambio de datos con la FPGA a través de la comunicación UART (*Universal Asynchronous Receiver-Transmitter*).

El cálculo de distancias emplea la métrica Manhattan por su bajo costo computacional, y se analizan dos métodos de selección de vecinos: el ordenamiento burbuja, sencillo pero con mayor latencia, y un esquema de inserción optimizado que mantiene dinámicamente los $k$ valores mínimos. Esta comparación permite evaluar su efecto sobre la complejidad y el tiempo de respuesta.

La lógica se implementa en SystemVerilog, priorizando claridad estructural, portabilidad y reutilización de módulos. Todas las operaciones se realizan con números enteros, lo que contribuye a reducir la complejidad del diseño y a aprovechar de forma eficiente los recursos disponibles.

## 1.7. Organización de la Tesis

Esta tesis se estructura en cinco capítulos que abarcan los aspectos teóricos, metodológicos, técnicos y experimentales del trabajo:

- **Capítulo 1 – Introducción:** Presenta el contexto del estudio, la motivación, el problema, los objetivos y los alcances de la investigación.
- **Capítulo 2 – Marco Teórico:** Revisa los conceptos clave, incluyendo el algoritmo k-NN, métricas de distancia, métodos de ordenamiento y características de las FPGAs.
- **Capítulo 3 – Implementación:** Describe la metodología, herramientas y la arquitectura desarrollada en FPGA, detallando los módulos de cálculo, ordenamiento y clasificación.
- **Capítulo 4 – Resultados:** Presenta la validación experimental, evaluando latencia, rendimiento y uso de recursos.
- **Capítulo 5 – Conclusiones:** Expone los hallazgos principales y propone líneas de trabajo futuro.

Capítulo 2

# MARCO TEÓRICO

En este capítulo se presentan los fundamentos conceptuales y tecnológicos que respaldan el desarrollo de un sistema de clasificación binaria basado en el algoritmo k-NN, implementado sobre una plataforma FPGA. El análisis comienza con una revisión del panorama actual de la computación, marcado por las limitaciones estructurales de las arquitecturas de propósito general y el auge de soluciones especializadas. En este contexto, se examina el rol de las FPGAs como plataformas que facilitan el diseño de sistemas lógicos a medida, ajustables tanto a nivel estructural como funcional, según los requerimientos específicos de cada aplicación. Esta capacidad de adaptación resulta especialmente valiosa en escenarios donde las soluciones convencionales no permiten optimizar simultáneamente el control del tiempo de ejecución y la distribución de recursos.

Sobre esta base, se introduce el algoritmo k-NN como técnica de clasificación supervisada, detallando sus componentes clave para la implementación en hardware: cálculo de distancias, selección de vecinos y decisión de clase. Se analizan las métricas empleadas para establecer similitud entre muestras, así como los algoritmos de ordenamiento requeridos para seleccionar los $k$ vecinos más próximos. Además, se describen los criterios de evaluación funcional y arquitectónica del sistema, incorporando herramientas de simulación, síntesis y validación. Finalmente, se incluye una revisión del estado del arte que sitúa esta propuesta dentro del panorama actual de implementaciones del algoritmo k-NN sobre hardware reconfigurable.

## 2.1. Computación Especializada

La computación especializada se centra en el diseño y uso de arquitecturas concebidas para ejecutar tareas concretas con altos niveles de rendimiento, eficiencia energética y control. A diferencia de los sistemas de propósito general, estas plataformas priorizan la optimización de factores críticos como la latencia, el paralelismo y la adecuación a requisitos funcionales específicos [14]. Este enfoque adquiere especial relevancia ante el crecimiento exponencial de datos y la demanda de procesamiento en tiempo real, contextos donde las arquitecturas tradicionales comienzan a mostrar limitaciones estructurales y de escalabilidad.

En este amplio ecosistema, las FPGA representan una opción fundamental, pero no la única. Existen otras tecnologías que han demostrado su eficacia en ámbitos particulares: la GPU (*Graphics Processing Unit*), orientada al paralelismo masivo en tareas como simulaciones científicas y redes neuronales profundas, aunque limitada por un consumo energético elevado y rigidez arquitectónica [15]; el ASIC (*Application-Specific Integrated Circuit*), con un rendimiento sobresaliente y eficiencia energética inigualable al estar diseñado para una única función, aunque sin capacidad de reconfiguración y con altos costos de producción [16]; la TPU (*Tensor Processing Unit*), optimizada

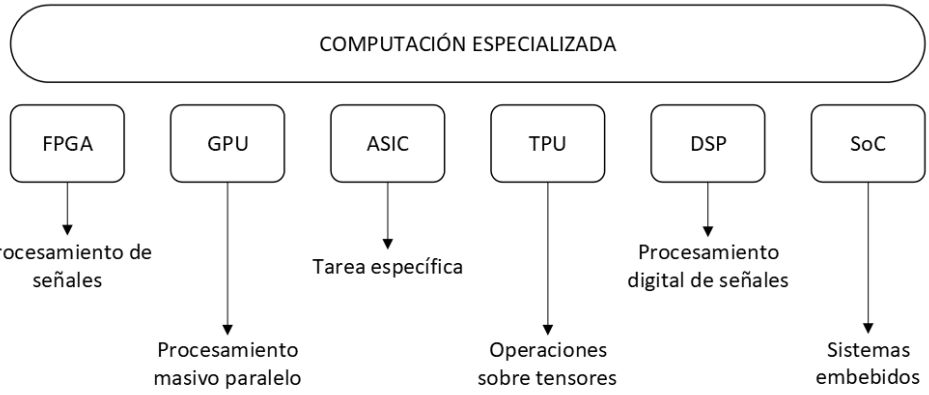


Fig. 2.1: Resumen funcional de las principales tecnologías de computación especializada y sus dominios de aplicación preferente. Elaboración propia a partir de [14] y [15].

para operaciones matriciales en aprendizaje profundo, destacada por su velocidad en flujos de datos estructurados pero con menor versatilidad [17]; el DSP (*Digital Signal Processor*), ideal para procesamiento continuo de señales gracias a su baja latencia y consumo reducido en aplicaciones como audio, filtrado y control de motores [18]; y el SoC (*System on Chip*), que integra múltiples bloques funcionales en un solo dispositivo, aportando soluciones compactas y versátiles para sistemas embebidos y plataformas IoT [19]. La Fig. 2.1 resume visualmente estas tecnologías y los dominios donde suelen desplegarse con mayor eficacia.

Considerando estas alternativas, la FPGA se adopta en este trabajo no solo por su capacidad de adaptación arquitectónica, sino también por su habilidad para unificar en una misma plataforma la eficiencia del hardware especializado y la flexibilidad del software programable. Esta sinergia permite diseñar aceleradores dedicados que pueden reconfigurarse para distintos escenarios, obteniendo así un balance óptimo entre rendimiento, escalabilidad y capacidad de evolución. Tales características la convierten en la base técnica más adecuada para el desarrollo del sistema de clasificación binaria propuesto.

## 2.1.1. Tecnología FPGA

Más allá de su rol dentro del ecosistema de cómputo especializado, resulta necesario comprender la estructura y las herramientas propias de una FPGA para explotar todo su potencial en el diseño de sistemas a medida. A continuación, se detalla su arquitectura interna y la manera en que sus recursos se organizan físicamente, se describe el flujo de desarrollo típico que permite pasar de una especificación lógica a un diseño implementado en el dispositivo, y finalmente se analiza su utilización como coprocesador especializado dentro de arquitecturas heterogéneas.

### 2.1.1.1. Arquitectura interna

Internamente, la FPGA se organiza como una matriz de recursos configurables. En esta matriz se integran bloques lógicos programables, memorias internas y unidades aritméticas dedicadas, todos conectados mediante redes de ruteo flexibles que permiten implementar diseños paralelos de forma eficiente [20]. Esta disposición posibilita ajustar tanto la organización física como el comportamiento temporal, adaptándose a los requisitos específicos de cada aplicación.

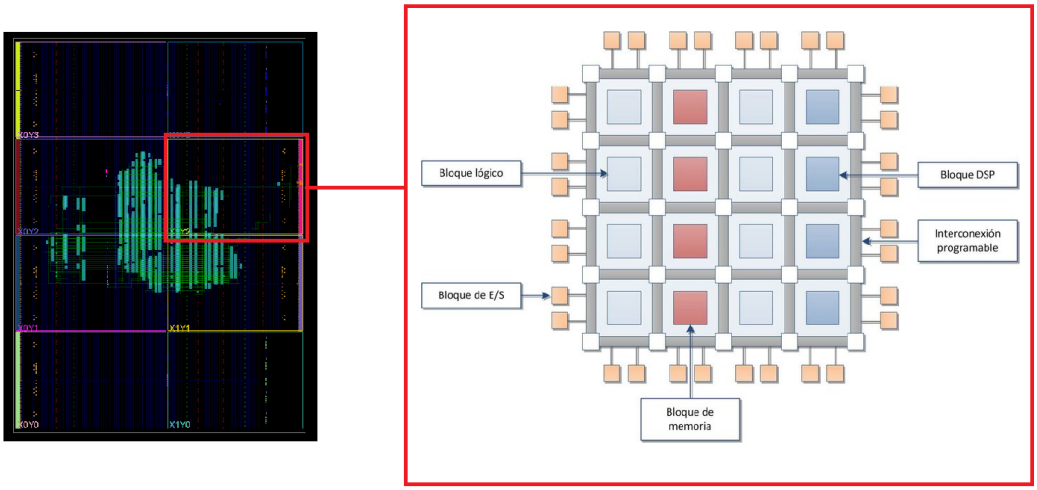


Fig. 2.2: Vista global (izquierda) y detallada (derecha) de la arquitectura interna de una FPGA. La imagen integra la distribución física en *clock regions* (extraída de Vivado Suite 2025) y el esquema interno típico de una región (adaptado de [**?**]).

Desde el punto de vista físico, el núcleo de una FPGA se divide en *clock regions*, rectángulos que actúan como unidades autónomas de sincronización. Esta disposición permite distribuir la señal de reloj de forma equilibrada mediante búferes dedicados que la propagan a nivel global, regional y local, garantizando sincronía en todo el dispositivo. Tal como se muestra en la Fig. 2.2, estas regiones se disponen en una malla de columnas y filas, cuya cantidad y distribución varían según el modelo de FPGA, optimizando la propagación del reloj y el aprovechamiento interno de recursos [21].

La misma figura muestra también el detalle interno de una *clock regions*, donde se distribuyen de manera homogénea los principales recursos funcionales. Cada región contiene una matriz de CLBs (*Configurable Logic Blocks*), BRAMs y DSPs, conectados a través de una red de ruteo jerárquica que enlaza recursos locales, intermedios y globales. Alrededor del núcleo se ubican los bloques de entrada y salida, que actúan como interfaz eléctrica con el entorno, asegurando compatibilidad con diversos estándares de señal y niveles de tensión.

En cada CLB se agrupan *slices* que combinan LUTs para funciones lógicas y registros FFs para sincronización secuencial. Existen dos tipos: los SLICEL (*Slice logic*), dedicados exclusivamente a lógica, y los SLICEM (*Slice with Memory*), que también pueden configurarse como memorias distribuidas para almacenar datos intermedios o constantes [20].

Las memorias internas BRAM se implementan como bloques configurables de puerto simple o doble, ofreciendo almacenamiento temporal de alta velocidad para búferes, colas o tablas de búsqueda. Los bloques DSP integran multiplicadores, sumadores y registros con opciones de *pipeline*, permitiendo encadenar operaciones aritméticas de forma eficiente y mejorar el rendimiento en tareas de procesamiento intensivo.

Finalmente, toda la infraestructura se integra mediante una red de interconexión programable que enlaza los recursos locales de cada slice, conecta bloques vecinos y coordina distintas *clock regions*, asegurando rutas de señal optimizadas y sincronización precisa en todo el dispositivo.

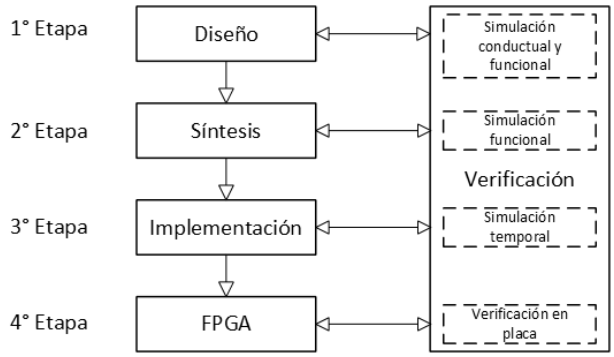


Fig. 2.3: Etapas principales del flujo de desarrollo en FPGA, desde el diseño lógico hasta la verificación en placa. Elaboración propia basada en la documentación de Vivado Design Suite 2025 (Xilinx).

#### 2.1.1.2. Flujo de desarrollo en FPGA

El diseño en FPGA sigue un flujo estructurado que transforma una descripción lógica del sistema en una configuración física lista para ser cargada al dispositivo [22]. Este proceso parte de una especificación funcional escrita en un HDL y culmina con la programación de la arquitectura interna del chip mediante la asignación concreta de recursos y conexiones lógicas. La Fig. 2.3 presenta de forma resumida las etapas principales de este flujo, las cuales se describen a continuación con mayor detalle.

1. **Descripción del sistema.** Se modela el comportamiento mediante lenguajes VHDL o SystemVerilog. Esta descripción a nivel de transferencia de registros RTL define la lógica combinacional y secuencial, incluyendo mecanismos de control, sincronización y procesamiento de datos. En esta fase se preparan además bancos de prueba (*testbench*) para validar funcionalmente el diseño ante distintos estímulos, así como componentes reutilizables (*Intellectual Property*, IP) y máquinas de estados finitos (*Finite State Machines*, FSM) para modelar sistemas con transiciones discretas.

2. **Síntesis lógica.** En esta etapa todavía no se implementa nada físicamente en la FPGA. El código HDL se convierte en una red lógica equivalente, generando un archivo intermedio llamado *netlist*, que describe compuertas, registros y conexiones. Esta fase garantiza que el diseño puede realizarse con los recursos disponibles y que es coherente a nivel funcional.

3. **Implementación física.** A partir de la *netlist*, las herramientas de desarrollo ejecutan los procesos de mapeo, colocación y ruteo. Aquí se decide cómo se distribuyen y asignan los bloques lógicos, memorias y rutas internas de la FPGA. Aunque todavía no se programa el dispositivo, ya se obtiene una representación física concreta optimizada según área, latencia y temporización.

4. **Generación del *bitstream*.** Completada la implementación, se genera el archivo de configuración o *bitstream*, que contiene toda la información necesaria para que la FPGA defina internamente sus LUTs, memorias, interconexiones y demás recursos. Este archivo se carga posteriormente en el dispositivo, por ejemplo a través de JTAG (*Joint Test Action Group*), una interfaz estándar utilizada para prueba y programación, permitiendo que la FPGA adopte la funcionalidad establecida en el diseño.

Tras la programación, se puede realizar la verificación en placa (*in-circuit verification*), comprobando el funcionamiento del sistema bajo condiciones reales de operación. Durante el análisis de temporización se evalúan los márgenes de seguridad temporal (*slack*) para garantizar el correcto funcionamiento secuencial del sistema. El margen negativo más desfavorable (*Worst Negative Slack*, WNS) indica cuánto falta para cumplir el tiempo mínimo de establecimiento de una señal, mientras que el total de márgenes negativos de establecimiento (*Total Negative Slack*, TNS) representa la suma de todos estos casos. Respecto al retardo de retención, el peor margen de retención (*Worst Hold Slack*, WHS) refleja el mayor incumplimiento de permanencia de una señal tras el flanco activo del reloj, y el total de errores por retención (*Total Hold Slack*, THS) corresponde a la suma de estos eventos. Finalmente, el peor margen de ancho de pulso del reloj (*Worst Pulse Width Slack*, WBSS) indica el pulso más corto registrado que incumple el mínimo necesario, y el total de errores por ancho de pulso (*Total Pulse Width Slack*, TPWS) cuantifica la cantidad de fallos relacionados. Estos parámetros permiten validar en conjunto que la suma de pulsos, flancos y permanencias se comporta correctamente durante cada ciclo de operación.

Entre las herramientas de desarrollo más utilizadas se encuentran Vivado (Xilinx), Quartus Prime (Intel) y Libero SoC (Microchip), que integran todas estas etapas en entornos de diseño unificado, facilitando un proceso iterativo de optimización y validación.

#### 2.1.1.3. La FPGA como coprocesador especializado

En muchas arquitecturas integradas, las FPGAs se utilizan para asumir tareas de procesamiento intensivo, descargando al procesador principal y aumentando el rendimiento global del sistema. Tal como señala [6], este enfoque permite delegar en la FPGA operaciones críticas mientras el procesador se concentra en la supervisión y coordinación, logrando así un mayor aprovechamiento de los recursos disponibles.

La efectividad de este rol no depende solo de la lógica implementada, sino también de las interfaces de comunicación que permiten el intercambio de datos y señales de control con el resto del sistema. Estas interfaces determinan la velocidad, la latencia y la flexibilidad con que la FPGA se integra al flujo de trabajo, y su elección se basa en el volumen de datos, la complejidad de implementación y los recursos disponibles. En sistemas embebidos de bajo consumo o espacio reducido se emplean enlaces simples como UART para tareas de configuración y lectura de estados internos [23], o SPI (*Serial Peripheral Interface*) para transferencias síncronas de velocidad moderada, utilizado con frecuencia en memorias externas y convertidores de datos [24]. También se usa $I^2C$ (*Inter-Integrated Circuit*) cuando varios periféricos comparten un mismo bus, minimizando pines [25]. Para mayores volúmenes de datos o sincronización estricta se recurre a buses de alta velocidad como PCIe (*Peripheral Component Interconnect Express*), común en tarjetas aceleradoras [26], o protocolos internos como AXI (*Advanced eXtensible Interface*), que permiten a la FPGA acceder a memorias compartidas y comunicarse con otros bloques lógicos, reduciendo la latencia y mejorando la respuesta del sistema [27].

Tabla 2.1: Resumen de interfaces de comunicación entre el *host* y la FPGA. Elaboración propia.

| **Interfaz** | **Uso principal** | **Velocidad típica** |
|---|---|---|
| UART | Comunicación serie simple para configuración y datos de control. | Hasta 1 Mbps; varios Mbps con hardware dedicado. |
| SPI | Transferencia síncrona para memorias y periféricos moderados. | 1–50 Mbps según dispositivo. |
| $I^2C$ | Bus multipunto de bajo consumo para sensores y periféricos lentos. | 100 kbps, 400 kbps o hasta 3.4 Mbps. |
| PCIe | Alta velocidad para grandes volúmenes de datos. | 2.5–16 GT/s por carril (según generación). |
| AXI | Protocolo interno para memoria y comunicación entre bloques IP. | Frecuencias de 100–500 MHz o más. |

## 2.2. Aprendizaje automático

Los modelos de aprendizaje automático permiten inferir relaciones subyacentes a partir de datos representativos del fenómeno que se desea modelar. En el caso del aprendizaje supervisado, este proceso se basa en ejemplos donde tanto las entradas como las salidas esperadas son conocidas, lo que permite ajustar el modelo para generalizar patrones y predecir el comportamiento de datos no observados [28, 29].

Desde un enfoque matemático, un modelo supervisado se construye a partir de un conjunto de entrenamiento compuesto por $N$ pares ordenados $(X^{(i)}, Y^{(i)})$, donde cada $X^{(i)}$ representa un vector de características extraídas del fenómeno observado, y $Y^{(i)}$ corresponde a su clase asociada. Por ejemplo, en el análisis de señales eléctricas, $X^{(i)}$ puede incluir parámetros como energía, frecuencia dominante o tiempo de subida, mientras que $Y^{(i)}$ indica si dicha señal pertenece a una condición normal o a un evento anómalo. El objetivo del aprendizaje consiste en encontrar una función $f^*$ tal que, dada una nueva entrada $X^{(i)}$, se obtenga una salida estimada $\hat{Y}^{(i)} = f^*(X^{(i)})$ que se aproxime a la etiqueta real $Y^{(i)}$.

Una vez entrenado, el modelo puede analizar las características numéricas de nuevas señales y emitir una clasificación automática sobre su estado. Esta capacidad de generalización, adquirida durante el proceso de aprendizaje, permite traducir patrones complejos en reglas de decisión prácticas, aplicables en sistemas que requieren respuestas rápidas y confiables.

Además de su solidez teórica, la clasificación supervisada posee una lógica de inferencia estructurada y modular, lo que facilita su implementación eficiente y reproducible. Estas propiedades la vuelven especialmente adecuada para su integración en plataformas reconfigurables como las FPGA, donde pueden desplegarse arquitecturas paralelas con baja latencia y control temporal preciso [16]. Tales capacidades resultan clave en sistemas embebidos dedicados a clasificación binaria, donde la velocidad de respuesta y la fiabilidad del procesamiento son factores determinantes. Bajo este enfoque, el diseño del sistema debe contemplar tanto la precisión del modelo como su eficiencia estructural al ser llevado a hardware. En particular, algunos esquemas de clasificación operan sobre espacios de características numéricas y toman decisiones según la similitud entre instancias, lo que introduce desafíos concretos al implementar su lógica en plataformas físicas.

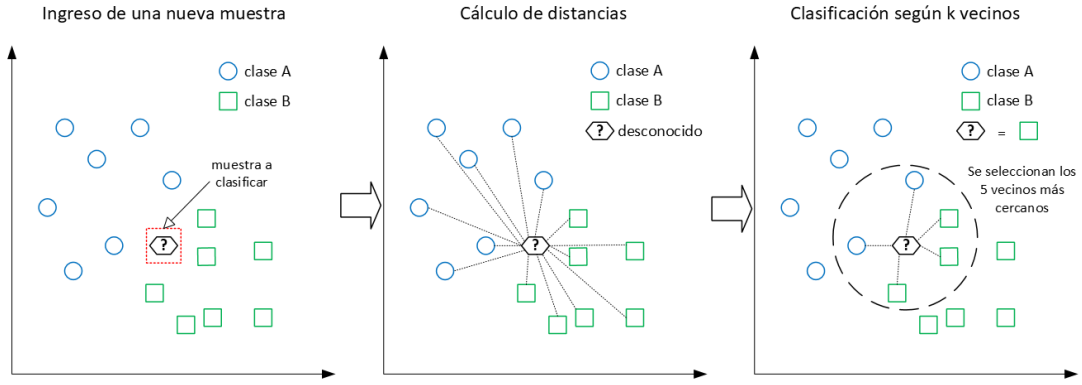


Fig. 2.4: Esquema funcional del algoritmo k-NN para clasificación supervisada. Elaboración propia.

### 2.2.1. Algoritmo $k$-Nearest Neighbors

En el ámbito de los métodos supervisados basados en proximidad, el algoritmo k-NN se reconoce como una técnica eficiente y ampliamente utilizada. Su naturaleza no paramétrica se sustenta en la noción de vecindad dentro de un espacio métrico: para cada muestra desconocida se identifican las k instancias más cercanas, donde k indica el número de vecinos considerados, y se asigna a la muestra la clase más frecuente entre ellas [11].

La proximidad entre muestras se establece mediante el cálculo de distancias entre vectores de características, permitiendo tomar decisiones directamente sobre los datos, sin necesidad de una fase de entrenamiento explícita ni de un modelo interno complejo. En esencia, $k$-NN realiza inferencias basadas exclusivamente en el conjunto de datos disponible, adaptándose dinámicamente al comportamiento observado [30].

Tal como se ilustra en la Fig. 2.4, el algoritmo $k$-NN sigue una secuencia clara que incluye el ingreso de una muestra, el cálculo de distancias respecto al conjunto de entrenamiento, la identificación de los $k$ vecinos más cercanos y la asignación de una clase mediante un mecanismo de votación. Este flujo, aunque simple desde el punto de vista conceptual, puede dividirse en módulos funcionales claramente delimitados al ser implementado en hardware especializado [28].

En una arquitectura digital reconfigurable, cada etapa del algoritmo se asocia a bloques lógicos específicos. El módulo de entrada adquiere y transforma las muestras en vectores internos; el cálculo de distancias se implementa mediante unidades aritméticas optimizadas; la selección de vecinos se organiza como un módulo de ordenamiento que determina las $k$ distancias mínimas, y la decisión de clase se resuelve mediante un bloque lógico que contabiliza las etiquetas para determinar la clase predominante. Este enfoque modular facilita tanto la estructuración del diseño como la optimización de recursos y la paralelización selectiva de las tareas más demandantes [31].

En este esquema, los bloques responsables del cálculo de distancias y la selección de vecinos representan los principales cuellos de botella durante la inferencia, especialmente en aplicaciones que exigen respuesta inmediata y alta confiabilidad. La elección de la métrica y del algoritmo de ordenamiento impacta directamente en el rendimiento global del sistema, por lo que resulta esencial analizar estos componentes tanto desde su formulación teórica como desde su viabilidad en arquitecturas digitales especializadas [12].

### 2.2.2. Métricas de Distancia

En términos formales, una métrica es una función que asigna un valor real no negativo a cada par de puntos en un espacio, cumpliendo con las propiedades de no negatividad, identidad (la distancia entre un punto y sí mismo es cero), simetría (la distancia de A a B es igual a la de B a A) y desigualdad triangular (la distancia directa entre dos puntos no excede la suma de distancias intermedias). Estas condiciones permiten establecer nociones de cercanía y separación fundamentales para los algoritmos de clasificación [29].

Una formulación general que abarca múltiples métricas es la distancia de Minkowski, definida por:

$$d_p(x, y) = \left( \sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p} \tag{2.2.1}$$

Esta expresión depende del parámetro $p$, que modula la geometría del espacio de decisión. La familia de Minkowski permite representar desde trayectorias ortogonales hasta diagonales, ofreciendo una transición continua entre distintos modelos de distancia. Esta flexibilidad resulta útil en entornos donde las variables no siguen una estructura uniforme. Sin embargo, valores de $p > 2$ implican operaciones costosas como potencias fraccionarias, lo que complica su implementación en hardware. Por ello, las aplicaciones prácticas suelen centrarse en dos casos particulares: $p = 2$ (distancia euclidiana) y $p = 1$ (distancia manhattan)[30].

La distancia euclidiana, correspondiente a $p = 2$, representa la trayectoria recta entre dos puntos en un espacio n-dimensional:

$$d_{\text{euclidiana}}(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{2.2.2}$$

Esta métrica preserva la estructura geométrica del espacio, siendo útil en problemas donde las relaciones espaciales entre atributos son significativas, como en reconocimiento de imágenes o análisis tridimensional. No obstante, su implementación digital requiere operaciones complejas, multiplicaciones y raíces cuadradas, lo cual limita su viabilidad en plataformas con recursos limitados o latencia crítica [29].

Por otro lado, la distancia Manhattan, derivada de $p = 1$, se define como:

$$d_{\text{manhattan}}(x, y) = \sum_{i=1}^{n} |x_i - y_i| \tag{2.2.3}$$

Evalúa la suma de diferencias absolutas entre coordenadas, describiendo trayectorias alineadas con los ejes ortogonales. Aunque es menos sensible a relaciones espaciales sutiles, su gran ventaja es la simplicidad computacional: no requiere multiplicaciones ni raíces, lo que la hace especialmente eficiente en arquitecturas digitales. Es ideal para sistemas embebidos, clasificadores en tiempo real y plataformas FPGA, donde se priorizan bajo consumo, latencia predecible y lógica aritmética elemental [12].

La Figura 2.5 ilustra cómo cada métrica genera una estructura de decisión distinta. Minkowski produce una familia de formas que varía con $p$, interpolando entre geometrías ortogonales y circulares. La Euclidiana define regiones radiales basadas en magnitud vectorial, apropiadas para problemas con sensibilidad espacial, mientras que la Manhattan delimita regiones cuadradas alineadas con los ejes cartesianos, favoreciendo su implementación digital. Esta diversidad geométrica permite evaluar el equilibrio entre representatividad matemática y eficiencia computacional en el diseño de clasificadores embebidos [29].

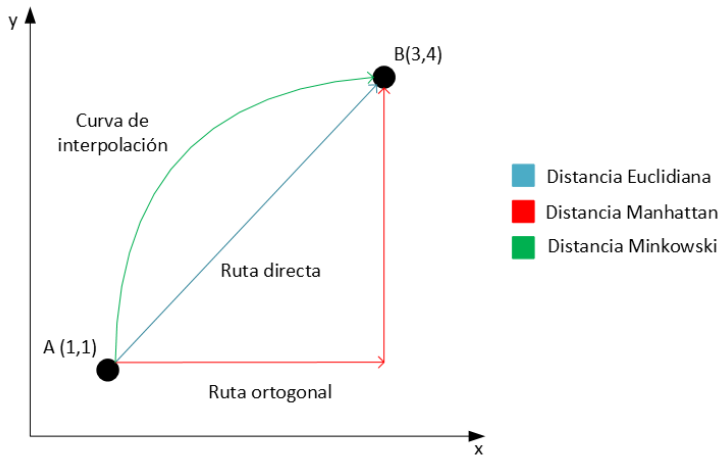


Fig. 2.5: Comparación gráfica entre las métricas de distancia Minkowski, Euclidiana y Manhattan. Elaboración propia.

### 2.2.3. Algoritmos de Ordenamiento

Los algoritmos de ordenamiento se diferencian principalmente por su complejidad computacional, la cual influye de forma directa en su escalabilidad y en su adecuación a distintas arquitecturas digitales. Esta consideración es especialmente relevante en plataformas como las FPGAs, donde la eficiencia en el uso de recursos lógicos y la reducción de la latencia son factores determinantes en el diseño [6].

Entre los métodos con complejidad cuadrática $O(n^2)$ se encuentran *Bubble Sort*, *Insertion Sort* y *Selection Sort*. Su estructura sencilla los hace apropiados para arquitecturas reconfigurables con recursos limitados. *Insertion Sort* mejora el rendimiento en secuencias parcialmente ordenadas, mientras que *Selection Sort* ofrece un patrón de acceso predecible. A pesar de sus limitaciones en cuanto a escalabilidad, estos métodos resultan útiles cuando el volumen de datos es reducido y se prioriza un flujo lógico trazable [7]. La Fig. 2.6 muestra el proceso clásico de ordenamiento mediante *Bubble Sort*, donde cada iteración desplaza los valores mayores hacia el extremo derecho hasta obtener la secuencia final ordenada. Estos algoritmos parten siempre del supuesto de disponer del conjunto completo de datos antes de iniciar el proceso.

Por otra parte, algoritmos con complejidad $O(n \log n)$, como *Merge Sort*, *Bitonic Sort* y *Quick Sort*, son más eficientes para manejar grandes volúmenes de datos, pero también asumen que el conjunto completo está disponible antes de iniciar la ordenación. *Merge Sort* permite una ejecución jerárquica segmentada, aunque requiere control adicional para la fusión; *Bitonic Sort*, gracias a su estructura paralelizable, se adapta especialmente bien al hardware de las FPGAs al permitir múltiples comparaciones simultáneas mediante redes de comparadores; y *Quick Sort*, pese a su buen rendimiento en software, introduce dificultades en hardware por su naturaleza recursiva y sus patrones de acceso irregulares [13].

Sin embargo, en muchas aplicaciones prácticas los datos no se presentan de manera completa, sino que llegan de forma continua. En estos escenarios resulta más eficiente mantener dinámicamente un subconjunto relevante en lugar de ordenar todo el conjunto. Para este propósito se emplea el enfoque de ordenamiento *k-top*, cuyo objetivo es conservar únicamente los $k$ valores más pequeños (o más relevantes) conforme se reciben los datos. Este procedimiento se adapta de manera natural a arquitecturas secuenciales implementadas en FPGA, ya que evita reorganizar la totalidad del arreglo

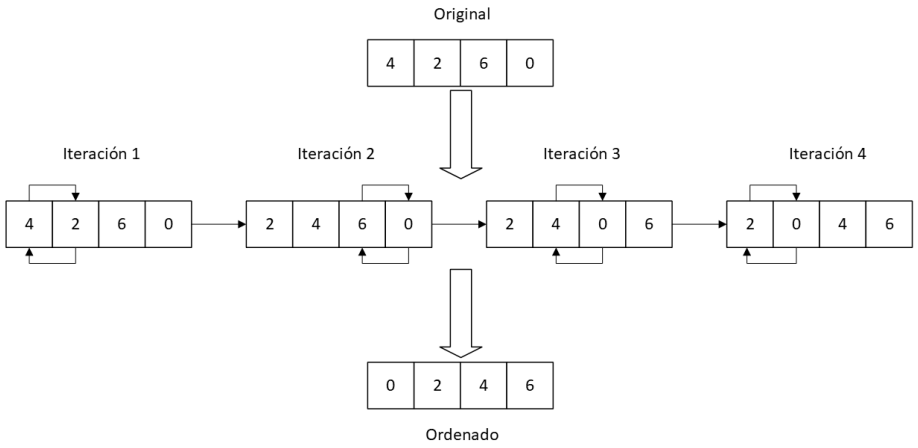


Fig. 2.6: Funcionamiento de Bubble Sort. Requiere disponer de todo el conjunto de datos para completar el ordenamiento, desplazando iterativamente los valores mayores. Elaboración propia.

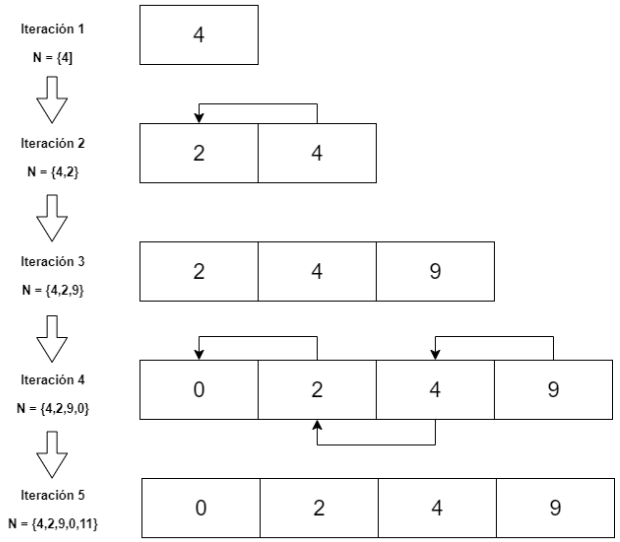


Fig. 2.7: Funcionamiento de un ordenamiento secuencial con $k = 4$. Cada nuevo dato se evalúa en tiempo real y solo se mantienen los cuatro valores más pequeños observados. Elaboración propia.

y concentra los recursos en un subconjunto acotado. En la Fig. 2.7 se ilustra este mecanismo, donde con cada nuevo dato se actualiza una lista parcial de manera inmediata [13].

Cada comparador necesario en esta técnica se traduce en LUTs y registros internos, de modo que el tamaño de $k$ impacta directamente en la cantidad de comparadores y en la latencia por ciclo. Así, el ordenamiento *k-top* ofrece una solución eficiente y especializada para flujos de datos en tiempo real, alineándose con los requerimientos de clasificación incremental en hardware [12].

### 2.2.4. Despliegue funcional de k-NN sobre FPGA

Una vez establecidos los fundamentos del clasificador, resulta relevante revisar cómo se ha implementado el algoritmo $k$-NN en plataformas de hardware reconfigurable. La literatura reciente muestra múltiples experiencias que confirman no solo la viabilidad de su despliegue en FPGA, sino también su capacidad para ofrecer procesamiento en tiempo real con un uso eficiente de los recursos disponibles.

En el ámbito industrial, [32] desarrollaron un sistema embebido basado en un clasificador $k$-NN ponderado para el diagnóstico de motores eléctricos, implementado sobre una tarjeta Xilinx PYNQ-Z2. Por su parte, [33] presentaron una arquitectura en VHDL sobre una FPGA Artix-7, aplicada a métodos de modulación por ancho de pulso digital (DPWM) para el control de sistemas de potencia, demostrando la utilidad del algoritmo en entornos con restricciones estrictas de latencia.

En el área biomédica, [34] implementaron un sistema de clasificación de señales EEG epilépticas sobre un SoC Zynq-7000, integrando descriptores de Hjorth como características y alcanzando latencias cercanas a 15 ms, validando así el procesamiento neurofisiológico en tiempo real sobre hardware reconfigurable. De manera complementaria, [35] desarrollaron un acelerador específico para el algoritmo, optimizando el flujo de comparación y almacenamiento de datos con el objetivo de reducir el consumo energético, factor clave en dispositivos embebidos portátiles y *wearables*.

Otras áreas también se benefician de este tipo de soluciones. [36] presentaron un acelerador de búsqueda sobre FPGA para el registro de nubes de puntos, incorporando un enfoque mejorado de *Locality Sensitive Hashing* (LSH) que permite localizar vecinos más cercanos en datos tridimensionales con una latencia de apenas 0.64 ms.

En aplicaciones orientadas a seguridad, [37] propusieron un esquema de inferencia privada que combina $k$-NN con encriptación homomórfica acelerada en FPGA. Su implementación sobre una Intel Agilex 7 alcanzó latencias de 0.85 ms para datos cifrados, frente a los 41.72 ms obtenidos mediante un esquema tradicional, gracias al procesamiento paralelo en 32 canales.

También se encuentran propuestas en dominios especializados. Por ejemplo, [38] plantearon una variante del algoritmo $k$-NN implementada en FPGA para la demodulación en tiempo real de señales m-QAM, alcanzando tasas superiores a 100 Msímbolos/s en un dispositivo Zynq UltraScale+ y reduciendo la latencia cerca de un 30 % respecto a soluciones convencionales en software. De forma similar, [39] presentaron una arquitectura que acelera el algoritmo $k$-NN sobre datos cifrados mediante encriptación homomórfica, logrando tiempos de inferencia equivalentes al procesamiento sobre texto plano.

En conjunto, estos trabajos evidencian la adaptabilidad del algoritmo $k$-NN y su capacidad para integrarse de manera eficiente en arquitecturas reconfigurables, demostrando su idoneidad para aplicaciones de clasificación en tiempo real y bajo restricciones de recursos.

Capítulo 3

# METODOLOGÍA

En este capítulo se describe la metodología seguida para diseñar e implementar el sistema de clasificación k-NN en FPGA. El proceso se estructuró en etapas progresivas que abarcan desde la preparación del conjunto de datos y el entrenamiento inicial del algoritmo en un entorno de software, hasta la definición de hiperparámetros, la arquitectura del coprocesador en RTL y la integración de interfaces de comunicación. Cada sección detalla los pasos realizados, las herramientas empleadas y la forma en que los bloques se conectaron para construir un sistema modular, escalable y reproducible.

Primero se establecieron los vectores de características a partir de señales adquiridas y se ajustaron parámetros como el número de vecinos $k$ y la métrica Manhattan, optimizando el diseño para su posterior implementación digital. Luego se desarrollaron módulos independientes en SystemVerilog que abarcan la recepción de datos vía UART, el empaquetado secuencial de muestras, el acceso a memorias BRAM, el cálculo de distancias, la selección de vecinos y el cómputo final de la clase mayoritaria. Finalmente, se integró una interfaz de comunicación externa para enviar los datos procesados y recibir los resultados de clasificación, completando así un flujo de trabajo que va desde la definición algorítmica hasta la realización física sobre la FPGA.

## 3.1. Conjunto de datos y entrenamiento del clasificador $k$-NN

El desarrollo del clasificador $k$-NN se llevó a cabo siguiendo un flujo metodológico definido que integra adquisición experimental, procesamiento espectral y preparación de datos para su despliegue en hardware reconfigurable.

El proceso comienza con la adquisición de un conjunto de 742 eventos experimentales obtenidos en laboratorio[1]. Cada evento corresponde a una señal de descargas parciales, fenómeno asociado a procesos de ionización localizada en sistemas de aislamiento eléctrico, o bien a registros de ruido eléctrico. Se capturaron 455 eventos identificados como descargas tipo corona y 287 como ruido, cada uno compuesto por 200 muestras obtenidas a una frecuencia de muestreo de 200 MHz. Esta resolución temporal de 1 µs permite capturar con precisión la dinámica de los pulsos y constituye una base sólida para el análisis espectral posterior. Un ejemplo representativo de ambas clases se muestra en la Fig. 3.1, donde se aprecia claramente la morfología impulsiva de la señal de descarga tipo corona frente a la oscilación irregular de la señal de ruido.

Cada señal fue transformada al dominio de la frecuencia mediante una transformada rápida de Fourier (*Fast Fourier Transform*, FFT), calculando a partir del espectro dos métricas representativas:

[1] Datos obtenidos en el Departamento de Ingeniería Eléctrica de la Universidad Técnica Federico Santa María, sede San Joaquín.

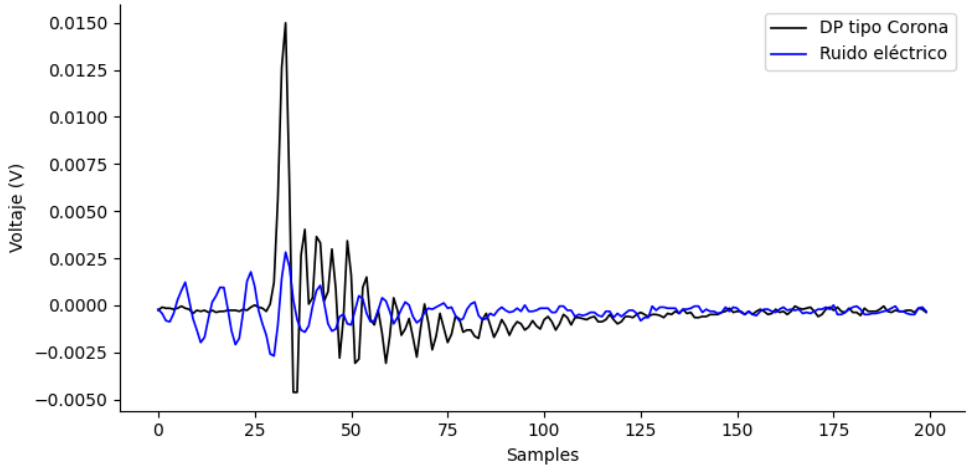


Fig. 3.1: Señales representativas del conjunto de datos adquiridos: descarga parcial tipo corona y señal de ruido eléctrico, cada una compuesta por 200 muestras a 200 MHz.

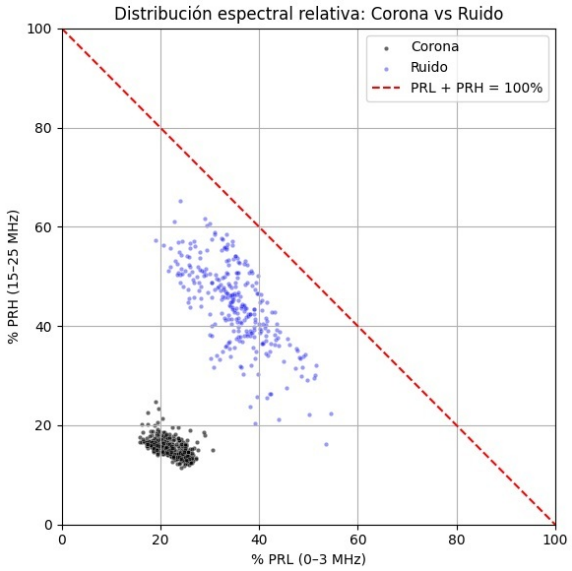


Fig. 3.2: Distribución espectral de los eventos en el plano PRL–PRH. La línea punteada roja indica el límite físico PRL + PRH = 100 %.

la relación de potencia en baja frecuencia (*Power Ratio Low*, PRL), asociada al rango de 0–3 MHz, y la relación de potencia en alta frecuencia (*Power Ratio High*, PRH), correspondiente al rango de 15–25 MHz. Estos valores se normalizaron respecto al total de energía espectral y se escalaron a enteros de 16 bits, generando así vectores compactos y directamente utilizables en hardware. La Fig. 3.2 muestra la distribución resultante en el plano PRL–PRH, evidenciando una separación natural entre clases y validando la idoneidad de estas características para el proceso de clasificación.

Para su integración en la FPGA, los vectores de entrenamiento fueron almacenados en memorias internas BRAM mediante un archivo de inicialización `.coe`. Este archivo contiene los valores enteros

Código 3.1: Fragmento del archivo `.coe` utilizado para inicializar las BRAM con vector PRL. El archivo .coe para PRH y label siguen la misma estructura.

```
memory_initialization_radix=10;
memory_initialization_vector=
1500,
1430,
1302,
...
;
```

de PRL y PRH junto con el respectivo label de 1 bit que indica la clase, y se carga automáticamente durante la síntesis del sistema, eliminando la necesidad de transferencias externas y garantizando acceso inmediato desde la lógica implementada. Un fragmento de dicho archivo se presenta en el Código 3.1, que actúa como punto de partida para la operación autónoma del clasificador en tiempo real.

Este procedimiento asegura que la FPGA disponga desde el inicio de un conjunto de datos consistente y optimizado, permitiendo que el clasificador $k$-NN funcione con latencia reducida y total determinismo, sin depender de cargas dinámicas ni procesos adicionales de configuración.

## 3.2. Selección de hiperparámetros del algoritmo $k$-NN

La configuración de los parámetros internos del clasificador se definió bajo criterios metodológicos que equilibran rendimiento, consumo de recursos y facilidad de implementación en hardware reconfigurable.

### 3.2.1. Selección del número de vecinos $k$

Para determinar el valor adecuado de $k$, se aplicó el método empírico conocido como método del codo (*elbow method*), que permite identificar un punto de equilibrio entre complejidad y rendimiento. Este procedimiento consistió en evaluar la tasa de error del clasificador para distintos valores de $k$ en la validación por software, graficar los resultados y analizar la curva obtenida. El valor óptimo corresponde al punto donde la tasa de error deja de decrecer de forma significativa, logrando así un balance adecuado entre sobreajuste y capacidad de generalización.

Con el conjunto de entrenamiento definido, la clase asignada a cada evento se obtiene mediante una votación por mayoría simple entre los $k$ vecinos más cercanos. En caso de empate, se adopta una convención fija que privilegia una de las clases, garantizando un comportamiento determinista. Esta estrategia es eficiente para arquitecturas secuenciales, pues minimiza la latencia y simplifica la lógica de decisión.

### 3.2.2. Elección de la métrica Manhattan

Durante el diseño del clasificador, se evaluaron tres métricas de distancia aplicables al algoritmo $k$-NN: Manhattan, Euclidiana y Minkowski. El objetivo fue seleccionar la opción más eficiente para su implementación en FPGA, considerando criterios clave como el costo computacional, la facilidad de implementación y el uso de recursos lógicos, sin comprometer la capacidad de clasificación sobre los datos PRL–PRH discretizados a 16 bits.

Tabla 3.1: Comparación simplificada de métricas para k-NN en FPGA.

| Métrica | Costo computacional | Facilidad de implementación | Uso de recursos |
|---|---|---|---|
| Manhattan | 3 | 3 | 3 |
| Euclidiana | 2 | 2 | 1 |
| Minkowski | 1 | 1 | 1 |

**Notas:** Puntajes del 1 al 3, donde 3 indica mayor favorabilidad. Manhattan es la opción preferente para datos discretizados a 16 bits y arquitecturas FPGA debido a su simplicidad y eficiencia. Por otro lado, Euclidiana y Minkowski resultan más adecuadas para conjuntos de datos continuos o de mayor complejidad, pero su mayor costo computacional las hace menos idóneas para implementaciones con recursos limitados.

Como se aprecia en la Tabla 3.1, la métrica Manhattan obtiene la puntuación más elevada, fundamentada en que su cálculo se basa exclusivamente en sumas y restas de valores absolutos, lo que evita multiplicaciones y raíces cuadradas que aumentan considerablemente la complejidad aritmética y el consumo de recursos lógicos en el caso de la métrica Euclidiana. En contraste, Minkowski, pese a su generalidad, no aporta beneficios prácticos significativos y eleva el costo computacional, razón por la cual es escasamente empleada en implementaciones reales [28].

Además, implementaciones previas de $k$-NN en FPGA han destacado que la métrica Manhattan posibilita una reducción sustancial en el consumo de LUTs y registros, disminuye la latencia y permite alcanzar mayores frecuencias de operación, manteniendo un desempeño competitivo [40, 41].

Por estos motivos, y en consideración a las particularidades de los datos discretizados a 16 bits y las exigencias de eficiencia propias del hardware reconfigurable, la métrica Manhattan se adopta como la opción óptima para este trabajo, equilibrando desempeño y eficiencia en la implementación.

### 3.2.3. Selección del algoritmo de ordenamiento

La metodología contempla comparar dos algoritmos para identificar los $k$ vecinos más cercanos tras el cálculo de distancias: la propuesta principal basada en el algoritmo incremental *k-top* y un algoritmo clásico secuencial, representado por *Bubble Sort* debido a su simplicidad y naturaleza secuencial del flujo de datos. La comparación se realizará a nivel de implementación hardware sobre FPGA, evaluando eficiencia en el uso de recursos lógicos, tiempos de ejecución y exactitud en la selección de los $k$ vecinos. Estas comparaciones permitirán determinar cuál algoritmo ofrece el mejor compromiso entre precisión y eficiencia para la arquitectura reconfigurable implementada.

## 3.3. Coprocesador en FPGA

La Fig. 3.3 muestra la arquitectura funcional del coprocesador implementado en la FPGA, sincronizado por un reloj global de 100 MHz y una señal de reinicio (`reset`) conectada al bloque `global_reset` que asegura la inicialización ordenada. Las entradas llegan por `rx` mediante UART y el resultado se entrega por `tx`, los módulos de recepción, procesamiento, cálculo de distancias, ordenamiento y transmisión se comunican entre sí mediante buses y señales de control sincronizadas, garantizando la correcta transferencia de datos; cada uno de estos bloques se describe a continuación.

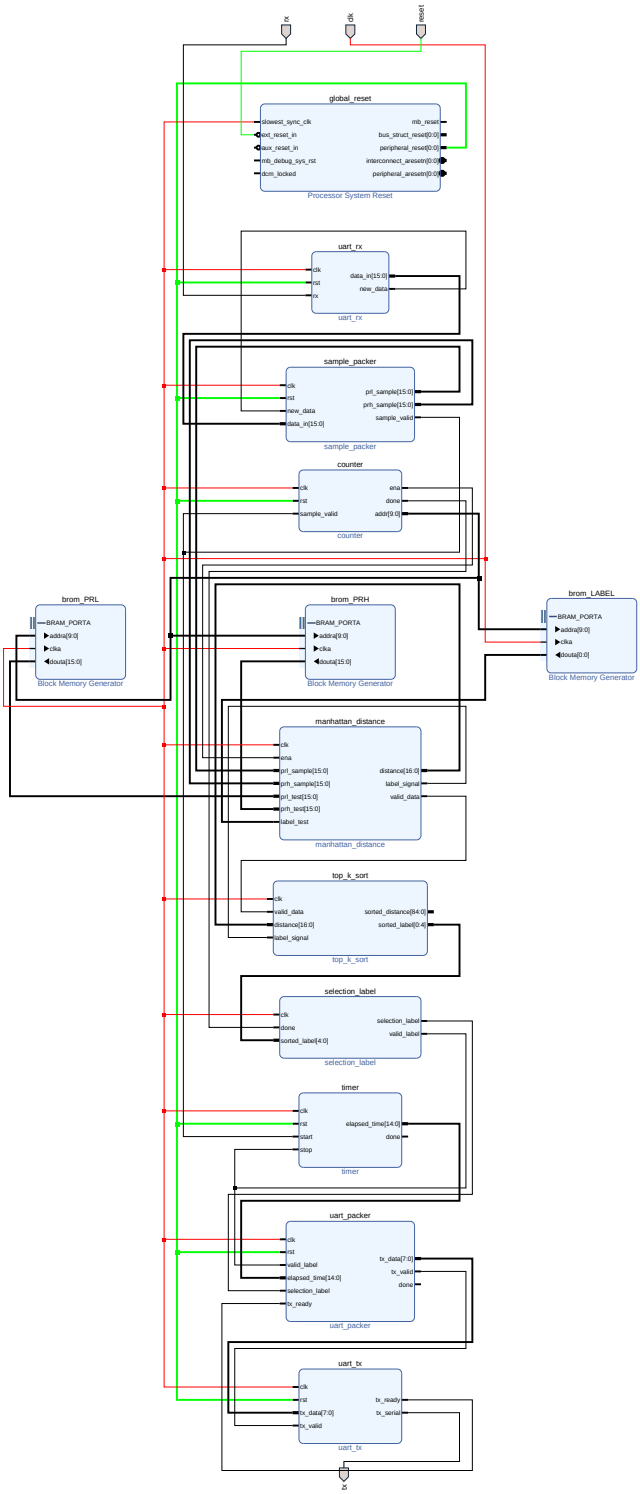


Fig. 3.3: Arquitectura funcional del sistema de clasificación k-NN implementado en FPGA. Las líneas rojas indican el reloj principal (100 MHz), las verdes corresponden a la señal de reinicio, y las negras representan el flujo interno de datos.

### 3.3.1. Recepción UART y ensamblado de entrada

Como etapa inicial del flujo de clasificación, se diseñó un subsistema dedicado a la recepción de datos mediante la interfaz UART. En este módulo, los valores espectrales PRL y PRH se transmiten en formato entero sin signo de 16 bits, divididos en dos bytes consecutivos. Para su correcta reconstrucción se implementó el bloque `uart_rx`, encargado de ensamblar las palabras y generar pulsos de validación sincronizados con el sistema.

Este bloque está compuesto por dos submódulos. El primero, denominado `rx`, implementa un receptor UART básico de 8 bits, sin bit de paridad, con bit de inicio y parada, operando a una velocidad de hasta 460800 baudios. Su funcionamiento se basa en una máquina de estados finita que recorre las fases `IDLE`, `START`, `DATA` y `STOP`, junto con un contador de ciclos que asegura la captura precisa de cada bit, de acuerdo con el parámetro `CLKS_PER_BIT`. Los bits seriales se almacenan secuencialmente en un registro de desplazamiento, reconstruyendo así cada byte recibido (ver Código 3.2).

Código 3.2: Captura de bits en el estado DATA del receptor UART

```
if (clk_count == CLKS_PER_BIT - 1) begin
    clk_count <= 0;
    rx_shift_reg[bit_index] <= rx;
    bit_index <= (bit_index == 7) ? 0 : bit_index + 1;
end else begin
    clk_count <= clk_count + 1;
end
```

El segundo submódulo se encarga de ensamblar las palabras de 16 bits a partir de los bytes recibidos. Para ello, emplea una máquina de estados con dos fases: `WAIT_MSB` y `WAIT_LSB`. En la primera se almacena el byte más significativo en un registro temporal; en la segunda se concatena con el byte menos significativo, formando la palabra completa bajo la convención *big-endian*. Al completarse este proceso, se genera un pulso de sincronización (`new_data`) que indica la disponibilidad del dato ensamblado para su posterior procesamiento (ver Código 3.3).

Código 3.3: Máquina de estados para ensamblado de palabra de 16 bits

```
if (byte_valid_pulse) begin
    case (state)
        WAIT_MSB: msb_reg <= byte_data;
        WAIT_LSB: begin
            data_in <= {msb_reg, byte_data};
            new_data <= 1;
        end
    endcase
end
```

Para garantizar una captura confiable y evitar múltiples ensamblados del mismo byte, se utiliza un detector de flanco ascendente sobre la señal `data_valid`. Esto permite generar un pulso único por byte recibido, asegurando una transferencia sincronizada incluso bajo condiciones de alta velocidad de transmisión.

### 3.3.2. Empaquetado secuencial de muestras

Tras la recepción UART y el ensamblado de palabras de 16 bits, se incorporó el módulo intermedio `sample_packer`, cuya función es reagrupar las secuencias binarias para conformar pares

espectrales representativos de cada evento. Este bloque recibe de forma secuencial dos palabras: primero el componente de baja frecuencia y luego el de alta frecuencia. Una vez completado el par, el sistema emite las señales `prl_sample` y `prh_sample`, junto con un pulso de un solo ciclo denominado `sample_valid`, que indica que la muestra está completa y lista para ser procesada por la lógica de clasificación.

La lógica interna está controlada por un selector (`sel`) que alterna entre los estados de espera de la primera y la segunda palabra. En la primera transición, el valor recibido se almacena como PRL; al llegar la segunda palabra, se guarda como PRH y se genera el pulso de validación correspondiente. Esta lógica secuencial permite delimitar eventos de forma precisa y sin necesidad de temporización externa (ver Código 3.4).

Código 3.4: Alternancia secuencial y generación de muestra válida

```
if (valid) begin
    if (sel == 0) begin
        prl_sample <= data_in;
        sel <= 1;
    end else begin
        prh_sample <= data_in;
        sel <= 0;
        sample_valid <= 1;
    end
end
```

Además de marcar la completitud de la muestra, la señal `sample_valid` actúa como *trigger* sincronizado que activa simultáneamente tanto el bloque de control principal como el temporizador encargado de contabilizar los ciclos de clasificación. Esta doble función la posiciona como eje temporal del sistema, con una implementación compacta y fácilmente integrable.

### 3.3.3. Contador de control secuencial

Para gestionar el acceso ordenado a la memoria y regular el ciclo de inferencia, se implementó un contador parametrizable denominado `counter`, que actúa como núcleo de control del clasificador. Este bloque coordina el inicio del procesamiento, el avance secuencial por las muestras almacenadas en la BRAM y la señalización del término del proceso. La operación comienza al recibir un pulso de arranque (`sample_valid`), reiniciando el contador a cero, y continúa incrementando su valor en cada flanco de reloj hasta alcanzar el umbral definido por el parámetro `MAX_ADDR`. Durante esta fase, la salida `addr` entrega en cada ciclo la dirección correspondiente para el acceso a memoria.

Código 3.5: Conteo secuencial y pipeline de señales de control

```
if (rst)
  cnt <= '0;
else if (start)
  cnt <= '0;
else if (cnt < MAX_ADDR)
  cnt <= cnt + 1;

ena_pipe <= { ena_pipe[DELAY-2:0], (cnt < MAX_ADDR) };
done_pipe <= { done_pipe[DELAY-1:0], (cnt == MAX_ADDR - 1) };
```

Además del conteo principal, el módulo implementa una lógica de propagación temporal mediante registros tipo *pipeline*, que permite generar con retardo controlado las señales de activación `ena` y finalización `done`. La primera habilita el bloque de cálculo de distancias, mientras que la segunda

marca el término del ciclo de clasificación y activa el bloque de votación. Este esquema introduce latencias conocidas entre etapas, asegurando una transición sincronizada y estable a lo largo del flujo de procesamiento.

### 3.3.4. Memorias configuradas como ROM

Para almacenar el conjunto de datos de entrenamiento se utilizaron tres bloques de memoria interna generados con el asistente *Block Memory Generator* de Vivado Suite, configurados en modo solo lectura (*Block Read-Only Memory*, BROM) e inicializados a partir de archivos `.coe` generados en la etapa de preprocesamiento. Cada bloque almacena un componente específico del dataset: valores PRL, valores PRH y las etiquetas binarias asociadas.

Todas las memorias comparten una misma señal de dirección proveniente del módulo `counter`, lo que permite la lectura paralela y sincronizada de los tres datos correspondientes a cada muestra, garantizando su alineación y evitando ambigüedades.Cada bloque entrega sus valores a través de `douta`: PRL y PRH son valores de 16 bits cada uno, mientras que el *label* asociado es de 1 bit. Al operar en modo BROM, no requiere señales de escritura ni control adicional, lo que simplifica la lógica y evita condiciones de carrera.

### 3.3.5. Cálculo de distancia Manhattan

El módulo `manhattan_distance` calcula la distancia entre la muestra de entrada y cada vector del conjunto de entrenamiento almacenado en memoria ROM, utilizando la métrica Manhattan implementada directamente sobre los valores espectrales. La operación se expresa como:

$$\texttt{distance} = |\texttt{prl\_sample} - \texttt{prl\_test}| + |\texttt{prh\_sample} - \texttt{prh\_test}|$$

Los valores `prl_sample` y `prh_sample` provienen del bloque `sample_packer`, mientras que `prl_test` y `prh_test` se leen en paralelo desde las memorias ROM mediante la dirección entregada por el contador principal, lo que garantiza la alineación temporal de los datos. La señal de habilitación `ena`, emitida con un retardo controlado, activa el cálculo durante un ciclo de reloj, y la salida se estabiliza en el ciclo siguiente.

Además del valor de distancia, el módulo propaga la etiqueta binaria correspondiente al vector de entrenamiento mediante la señal `label_test`, la cual se retransmite sin modificaciones como `label_signal`. Ambas salidas se consideran válidas cuando se activa la señal `valid_data`, la cual se genera a partir de `ena` para mantener la coherencia temporal del sistema. Este esquema asegura que los resultados se transmitan en condiciones estables y sincronizadas hacia las etapas de ordenamiento y clasificación. El núcleo funcional se muestra en el Código 3.6.

Código 3.6: Cálculo de distancia Manhattan en hardware

```
if (ena) begin
  automatic logic [WIDTH:0] d_prl =
       (prl_sample > prl_test) ? prl_sample - prl_test : prl_test - prl_sample;

  automatic logic [WIDTH:0] d_prh =
       (prh_sample > prh_test) ? prh_sample - prh_test : prh_test - prh_sample;

  distance <= d_prl + d_prh;
  label_signal <= label_test;
end
```

### 3.3.6. Ordenamiento de distancias y selección de vecinos

Para identificar los $k$ vecinos más cercanos a la muestra actual, se implementó el módulo `top_k_sort`, encargado de mantener en tiempo real un arreglo ordenado con las menores distancias calculadas. En una primera etapa de diseño se evaluó un ordenamiento completo mediante el algoritmo Bubble Sort, como se muestra en el Código 3.7, con fines de exploración funcional.

Código 3.7: Algoritmo Bubble Sort clásico

```
for i = 0 to N-2
   for j = 0 to N-i-2
       if array[j] > array[j+1]
           swap array[j] and array[j+1]
```

Dado el carácter secuencial del sistema, se adoptó una estrategia más eficiente basada en inserción ordenada. El bloque almacena localmente un arreglo de tamaño fijo $K$, inicializado con valores máximos. En cada ciclo válido (controlado por `valid_data`), la nueva distancia se compara secuencialmente con los registros existentes. Si resulta menor, los valores se desplazan hacia la derecha y se inserta en la posición correspondiente, manteniendo el orden creciente. Este mecanismo se ilustra en el Código 3.8.

Código 3.8: Inserción ordenada y desplazamiento

```
if (!inserted && distance < temp_dists[i]) begin
   for (int j = K-1; j > i; j--) begin
       temp_dists[j] = temp_dists[j-1];
       temp_classes[j] = temp_classes[j-1];
   end
   temp_dists[i] = distance;
   temp_classes[i] = label_signal;
   inserted = 1;
   break;
end
```

El bloque mantiene de forma paralela las etiquetas de clase asociadas a cada distancia, entregando como salida los arreglos `sorted_distance` y `sorted_label`. De este modo, se conserva un conjunto actualizado de vecinos ordenados, listo para ser evaluado por el bloque de decisión final.

### 3.3.7. Decisión binaria y temporización sincronizada

El sistema activa en paralelo dos bloques complementarios que cumplen funciones críticas y temporales diferenciadas. El primero, `selection_label`, determina la clase predominante entre los $K$ vecinos más cercanos a partir del vector de etiquetas `sorted_label` generado por el bloque de ordenamiento. La decisión se toma mediante un conteo combinacional de ocurrencias por clase, priorizando la clase '1' en caso de empate. Esta lógica, basada en un recorrido secuencial del arreglo de etiquetas y un mecanismo de resolución de empates simple pero robusto, se sintetiza en el Código 3.9, donde se observa cómo se acumulan las ocurrencias y se define la clase resultante.

Código 3.9: Conteo y votación mayoritaria

```
count_ones = '0;
count_zeros = '0;
for (int i = 0; i < K; i++) begin
    if (sorted_label[i])
        count_ones++;
    else
        count_zeros++;
end
selection_label = (count_ones >= count_zeros); // Empate: predomina el '1'
```

En paralelo, el módulo `timer` cuantifica los ciclos de reloj transcurridos entre el inicio de la clasificación y la validación del resultado final. Su activación se produce con el flanco ascendente de `sample_valid`, que marca el inicio del proceso inferencial, y finaliza al capturar el flanco de `valid_label`, emitido justo cuando la decisión ha sido tomada. El valor total registrado se guarda en `elapsed_time`, acompañado de un pulso de sincronización a través de la señal `done`. El Código 3.10 detalla esta lógica secuencial, donde el contador interno se habilita, acumula y detiene de forma perfectamente alineada con los eventos críticos del sistema.

Código 3.10: Conteo de ciclos entre sample_valid y valid_label

```
if (start_rise) begin
    contando <= 1;
    contador <= 0;
end else if (stop_rise && contando) begin
    elapsed_time <= contador;
    contando <= 0;
end else if (contando) begin
    contador <= contador + 1;
end
done <= stop_rise && contando;
```

Gracias a esta arquitectura paralela, el sistema entrega de forma simultánea la clase inferida mediante `selection_label` y la latencia total de clasificación a través de `elapsed_time`, ambos completamente sincronizados con el flujo lógico de procesamiento digital.

### 3.3.8. Empaquetamiento y transmisión secuencial UART

El módulo `uart_packer` encapsula de forma ordenada la clase inferida y el tiempo de inferencia en un único paquete de 16 bits, cuya estructura corresponde a {`selection_label, elapsed_time`}. Esta operación se activa mediante la señal `valid_label`, que marca la disponibilidad simultánea de ambos valores. Al detectarse dicha condición, el bloque almacena el paquete en un registro intermedio, iniciando el proceso de transmisión.

La lógica de control sigue una máquina de estados que gestiona la salida secuencial del byte alto y bajo, respetando la señalización de disponibilidad del transmisor UART mediante `tx_ready`. Una vez completado el segundo envío, el módulo genera un pulso `done`, indicando que el proceso ha finalizado exitosamente y el dato ha sido completamente despachado. La implementación de esta lógica secuencial se sintetiza en el Código 3.11.

Código 3.11: Núcleo de la máquina de estados del empaquetador UART

```
case (estado)
    IDLE: if (valid_label) begin
              buffer <= {selection_label, elapsed_time};
              estado <= ENVIA_BYTE1;
          end
    ENVIA_BYTE1: if (tx_ready) begin
                     tx_data <= buffer[15:8];
                     tx_valid <= 1;
                     estado <= WAIT_READY_1;
                 end
    WAIT_READY_1: if (!tx_ready)
                      estado <= ENVIA_BYTE0;
    ENVIA_BYTE0: if (tx_ready) begin
                     tx_data <= buffer[7:0];
                     tx_valid <= 1;
                     estado <= WAIT_READY_0;
                 end
    WAIT_READY_0: if (!tx_ready) begin
                     estado <= IDLE;
                     done <= 1;
                 end
endcase
```

Este mecanismo garantiza una transferencia compacta, secuencial y sincronizada del resultado final del sistema, asegurando compatibilidad directa con interfaces UART estándar y manteniendo la integridad de los datos en todo momento.

### 3.3.9. Transmisión final por interfaz UART

La etapa final del flujo corresponde al módulo `uart_tx`, cuya función es transmitir de manera serial cada byte generado por el bloque anterior. Este transmisor implementa una máquina de estados que sigue el protocolo UART convencional, comenzando con un bit de inicio, seguido por la transmisión de los ocho bits del dato (de menor a mayor peso) y concluyendo con un bit de parada. Todo el proceso opera a una tasa de hasta 460800 baudios, sincronizada con el reloj del sistema.

Para ejecutar esta tarea, el módulo utiliza un registro de desplazamiento que conserva el byte a emitir, junto con un contador de ciclos parametrizado mediante la constante `CLKS_PER_BIT`, calculada en función de la frecuencia de reloj y la velocidad deseada. Cuando la señal `tx_valid` indica que un nuevo dato está disponible y la línea está ociosa, el sistema captura el byte mediante `tx_data` y avanza por los distintos estados de control. El recorrido completo de la máquina se muestra en el Código 3.12.

Una vez completada la transmisión, el módulo restablece su estado y reactiva la señal `tx_ready`, indicando que está disponible para recibir un nuevo byte. De esta forma, se garantiza una comunicación fiable, secuencial y sin pérdida de datos, cerrando adecuadamente el ciclo de procesamiento del sistema.

Código 3.12: Máquina de estados del transmisor UART

```
case (state)
    IDLE: begin
        tx_serial_reg <= 1'b1;
        if (tx_valid_reg) begin
            shift_reg <= tx_data_reg;
            clk_count <= '0;
            state <= START_BIT;
        end
    end
    START_BIT: begin
        tx_serial_reg <= 1'b0;
        if (clk_count == CLKS_PER_BIT-1) begin
            clk_count <= '0;
            bit_index <= '0;
            state <= DATA_BITS;
        end else clk_count <= clk_count + 1;
    end
    DATA_BITS: begin
        tx_serial_reg <= shift_reg[bit_index];
        if (clk_count == CLKS_PER_BIT-1) begin
            clk_count <= '0;
            bit_index <= bit_index + 1;
            if (bit_index == 3'd7) state <= STOP_BIT;
        end else clk_count <= clk_count + 1;
    end
    STOP_BIT: begin
        tx_serial_reg <= 1'b1;
        if (clk_count == CLKS_PER_BIT-1) begin
            clk_count <= '0;
            state <= CLEANUP;
        end else clk_count <= clk_count + 1;
    end
    CLEANUP: state <= IDLE;
endcase
```

## 3.4. Interfaz de comunicación y preprocesamiento de los datos

La interfaz gráfica fue desarrollada en Python empleando bibliotecas como `tkinter` y `ttkbootstrap` para la construcción de la GUI, `matplotlib` para visualización dinámica, y `pandas` junto con `numpy` para el manejo y procesamiento eficiente de datos. La comunicación UART asíncrona con la FPGA se implementó mediante `pyserial`, `threading` y `queue`, asegurando un intercambio fluido y no bloqueante de información. Este entorno integra de forma coherente el control del coprocesador, el preprocesamiento de señales y la validación cruzada de los resultados, facilitando una evaluación integral del sistema.

### Métrica y ordenamiento utilizados

Con el objetivo de mantener la coherencia entre software y hardware, el entorno en Python replica la métrica Manhattan utilizada en la FPGA. La distancia se calcula como la suma de las diferencias absolutas entre los componentes PRL y PRH del evento de prueba y los vectores de entrenamiento, empleando operaciones vectorizadas provistas por `NumPy`. Posteriormente, se aplica `numpy.argsort` para ordenar las distancias y seleccionar los $k$ vecinos más cercanos, siguiendo un enfoque directo y reproducible.

Mientras que en software se utiliza un ordenamiento global basado en índices, la FPGA realiza esta tarea mediante una inserción ordenada incremental en el bloque `top_k_sort`, optimizado para hardware. A pesar de sus diferencias estructurales, ambas estrategias producen resultados equivalentes, lo cual garantiza consistencia metodológica a lo largo del flujo de validación.

### Generación de archivos de entrenamiento

Los datos de entrenamiento, capturados originalmente en entorno de laboratorio, son sometidos a una FFT y posteriormente normalizados. Tras este procesamiento, los valores espectrales se escalan por un factor de 100 y se redondean para representarlos como enteros de 16 bits, adaptándolos al dominio digital del coprocesador. El sistema genera tres archivos `.coe` a partir de estos datos: dos que contienen los valores de PRL y PRH, y uno adicional con las etiquetas binarias asociadas. Estos archivos se almacenan en el directorio de trabajo y se utilizan para inicializar las memorias BROM del sistema, evitando la necesidad de transferencias dinámicas durante la inferencia.

### Control y pruebas disponibles

Una vez cargados los datos y generados los archivos de inicialización, el sistema permite ejecutar la inferencia en FPGA enviando los vectores espectrales por UART, utilizando dos palabras de 16 bits por evento. La transmisión opera a velocidades de hasta 460800 baudios, permitiendo un procesamiento eficiente incluso bajo alta carga de eventos. La FPGA responde con un paquete de salida que contiene el bit de clase asignado y la cantidad de ciclos empleados durante la inferencia, reflejando directamente el rendimiento del sistema.

Complementariamente, la interfaz permite ejecutar la clasificación en software sobre la misma base de datos, lo que posibilita una comparación directa entre ambos dominios. Esta evaluación se realizó en un equipo con procesador AMD Ryzen 5 5600 (6 núcleos a 3.5 GHz), 16 GB de RAM y tarjeta gráfica NVIDIA GeForce GTX 1660 SUPER de 6 GB, cuyas capacidades permitieron una ejecución fluida y comparaciones precisas de tiempo y precisión.

Para asegurar la continuidad del flujo de datos sin pérdidas, se empleó un mecanismo de recepción asíncrona basado en hilos y colas de mensajes, que desacopla la recepción serial del procesamiento de eventos. Todos los resultados se presentan de forma inmediata en la GUI, mostrando la clase asignada (descarga parcial o ruido), junto con la estimación temporal obtenida a partir de los ciclos consumidos. La interfaz también permite exportar los resultados a archivos para análisis posterior sin necesidad de repetir las pruebas.

## 3.5. Monitoreo y validación en hardware

Para verificar el funcionamiento interno del sistema se incorporaron analizadores lógicos integrados (*Integrated Logic Analyzer*, ILA) en puntos clave de la arquitectura, permitiendo un análisis sectorizado de las señales. En particular, se utilizaron los núcleos `ILA_rx`, `ILA_distance` e `ILA_selection`, que facilitaron la observación de las transacciones internas durante la etapa de recepción de datos, el cálculo de distancias y la selección de vecinos, respectivamente. Estos analizadores fueron ubicados estratégicamente en cada subsistema clave, proporcionando trazabilidad completa del flujo de datos en tiempo real.

De manera complementaria, la validación externa se llevó a cabo utilizando el equipo *Analog Discovery 3*, conectado directamente a las líneas físicas `TX` y `RX` de la interfaz UART del coprocesador. Para ello se emplearon los puertos `JA` y `JB` de la placa, estableciendo la conexión eléctrica necesaria. La verificación de la señal se realizó mediante el software *WaveForms*, herramienta oficial del dispositivo, que permitió capturar y analizar las formas de onda transmitidas y recibidas, asegurando una comunicación fiable entre la FPGA y la aplicación en Python.

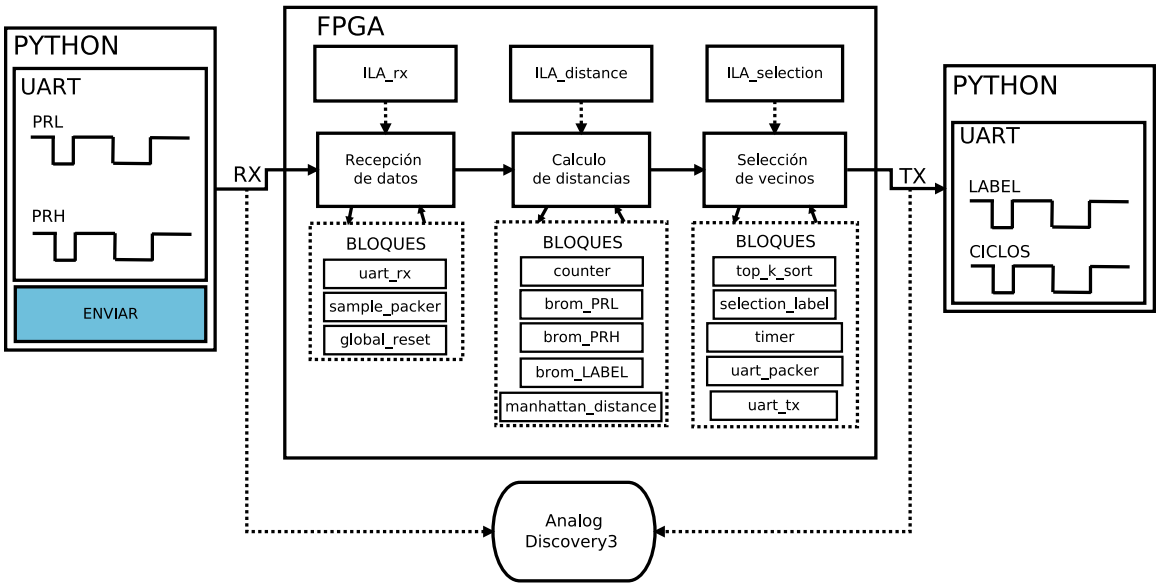


Fig. 3.4: Arquitectura de monitoreo y validación en hardware con ILAs internos y validación externa mediante Analog Discovery 3 usando la aplicación WaveForms.

La Fig. 3.4 muestra de forma esquemática la arquitectura de validación empleada, donde se distinguen los bloques funcionales del diseño en FPGA, las rutas de datos entre ellos y la interacción con la aplicación en Python mediante UART. El monitoreo interno mediante ILA, junto con la corroboración externa utilizando instrumentación física, permite una verificación integral del sistema, tanto a nivel lógico como en su interacción con el entorno operativo.

Capítulo 4

# RESULTADOS

Este capítulo presenta los resultados obtenidos a partir de la implementación del clasificador binario basado en el algoritmo k-NN sobre una plataforma FPGA, resaltando el comportamiento alcanzado en términos de exactitud, eficiencia y consistencia interna. Los datos provienen de un proceso de validación con un conjunto representativo de eventos etiquetados y permiten sustentar con solidez las conclusiones expuestas. Se incluyen las métricas empleadas para caracterizar el rendimiento, tales como la precisión de clasificación, la latencia de inferencia y el uso de recursos lógicos.

Los resultados se organizan en dos etapas complementarias. Primero, se expone una validación funcional que confirma la correspondencia entre el sistema embebido y el modelo de referencia en software. Luego, se presenta un análisis cuantitativo y estructural que permite caracterizar con mayor detalle el desempeño de la solución implementada. Esta organización facilita una lectura progresiva de los hallazgos.

## 4.1. Verificación funcional del sistema

Durante las pruebas se verificó que el sistema embebido entregó, efectivamente y conforme a lo esperado, para cada evento procesado, un paquete de 16 bits en el que los 15 bits menos significativos indicaron el número de ciclos de reloj empleados, mientras que el bit más significativo reflejó la clase asignada (1 para descarga parcial y 0 para ruido).

La interfaz ilustrada en la Fig. 4.1 permitió monitorear en tiempo real la clase resultante, la latencia medida por hardware y la posición espectral de cada evento clasificado. La evidencia obtenida confirmó una correspondencia directa entre los eventos enviados, los resultados obtenidos y los tiempos de respuesta reportados por la FPGA.

En todos los casos evaluados se obtuvo una coincidencia del 100 % entre la clasificación generada por el sistema embebido y la producida por el clasificador de referencia en software, lo que respalda la consistencia lógica de la implementación sobre la plataforma. Además, los resultados confirman que el sistema embebido no solo reproduce de manera exacta la lógica del clasificador, sino que también entrega información interpretable y trazable desde la capa de aplicación. Esta verificación funcional constituye una base sólida para los análisis que se desarrollan en apartados posteriores de este capítulo.

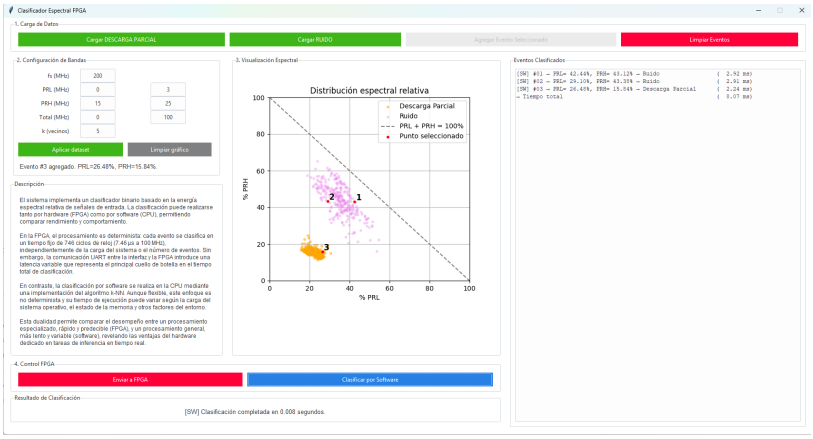


Fig. 4.1: Interfaz gráfica en Python utilizada para visualizar en tiempo real la clase resultante, el tiempo de clasificación y la posición espectral de cada evento.

### 4.1.1. Evaluación del parámetro $k$

El análisis evidenció que la estabilidad del modelo se mantiene frente a variaciones del hiperparámetro $k$. Como se muestra en la Fig. 4.2, todos los valores evaluados alcanzaron una precisión del 100 %, lo que evidencia una separación clara entre las clases consideradas: ruido y descarga parcial.

Dado que no se observó pérdida de exactitud en ninguno de los casos, la selección final respondió a criterios de diseño: se adoptó $k = 5$ por ofrecer un equilibrio adecuado entre robustez estadística, simplicidad lógica y eficiencia de implementación en la arquitectura embebida.

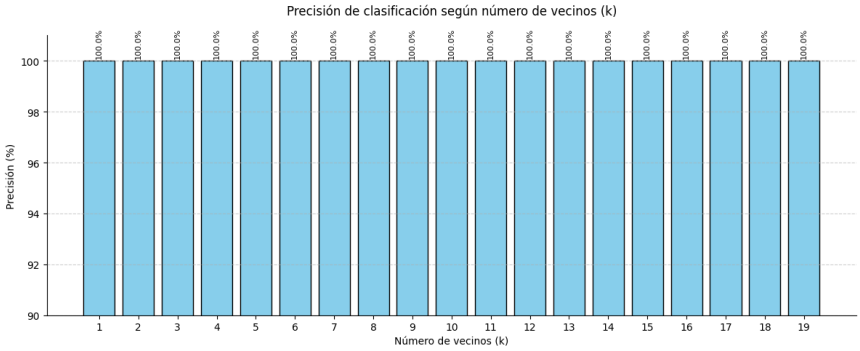


Fig. 4.2: Precisión de clasificación en función del parámetro $k$. Todos los valores evaluados entregan clasificación perfecta.

### 4.1.2. Validación estructural RTL

Las señales capturadas evidenciaron que la secuencia lógica de cada etapa, desde el ingreso de datos hasta la transmisión del resultado, se ejecutó de manera determinista y sincronizada, cumpliendo con el diseño especificado. Los resultados se presentan siguiendo tres bloques funcionales: ingreso de datos y control inicial, carga de características desde BRAM y cálculo de distancias, y ordenamiento, selección de vecinos y empaquetado del resultado final.

#### 4.1.2.1. Etapa 1: Ingreso de datos y control inicial

Tal como se aprecia en la Figura 4.3, el sistema organiza los datos UART entrantes en pares `(PRL, PRH)` mediante el bloque `sample_packer`, que emite la señal `sample_valid` al completar una muestra. Este flujo está habilitado por la señal `new_data`, generada por el bloque `uart_rx` al recibir correctamente una palabra de 16 bits.

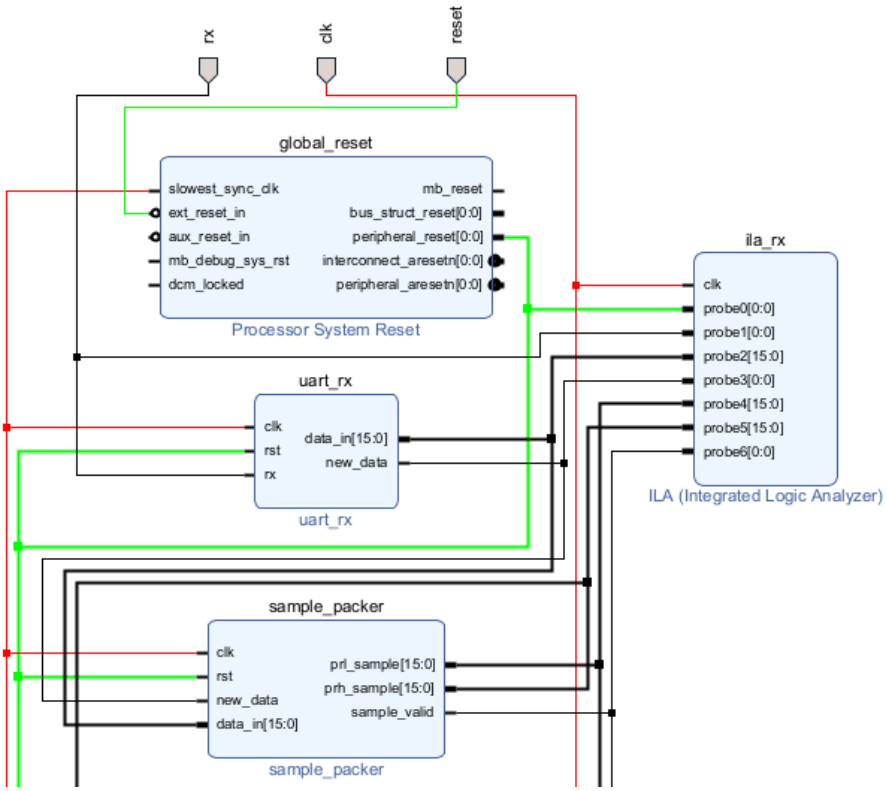


Fig. 4.3: Vista RTL de la etapa de ingreso de datos. El bloque `uart_rx` entrega los datos recibidos a `sample_packer`, que reorganiza los valores y activa `sample_valid` cuando la muestra está completa.

En la traza ILA de la Fig. 4.4, se observa el proceso completo de adquisición. Primero se recibe el valor `2506`, que corresponde al componente PRL. Este dato es almacenado internamente y se mantiene estable hasta la llegada del segundo valor `1481`, correspondiente al PRH. La señal `new_data` se activa en el instante en que finaliza la recepción del segundo byte por UART, y en el ciclo inmediatamente posterior se genera un pulso en `sample_valid`. Esto indica que el par `(2506, 1481)` ha sido ensamblado correctamente y está listo para su procesamiento. Cabe destacar que en la primera mitad de la trama UART pueden visualizarse datos anteriores aún presentes en las líneas internas, sin interferir en el flujo actual. La transición es clara y no se observan superposiciones ni pérdidas de validez. Este comportamiento confirma que la lógica de control asegura una captura robusta y sin ambigüedades, consolidando el correcto funcionamiento de la etapa de ingreso de datos.

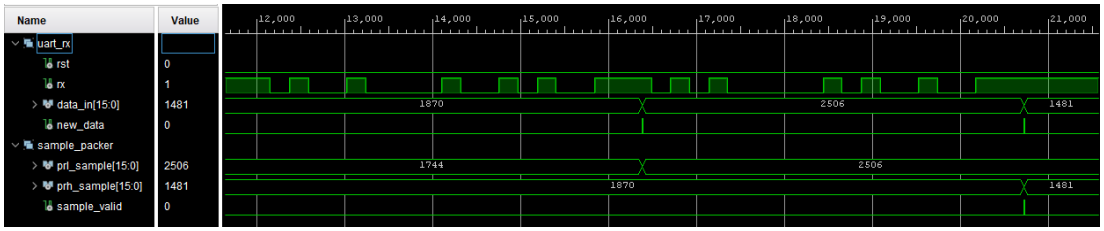


Fig. 4.4: Captura ILA de la etapa de ingreso. Se visualiza la recepción secuencial de `2506` (PRL) y `1481` (PRH). La señal `sample_valid` se activa justo después de completar la muestra.

#### 4.1.2.2. Etapa 2: Carga desde BROM y cálculo de distancia

Luego de la activación de `sample_valid`, el sistema inició automáticamente el recorrido del conjunto de entrenamiento almacenado en BROM, habilitando la lectura secuencial de características y etiquetas sin interrupciones. Tal como se aprecia en la arquitectura RTL de la Fig. 4.5, el módulo `counter` generó direcciones ascendentes mediante la señal `addr`, manteniendo activa la línea `ena` durante toda la operación. Esta señal permitió extraer, en cada ciclo, los datos correspondientes desde las memorias internas `PRL`, `PRH` y `LABEL`, que fueron entregados directamente al bloque `manhattan_distance` para su procesamiento.

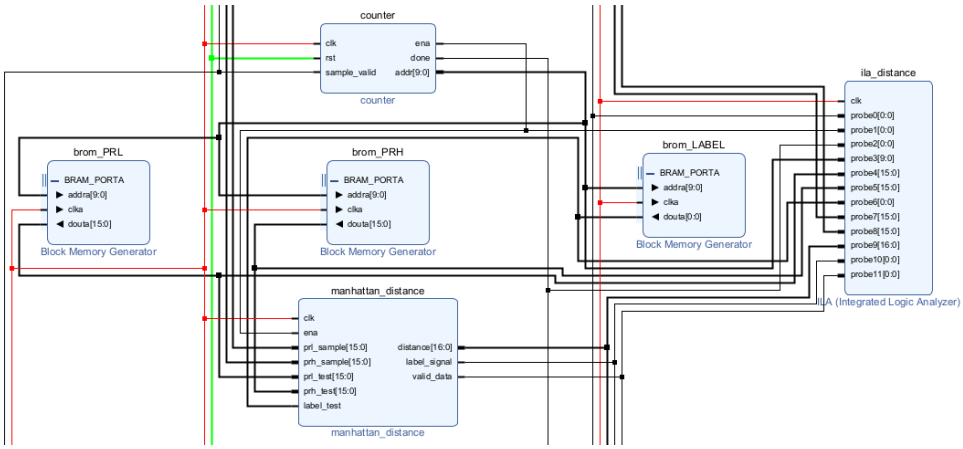


Fig. 4.5: Vista RTL de la etapa de cálculo de distancias. Las memorias `PRL`, `PRH` y `LABEL` son recorridas por un contador sincronizado que entrega sus salidas al bloque `manhattan_distance`.

En la traza de la Fig. 4.6 se verifica que el recorrido comienza desde `addr = 0`, pero la primera salida válida corresponde a `addr = 1`. Esto se debe a la latencia estructural del sistema: en el ciclo $S_{510}$ se solicita la dirección, en $S_{511}$ se reflejan los datos leídos desde memoria, y en $S_{512}$ se genera el resultado de distancia. Esta secuencia se mantiene constante durante toda la operación, permitiendo un flujo de procesamiento continuo, con una nueva salida válida en cada ciclo a partir del segundo acceso.

Un ejemplo concreto se observa cuando se accede a la dirección `addr = 1`. En ese instante, el sistema compara la muestra de prueba `(2506, 1481)` con el vector de entrenamiento `(1940, 1702)`. La distancia Manhattan se calcula como:

$$|2506 - 1940| + |1481 - 1702| = 566 + 221 = \mathbf{787}$$

Este valor que coincide exactamente con el resultado observado en la señal `distance` durante el ciclo $S_{512}$. La clase correspondiente también se refleja correctamente en la salida `label_signal`, y ambas señales se encuentran validadas por `valid_data`. Este comportamiento consistente confirma el alineamiento temporal entre dirección, lectura y resultado, validando el funcionamiento continuo y sin pérdidas del bloque de distancia.

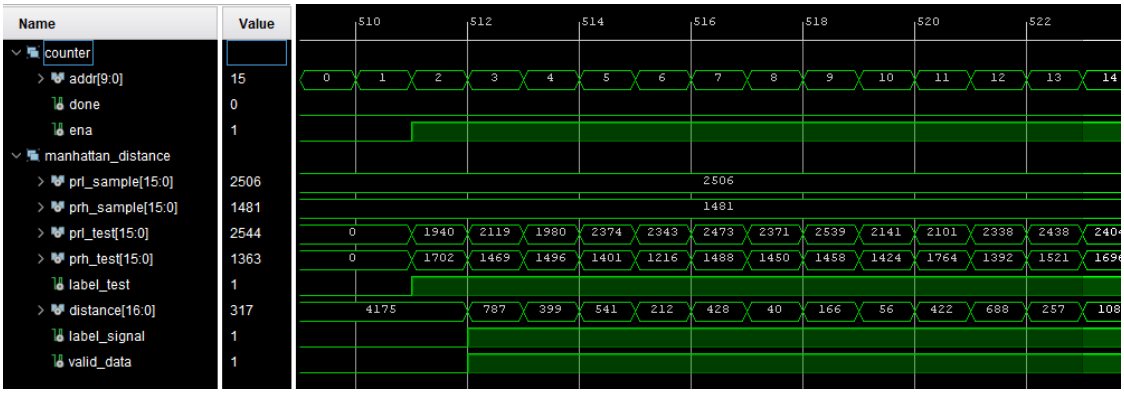


Fig. 4.6: Captura ILA de la etapa de cálculo de distancia. Se observa el recorrido secuencial de la señal `addr`, la lectura en paralelo de `prl_test`, `prh_test` y `label_test`, y la emisión sincronizada de resultados en `distance`, `label_signal` y `valid_data`.

#### 4.1.2.3. Etapa 3: Ordenamiento, selección y transmisión por UART

Al completarse el recorrido de distancias, el sistema ingresó automáticamente en la etapa final de inferencia, compuesta por tres bloques clave: ordenamiento, selección de clase y transmisión. Como se observa en la arquitectura RTL de la Fig. 4.7, las distancias generadas por el bloque `manhattan_distance` fueron entregadas directamente al módulo `top_k_sort`, que mantuvo en tiempo real las $k$ distancias mínimas junto con sus respectivas etiquetas, utilizando un mecanismo de inserción ordenada.

Una vez ingresada la última distancia válida, el bloque `selection_label` evaluó las etiquetas almacenadas para determinar la clase mayoritaria. En la traza de la Fig. 4.8 se observa que este proceso culmina en el ciclo $S_{508}$ con la activación de la señal `done`, seguido en $S_{509}$ por la habilitación de `valid_label` junto con la clase resultante (`selection_label = 1`), ya lista para ser empaquetada.

En paralelo, el módulo `timer` acumula el conteo de ciclos desde la activación de `sample_valid`, y en el ciclo $S_{510}$ se genera un pulso en la señal `done` del temporizador; en $S_{511}$ se activa `tx_valid`, indicando que el bloque `uart_packer` ha finalizado la composición de la palabra de salida. Esta incluye la clase final en el bit más significativo y el tiempo de inferencia en los 15 bits restantes, formando la palabra `1000001011101001`, correspondiente a 745 ciclos y clase 1. La transmisión comienza inmediatamente a través de `tx_serial`, completando la inferencia sin interrupciones.

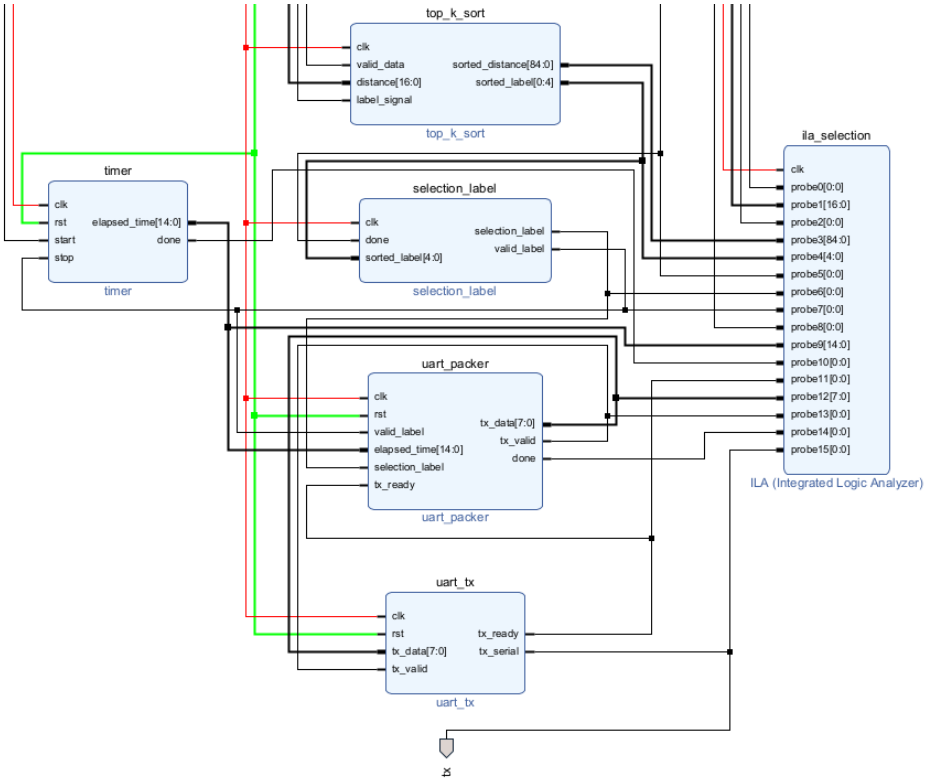


Fig. 4.7: Vista RTL de la etapa final. Las distancias se ordenan, se selecciona la clase mayoritaria, y junto con el tiempo de inferencia se empaquetan y transmiten mediante UART.

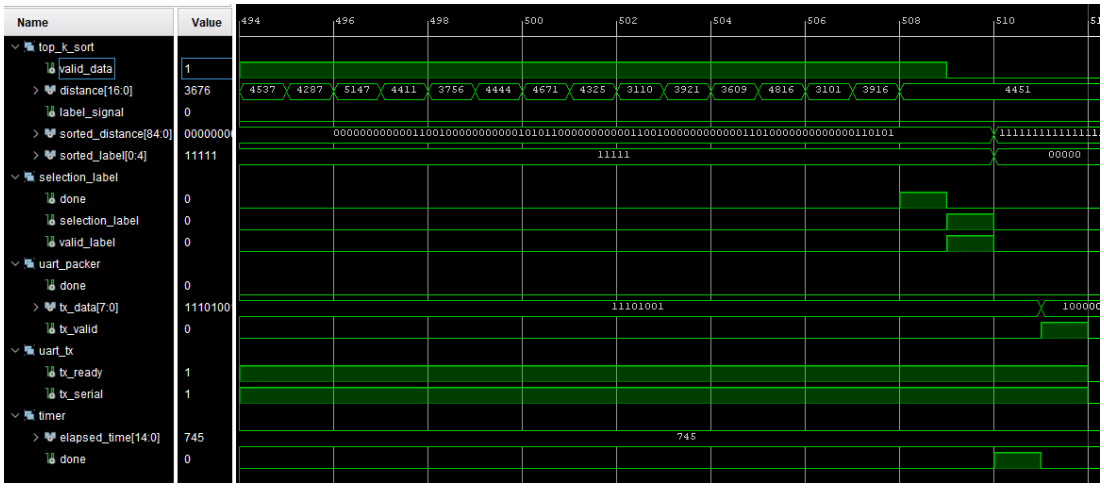


Fig. 4.8: Captura ILA durante la etapa final. En $S_{508}$ se activa `done` de `selection_label`, seguido en $S_{509}$ por `valid_label`. En $S_{510}$ finaliza el temporizador y en $S_{511}$ se inicia la transmisión de la palabra `1000001011101001`, correspondiente a 745 ciclos y clase 1.

### 4.1.3. Análisis del flujo UART

El canal UART mostró un funcionamiento estable bajo cargas prolongadas, procesando múltiples eventos sin pérdidas ni desincronizaciones. Un analizador lógico, conectado a los pines JA y JB de la placa, capturó las señales mediante la herramienta *WaveForms*, confirmando la correcta sincronización y codificación de las tramas. El sistema fue probado exitosamente hasta una velocidad de 460 800 baudios, sin evidenciar errores ni degradación en la transmisión. En tandas de 10, 20 y 30 eventos consecutivos, sin reinicios intermedios, se mantuvo un patrón constante: transmisión secuencial de PRL y PRH, procesamiento interno y retorno de un paquete de 16 bits con la clase (1 bit) y la latencia (15 bits). La Fig. 4.9 muestra ciclos repetitivos y ordenados, sin colisiones ni redundancias, lo que confirma la estabilidad del canal UART bajo distintas cargas de trabajo.

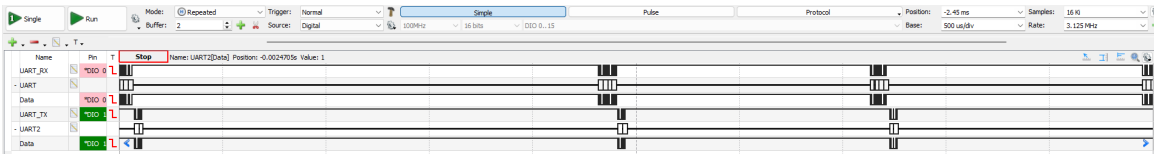

Fig. 4.9: Captura UART representando una tanda de clasificación continua. Se observan múltiples ciclos de transmisión y respuesta que siguen el mismo patrón estructural, confirmando el comportamiento estable del sistema embebido.

La Fig. 4.10 muestra un evento individual capturado durante la transmisión UART. Se observa la transmisión secuencial de PRL = 3291 y PRH = 4234, codificados como palabras de 16 bits, seguida del procesamiento y entrega de la respuesta. Esta corresponde a la palabra `0000001011101000`, que representa una latencia de 744 ciclos de reloj y la asignación de clase 0 en el bit más significativo, evidenciando la correcta ejecución del ciclo completo de transmisión y retorno de datos.

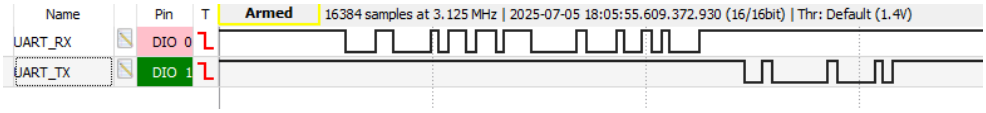


Fig. 4.10: Detalle de un evento de clasificación individual. Se muestra la estructura típica de transmisión UART: envío de PRL y PRH, procesamiento interno y respuesta. Este mismo patrón se repite durante toda la tanda.

La Fig. 4.11 muestra el análisis binario del flujo UART, donde se identifican las entradas `0000110011011011` (3291) y `0001000010001010` (4234), junto con la salida `0000001011101000`. Esta traza confirma la integridad en la codificación y la sincronización precisa entre envío, recepción y empaquetado de resultados.

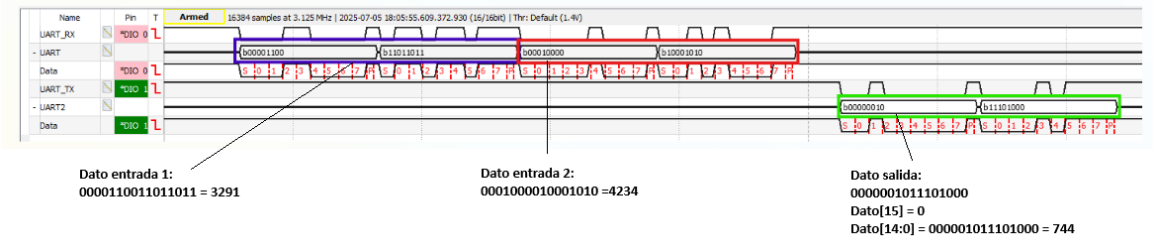


Fig. 4.11: Análisis detallado del flujo UART. Se identifican los datos de entrada (PRL y PRH) y la salida generada por la FPGA.

La Fig. 4.12 muestra la validación final realizada mediante la interfaz gráfica en Python, donde se reflejan los valores procesados y el tiempo de clasificación medido. Se reportan PRL = 32.91 % y PRH = 42.34 %, junto con un tiempo total de 0.65 ms (0.644 ms atribuibles a la latencia UART y 0.00744 ms al cómputo interno), coincidiendo con los ciclos observados en la captura hardware. Esta correspondencia refuerza la trazabilidad entre la ejecución interna y los datos entregados al usuario.

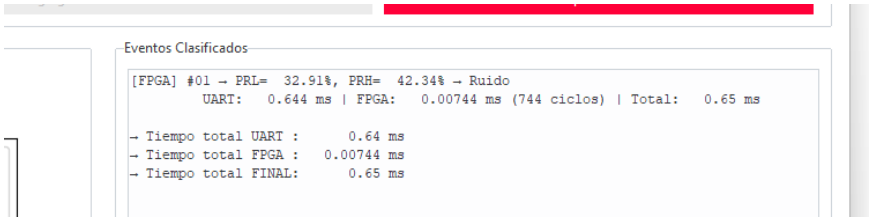


Fig. 4.12: Captura de la interfaz gráfica con el evento clasificado. Se confirman los valores de entrada, la clase asignada y los tiempos de respuesta medidos, los cuales concuerdan con la información registrada en la traza UART.

## 4.2. Evaluación de Desempeño del Sistema

Esta sección presenta los resultados obtenidos sobre el rendimiento del sistema en condiciones reales de operación, considerando métricas clave como latencia, utilización de recursos y estabilidad temporal. Los datos se derivan de trazas ILA, capturas UART, informes de síntesis y comparaciones con la referencia en software, evidenciando la capacidad del diseño para tareas de inferencia en tiempo real.

### 4.2.1. Rendimiento Temporal

La latencia interna del sistema, medida mediante el bloque `timer` descrito en la Sección 3.3.7, fue de 745 ciclos de reloj, equivalentes a 7,45 µs a una frecuencia de 100 MHz. Este tiempo corresponde al procesamiento completo de un evento sobre un conjunto de 742 elementos de entrenamiento y se mantuvo invariable en todas las pruebas, reflejando un comportamiento completamente determinista. Bajo estas condiciones, el throughput estimado fue de 134,228 eventos por segundo.

Al evaluar el sistema desde la interfaz Python, los tiempos por evento presentaron una mayor dispersión, producto de las latencias externas introducidas por el entorno de ejecución. En una secuencia de 20 eventos, los tiempos medidos oscilaron entre 0.31 ms y 1.24 ms, con una media de 0.804 ms y una desviación estándar de 0.306 ms, lo que corresponde a una variación relativa del 38 %. La Fig. 4.13 ilustra esta variación temporal y su distribución estadística.

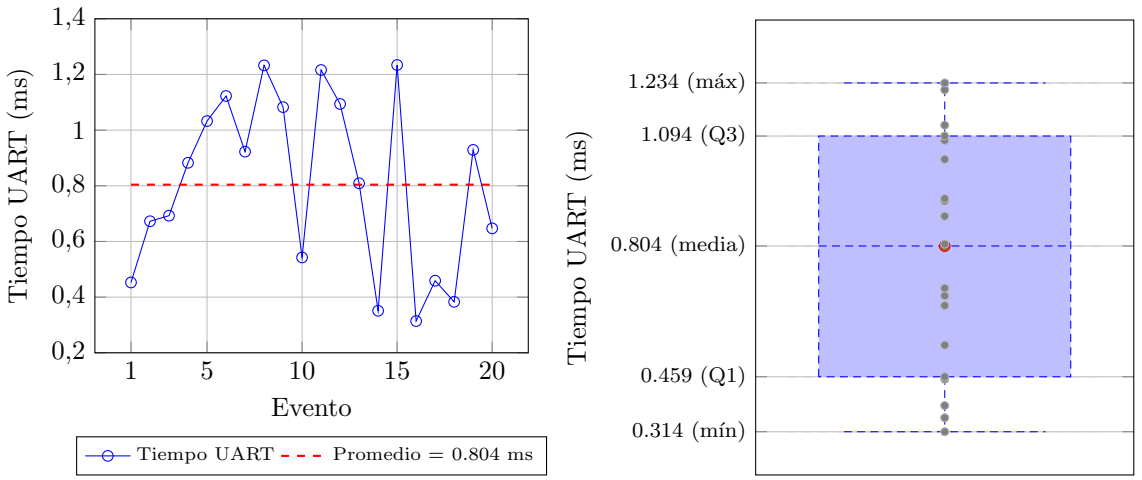


Fig. 4.13: Análisis del tiempo de transmisión UART en 20 eventos consecutivos: (izquierda) variación temporal por evento; (derecha) distribución estadística con media, cuartiles, extremos y puntos individuales.

Una validación más precisa se realizó mediante el analizador lógico externo, cuya captura se muestra en la Fig. 4.14. En ella se confirma que el ciclo completo por evento, incluyendo la transmisión UART de entrada, el procesamiento interno y la transmisión de salida, se completa en 136 µs de manera prácticamente estable. Dentro de este período, el intervalo correspondiente a la clasificación es de 7.45 µs, en concordancia exacta con el valor entregado por el temporizador `timer`.

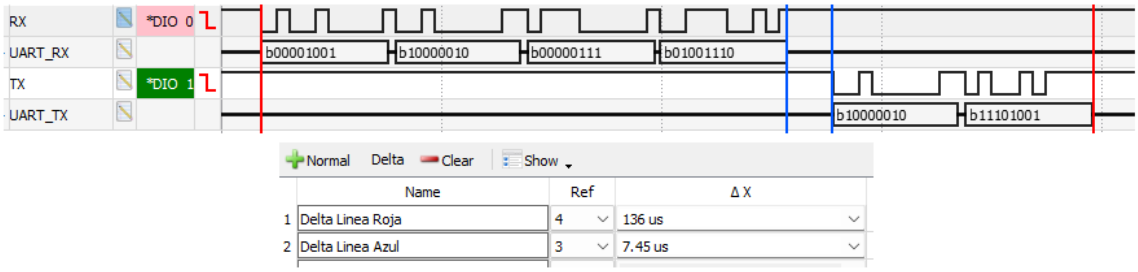


Fig. 4.14: Captura del analizador lógico externo durante un ciclo completo de clasificación. El proceso total, incluyendo transmisión UART de entrada, clasificación y respuesta, se completa en 136 µs. Entre ambas transmisiones se observa un intervalo de 7.45 µs correspondiente al tiempo exacto de inferencia, validado por el bloque `timer`.

Comparando estos resultados, se observa que la etapa de inferencia representa apenas un 5.5 % del tiempo total en hardware por evento. Si se contrasta con el valor promedio medido desde la interfaz Python (804 µs), esta proporción se reduce a solo un 0.93 %. Incluso considerando todo el procesamiento embebido, incluidas las transmisiones UART, el sistema ocupa solo un 16.9 % del tiempo total. El 83.1 % restante corresponde a sobrecargas introducidas por la interfaz software, el sistema operativo y la comunicación con el entorno de prueba, confirmando que las principales limitaciones temporales no provienen del diseño embebido.

### 4.2.2. Comparación con ejecución en software

Para evaluar el desempeño del sistema, se contrastaron los tiempos de procesamiento de la clasificación en FPGA con los obtenidos en un computador convencional. La Tabla 4.1 resume los resultados para distintos tamaños de muestra, diferenciando tres componentes en la ejecución embebida: el tiempo del núcleo de inferencia ($CORE$), la latencia de transmisión UART ($UART$) y el total en hardware ($HW$). También se incluye el tiempo requerido en software ($SW$) y dos razones de aceleración: $R_1$, que relaciona el tiempo total en FPGA frente a software, y $R_2$, que compara solo el núcleo de inferencia con su contraparte en software.

Tabla 4.1: Comparación de tiempos entre implementación en FPGA y software (en milisegundos).

| *Eventos* | CORE [ms] | UART [ms] | HW [ms] | SW [ms] | SW/HW $R_1$ | SW/CORE $R_2$ |
|---|---|---|---|---|---|---|
| 1 | 0.0075 | 1.13 | 1.1375 | 2.55 | 2.24 | 340.00 |
| 5 | 0.0373 | 3.21 | 3.2473 | 12.84 | 3.95 | 344.40 |
| 10 | 0.0745 | 6.97 | 7.0445 | 24.57 | 3.49 | 329.93 |
| 25 | 0.1863 | 20.31 | 20.4963 | 58.69 | 2.86 | 315.01 |
| 50 | 0.3725 | 34.94 | 35.3125 | 112.53 | 3.19 | 302.00 |
| 75 | 0.5588 | 55.83 | 56.3888 | 157.19 | 2.79 | 281.20 |
| 100 | 0.7450 | 73.28 | 74.0250 | 210.46 | 2.84 | 282.65 |
| Promedio | — | — | — | — | **3.05** | **313.46** |

Los resultados confirman la consistencia del sistema embebido: el tiempo por evento se mantiene fijo en 7.45 µs, como se verifica con los 74 500 ciclos registrados para 100 eventos. En contraste, la ejecución en software se degrada conforme aumenta el tamaño del conjunto, superando los 200 ms al clasificar 100 muestras.

Aún considerando la latencia de transmisión, la solución en FPGA reduce el tiempo promedio de procesamiento a un tercio respecto al software convencional ($R_1 = 3{,}05$). Al comparar exclusivamente el núcleo de inferencia, la aceleración asciende a más de 300 veces ($R_2 = 313{,}46$), evidenciando la eficiencia del procesamiento dedicado para tareas de inferencia en tiempo real.

### 4.2.3. Utilización de recursos

La Tabla 4.2 presenta la utilización global de recursos en la FPGA, considerando la totalidad del diseño implementado. Se incluyen las celdas lógicas utilizadas, específicamente los LUTs, FFs, BRAMs y pines I/O, junto con su porcentaje de ocupación respecto a los recursos disponibles en el dispositivo. No se reporta consumo de bloques DSP, ya que el diseño no realiza operaciones en punto flotante ni multiplicaciones complejas. El cálculo de distancias se basa exclusivamente en sumas y restas entre enteros, permitiendo una implementación completamente secuencial mediante lógica combinacional y registros.

Tabla 4.2: Utilización global de recursos en la FPGA

| Recurso | Utilización | Disponible | Utilización [%] |
|---|---|---|---|
| LUT | 4547 | 63400 | 7.17 |
| FF | 6908 | 126800 | 5.45 |
| BRAM | 56.5 | 135 | 41.85 |
| IO | 6 | 210 | 2.86 |

La Tabla 4.3 muestra el desglose por módulo lógico. Se observa que los núcleos `ila_selection`, `ila_rx` e `ila_distance` son los principales consumidores de recursos, acumulando más del 80 % del total de LUTs y más del 90 % del uso de BRAM. En contraste, los módulos funcionales del clasificador, como `manhattan_distance`, `top_k_sort` y `uart_tx`, presentan una utilización mínima, lo que evidencia la eficiencia estructural del diseño.

Tabla 4.3: Utilización de recursos por módulo (detalle)

| Módulo | LUTs | FFs | BRAM |
|---|---|---|---|
| *Bloques de entrada* | | | |
| uart_rx | 31 | 57 | 0.0 |
| sample_packer | 3 | 34 | 0.0 |
| global_reset | 19 | 40 | 0.0 |
| *Bloques de procesamiento* | | | |
| counter | 11 | 15 | 0.0 |
| brom_prl | 0 | 0 | 0.5 |
| brom_prh | 0 | 0 | 0.5 |
| brom_label | 0 | 0 | 0.5 |
| manhattan_distance | 63 | 19 | 0.0 |
| *Bloques de salida* | | | |
| top_k_sort | 129 | 90 | 0.0 |
| selection_label | 2 | 1 | 0.0 |
| timer | 6 | 34 | 0.0 |
| uart_packer | 8 | 29 | 0.0 |
| uart_tx | 24 | 32 | 0.0 |
| *Bloques auxiliares de depuración* | | | |
| ila_distance | 1115 | 1617 | 48.0 |
| ila_rx | 1313 | 1973 | 3.0 |
| ila_selection | 1592 | 2335 | 4.0 |
| dbg_hub | 231 | 632 | 1.0 |
| **Totales** | **4547** | **6908** | **57.5** |

Cabe destacar que los bloques de depuración fueron incluidos exclusivamente con fines de validación interna mediante ILA. Por tanto, en una versión final de producción podrían eliminarse sin afectar el funcionamiento, lo que permitiría liberar una proporción considerable de los recursos ocupados. Bajo este escenario, la lógica principal del clasificador representa solo una pequeña fracción del total, lo que deja en evidencia el amplio margen para escalar el sistema o incorporar nuevos módulos. Se aprecia además que la utilización lógica se concentra en los cuadrantes centrales del dispositivo (X0Y1, X0Y2, X1Y1 y X1Y2), donde se ubican tanto los módulos funcionales como los bloques ila. Las regiones periféricas permanecen mayormente vacías, lo que facilita el enrutamiento y reduce la latencia entre bloques.

Finalmente, desde una perspectiva de implementación, la distribución espacial validada por el floorplan confirma una asignación automática eficiente, sin zonas de congestión ni rutas críticas prolongadas. No fue necesario aplicar restricciones físicas adicionales para cumplir con el temporizado, lo que demuestra que el diseño es robusto, portátil y escalable.

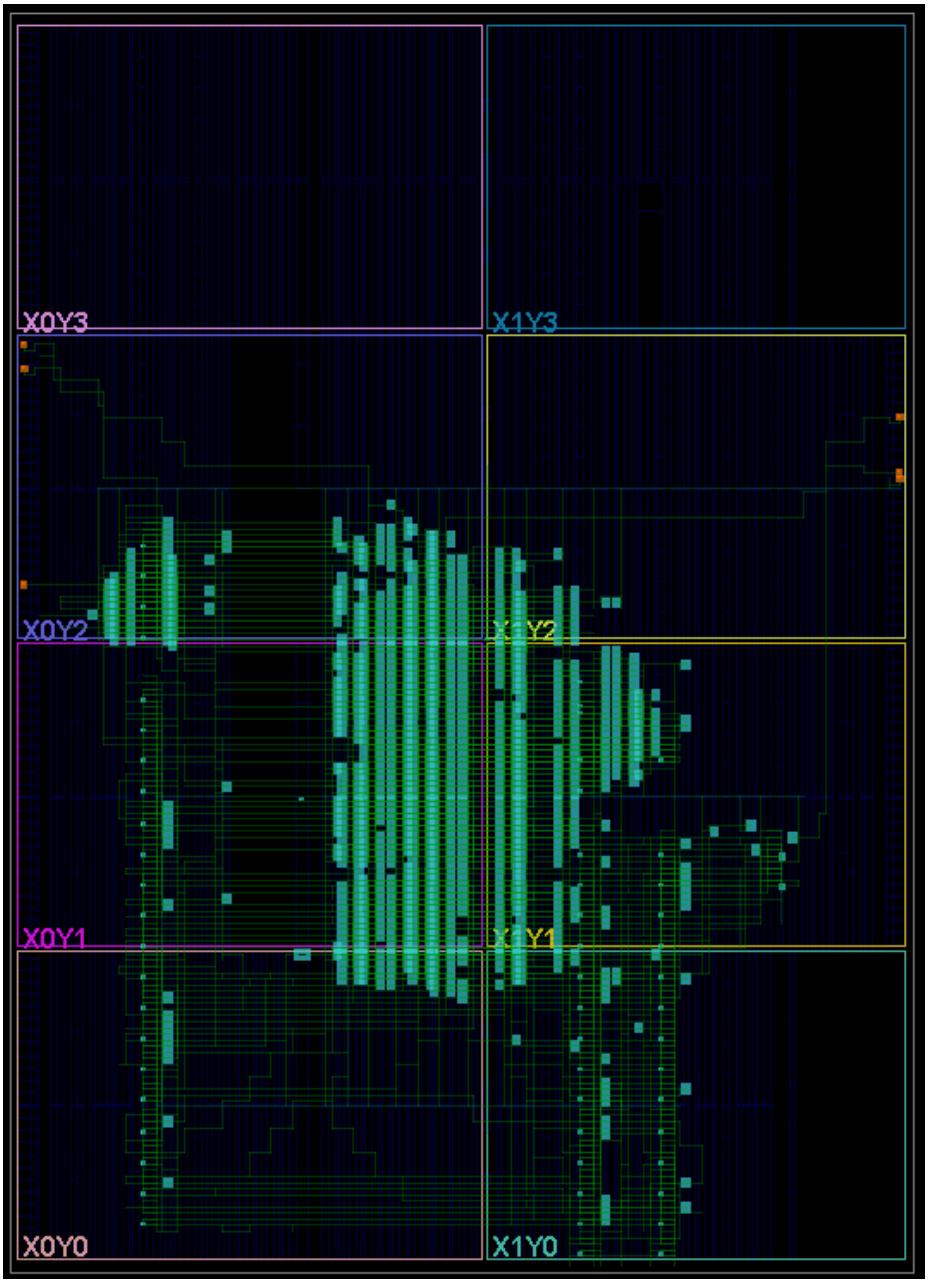


Fig. 4.15: Distribución espacial de la lógica implementada sobre la FPGA, concentrada principalmente en la zona central.

### 4.2.4. Análisis de Tiempo y Consumo de Energía

El análisis de temporización confirma que el diseño cumple todos los requisitos establecidos, sin presencia de *slack* negativo ni violaciones de tiempo. La Tabla 4.4 resume los valores registrados en la implementación final.

Tabla 4.4: Resumen de temporización

| | **WNS** | **WHS** | **WBSS** |
|---|---|---|---|
| Valor (ns) | 0,404 | 0,037 | 9,126 |

La ruta crítica se origina en la salida del módulo `manhattan_distance` y culmina en la lógica de inserción del bloque `top_k_sort`, concentrando en este trayecto el principal retardo del sistema. Esta trayectoria atraviesa varios niveles de lógica secuencial y constituye el punto más restrictivo en términos temporales, como se ilustra en la Fig. 4.16.

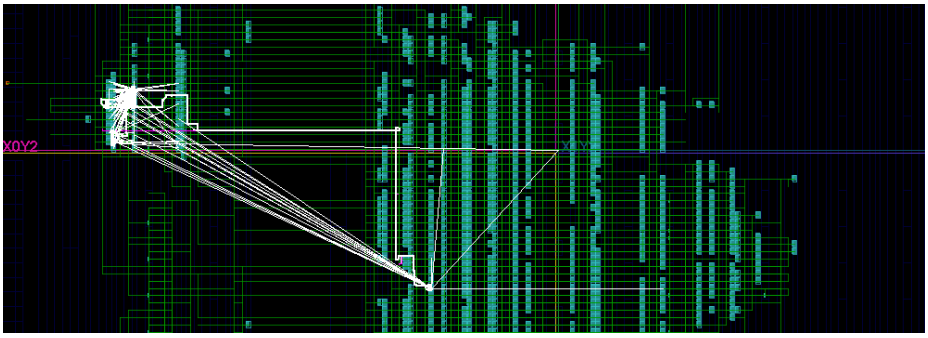


Fig. 4.16: Ruta crítica entre los módulos `manhattan_distance` y `top_k_sort`.

En cuanto al consumo energético, la potencia total estimada asciende a 0,159 W, con un 63 % correspondiente al componente estático y un 37 % al componente dinámico. La Fig. 4.17 muestra el desglose dinámico, donde los dominios de reloj concentran el mayor consumo (50 %), seguidos por los bloques de BRAM (19 %), señales internas (17 %), lógica (12 %) y entradas/salidas (2 %). La temperatura promedio del sistema fue de 25,7 °C durante la estimación.

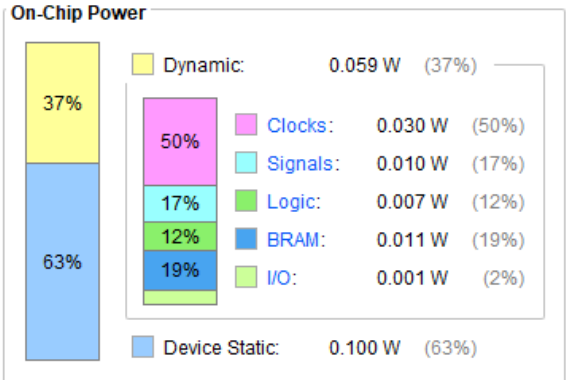


Fig. 4.17: Distribución del consumo dinámico y estático estimado por Vivado.

### 4.2.5. Pruebas límite

Se diseñaron tres pruebas límite enfocadas exclusivamente en el comportamiento interno de la FPGA, con el fin de validar su carácter determinista, escalabilidad y robustez estructural.

**Determinismo del sistema** La primera prueba tuvo como objetivo validar el comportamiento determinista del sistema al modificar el parámetro `max_addr`, el cual define la cantidad de direcciones emitidas por el bloque `counter` hacia las memorias BROM. Esta evaluación permitió confirmar que, ante una cantidad fija de datos, el sistema ejecuta la clasificación de forma estable, repetible y completamente independiente del contenido almacenado o de condiciones externas.

El experimento se realizó con un valor fijo de $k = 5$, verificando el comportamiento mediante simulaciones en software y análisis en hardware utilizando los analizadores `ila_distance` e `ila_selection`. Como se muestra en la Tabla 4.5, la latencia aumentó de manera lineal con la cantidad de datos, cumpliendo consistentemente la relación *latencia (ciclos) = datos + 3*. Esta tendencia se mantuvo invariable incluso para conjuntos mayores, validando el comportamiento determinista del sistema. Adicionalmente, al incrementar la cantidad de datos desde 10 hasta 742 eventos, el throughput teórico disminuyó desde 7,69 millones hasta 134 mil eventos por segundo, evidenciando la relación inversa esperada.

Tabla 4.5: Resultados al variar `max_addr` con $k = 5$

| `max_addr` | Datos utilizados | Latencia (ciclos) | Throughput (eventos/s) |
|---|---|---|---|
| 10 | 10 | 13 | 7 692 308 |
| 100 | 100 | 103 | 970 874 |
| 250 | 250 | 253 | 394 472 |
| 500 | 500 | 503 | 198 412 |
| 742 | 742 | 745 | 134 228 |

**Escalabilidad lógica: impacto del parámetro $k$** Esta prueba tuvo como objetivo determinar el valor máximo admisible del parámetro $k$ sin comprometer la integridad temporal del sistema. Para ello, se evaluó el bloque `top_k_sort` variando la cantidad de vecinos, manteniendo constante la base de datos de entrenamiento en 742 eventos.

El uso de recursos y los márgenes de temporización fueron obtenidos directamente de los reportes de implementación generados por Vivado. La frecuencia máxima alcanzable para cada configuración se calculó como:

$$F_{\text{máx}}\ (\text{MHz}) = \frac{1000}{10\,\text{ns} - \text{WNS}} \tag{4.2.1}$$

En la Tabla 4.6 se observa que a medida que $k$ aumenta, se incrementa el uso de LUTs y FFs, mientras que el uso de BRAM permanece constante. Desde $k = 75$ aparecen márgenes negativos, y a partir de $k = 200$ se evidencia un deterioro crítico en temporización. Se concluye que $k = 200$ representa el límite funcional superior recomendable para este bloque sin afectar la estabilidad ni la frecuencia de operación.

Tabla 4.6: Recursos y temporización variando $k$ (742 eventos, sin bloques `ila`)

| $k$ | LUTs | FFs | BRAMs | WNS (ns) | TNS (ns) | $F_{\text{máx}}$ (MHz) |
|---|---|---|---|---|---|---|
| 50 | 1541 | 1122 | 2.5 | 0.227 | 0.000 | 102.3 |
| 75 | 2381 | 1572 | 2.5 | -0.173 | -0.765 | 98.3 |
| 100 | 3139 | 2022 | 2.5 | -0.379 | -16.314 | 96.3 |
| 150 | 4816 | 2922 | 2.5 | -0.131 | -3.565 | 98.7 |
| 200 | 8064 | 3822 | 2.5 | -1.501 | -984.781 | 86.9 |

**Límite físico de almacenamiento en BRAM** Finalmente, se evaluó la cantidad máxima de eventos de entrenamiento que puede almacenar la FPGA Nexys A7-100T, la cual dispone de 135 bloques BRAM36 (memorias internas de 36 Kb). Para ello, se incrementó progresivamente la base de datos con eventos aleatorios de 16 bits generados en Python, manteniendo fijo $k = 5$, sin modificar la arquitectura ni comprometer la temporización.

La Tabla 4.7 presenta el uso de bloques BRAM por tipo de dato: espectros `prl`, `prh` y etiquetas `label`. Al llegar a 138 000 eventos, se ocupan 132,5 bloques BRAM36, lo que equivale al 98,1 % de la capacidad disponible. Superar dicha cantidad implicaría sobrepasar el límite físico de la FPGA.

Tabla 4.7: Uso de bloques BRAM36 según cantidad de eventos

| Eventos | `prl` | `prh` | `label` | Total BRAM36 | Utilización ( %) |
|---|---|---|---|---|---|
| 150 000 | 67 | 67 | 4,5 | 138,5 | 102,6 |
| 145 000 | 67 | 67 | 4,5 | 138,5 | 102,6 |
| 140 000 | 65,5 | 65,5 | 4,5 | 135,5 | 100,4 |
| 138 000 | 64 | 64 | 4,5 | 132,5 | 98,1 |

En conjunto, estas pruebas permitieron validar el comportamiento del sistema ante configuraciones extremas, evidenciando márgenes reales de escalabilidad sin necesidad de modificar la arquitectura base ni comprometer el cierre temporal del diseño.

Capítulo 5

# CONCLUSIONES

# REFERENCIAS


[1] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660, 2013.

[2] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela Hung Byers. Big data: The next frontier for innovation, competition, and productivity. Technical report, McKinsey Global Institute, 2011.

[3] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, 2019.

[4] M. Mitchell Waldrop. The chips are down for moore's law. *Nature*, 530:144–147, 2016.

[5] Lukáš Kohútka and Ján Mach. A new fpga-based task scheduler for real-time systems. *Electronics*, 12(8):1870, 2023.

[6] Katherine Compton and Scott Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, 2002.

[7] Peter Marwedel. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems*. Springer, 3 edition, 2021.

[8] William Derigent, Olivier Cardin, and Damien Trentesaux. Industry 4.0: contributions of holonic manufacturing control architectures and future challenges. *Journal of Intelligent Manufacturing*, 32(4):971–989, 2020.

[9] Shanhe Yi, Cheng Li, and Qun Li. A survey of fog computing: Concepts, applications and issues. In *Mobidata '15: Proceedings of the 2015 Workshop on Mobile Big Data*, pages 37–42. ACM, 2015.

[10] Vivienne Sze, Yu-Hsin Chen, Joel Emer, Amr Suleiman, and Zhengdong Zhang. Hardware for machine learning: Challenges and opportunities. In *IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–8, 2017.

[11] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[12] Nahin Ul Sadad, Afsana Afrin, and Md. Nazrul Islam Mondal. Binary classification using k-nearest neighbor algorithm on fpga. In *2021 International Conference on Computer, Communication, Chemical, Materials and Electronic Engineering (IC4ME2)*, pages 1–4, 2021.

[13] Saeid Gorgin, Mohammad Hosein Gholamrezaei, Danial Javaheri, and Jeong-A Lee. An efficient fpga implementation of k-nearest neighbors via online arithmetic. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–2, 2022.

[14] Cristina Silvano, Gianluca Palermo, Vittorio Zaccaria, et al. A survey on deep learning hardware accelerators for heterogeneous hpc platforms. *ACM Computing Surveys*, 2023.

[15] Shahanur Alam, Chris Yakopcic, Qing Wu, Mark Barnell, Simon Khan, and Tarek M. Taha. Survey of deep learning accelerators for edge and emerging computing. *Electronics*, 13(15):2988, 2024.

[16] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[17] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12. ACM, 2017.

[18] Sen M. Kuo and Woon-Seng Gan. *Digital Signal Processors: Architectures, Implementations, and Applications*. Prentice Hall, 2005.

[19] GeeksforGeeks. Architecture of soc, 2024.

[20] AMD (Xilinx). *7 Series FPGAs Configurable Logic Block User Guide*, 2024. UG474 (v1.9).

[21] Enzo Rucci. *Evaluación de rendimiento y eficiencia energética de sistemas heterogéneos para bioinformática*. 01 2018.

[22] AMD (Xilinx). *Vivado Design Suite User Guide: Design Flows Overview*, 2024. UG892 (v2024.1).

[23] Texas Instruments. *Interface Circuits for TIA/EIA-232-F*, 1997.

[24] Introduction to spi interface, 2020. Accessed: 2025-07-21.

[25] NXP Semiconductors. *I2C-bus specification and user manual*, 2014.

[26] PCI-SIG. *PCI Express Base Specification Revision 4.0*, 2017.

[27] ARM. *AMBA AXI and ACE Protocol Specification*, 2011.

[28] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[29] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, New York, 2006.

[30] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2 edition, 2009.

[31] Naveen Kumar Dumpala, Shivukumar B. Patil, Daniel Holcomb, and Russell Tessier. Loop unrolling for energy efficiency in low-cost field-programmable gate arrays. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 11(4):1–23, 2019.

[32] Roque Alfredo Osornio-Rios, Isaias Cueva-Perez, Alvaro Ivan Alvarado-Hernandez, Larisa Dunai, Israel Zamudio-Ramirez, and Jose Alfonso Antonino-Daviu. Fpga-microprocessor based sensor for faults detection in induction motors using time-frequency and machine learning methods. *Sensors*, 24(8):2653, 2024.

[33] P. Jegadeeshwari, T. Anitha, and M. Arulaalan. Fpga implementation of knn algorithm-based prediction for dpwm methods. *Journal of Propulsion Technology*, 44(3):2456–2464, 2023.

[34] Achmad Rizal, Sugondo Hadiyoso, and Ahmad Zaky Ramdani. Fpga-based implementation for real-time epileptic eeg classification using hjorth descriptor and knn. *Electronics*, 11(19):3026, 2022.

[35] Antonio Borelli, Fanny Spagnolo, Raffaele Gravina, and Fabio Frustaci. An fpga-based hardware accelerator for the k-nearest neighbor algorithm implementation in wearable embedded systems. In Mufti Mahmud, Cosimo Ieracitano, M. Shamim Kaiser, Nadia Mammone, and Francesco Carlo Morabito, editors, *Applied Intelligence and Informatics*, volume 1724 of *Communications in Computer and Information Science*, pages 44–56. Springer, 2022.

[36] Chengliang Wang, Zhetong Huang, Ao Ren, and Xun Zhang. An fpga-based knn search accelerator for point cloud registration. In *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2024.

[37] Muhammad Umair Mohsin, Saeed Kazmi, Nabeel Aslam, Amir Zahoor, Adel Almogren, and Shahid Malik. Accelerated homomorphic inference using ckks and fpga. In *Proceedings of the 2024 IEEE International Conference on Computer Design (ICCD)*, pages 153–160. IEEE, 2024.

[38] David Marquez-Viloria, Luis Castano-Londono, and Neil Guerrero-Gonzalez. A modified knn algorithm for high-performance computing on fpga of real-time m-qam demodulators. *Electronics*, 10(5):627, 2021.

[39] Sagarika Behera and Jhansi Rani Prathuri. Fpga-based acceleration of k-nearest neighbor algorithm on fully homomorphic encrypted data. *Cryptography*, 8(1):8, 2024.

[40] K. Huang. K-means parallelism on fpga. Master's thesis, Northeastern University, 2017. "However, the Manhattan-distance could reduce the hardware resource use and runs faster in hardware." Retrieved from https://coe.northeastern.edu/research/rcl/theses/huang-ms2017.pdf.

[41] Sina Bundschuh, Jan Kunze, and Klaus-Dieter Kuhnert. Implementation of an fpga-based system to process images and match keypoints on high-resolution pictures. *Electronics*, 13(23):4774, 2024.