

# TRABAJO FIN DE MÁSTER

*Sistema de reconocimiento de señales de  
tráfico*

**MÁSTER EN SISTEMAS ELECTRÓNICOS PARA  
ENTORNOS INTELIGENTES**

Departamento de Tecnología Electrónica  
Escuela Técnica Superior de Ingeniería de Telecomunicación  
Universidad de Málaga

Manuel Sánchez Natera,  
Málaga, 2025



## ***Sistema de reconocimiento de señales de tráfico***

**Autor:** Manuel Sánchez Natera

**Tutor:** Martín González García

**Departamento:** Tecnología Electrónica

**Titulación:** Máster en sistemas electrónicos para entornos inteligentes

## Resumen

Este proyecto desarrolla un sistema basado en FPGA para la detección y reconocimiento de señales de tráfico de límite de velocidad, combinando hardware dedicado con una aplicación de escritorio en PC. El objetivo ha sido integrar la captura de imágenes en tiempo real con el procesamiento mediante un modelo de *Machine Learning* entrenado específicamente para la detección de señales.

Como resultado, se ha implementado una solución funcional y modular, capaz de operar en tiempo real y adecuada para entornos académicos, simuladores de conducción y con potencial aplicación en futuros sistemas de asistencia avanzada a la conducción (ADAS).

## ***Sistema de reconocimiento de señales de tráfico***

**Author:** Manuel Sánchez Natera

**Supervisor:** Martín González García

**Department:** Electronic Technology

**Degree:** Master in electronic systems for smart environments

## **Abstract**

This project develops a system based on FPGA for the detection and recognition of traffic signs related to speed limits, combining dedicated hardware with a desktop application for PC. The main objective has been to integrate real-time image acquisition with processing through a *Machine Learning* model specifically trained for traffic sign detection.

As a result, a functional and modular solution has been implemented, capable of operating in real time and suitable for academic environments, driving simulators, and with potential application in future Advanced Driver-Assistance Systems (ADAS).



*A mi familia y a mi novia por su apoyo, cariño y paciencia infinita. En especial a mi Antoñita, que sé que, donde estés, estás haciendo de él un lugar mejor.*



## Tabla de contenido

Resumen .....	3
Abstract .....	4
Tabla de contenido .....	7
Lista de figuras .....	9
Lista de tablas .....	11
Lista de acrónimos .....	12
Capítulo 1 Requisitos y casos de uso .....	13
1.1    Introducción .....	13
1.2    Estado de la tecnología .....	13
1.3    Objetivos .....	14
1.4    Requisitos del proyecto .....	15
1.4.1 Diagrama de requisitos .....	15
1.4.2 Requisitos No Funcionales .....	16
1.4.3 Requisitos Funcionales .....	18
1.5    Casos de uso .....	20
Capítulo 2 Descripción del Sistema .....	23
2.1    Descripción General .....	23
2.2    Descripción Software .....	24
2.3    Descripción Hardware .....	25
2.3.1 Módulo UM232H-B .....	25
2.3.2 Sensor MT9V111 .....	25
Capítulo 3 Diseño e Implementación .....	27
3.1    Módulo Hardware .....	27
3.1.1 Introducción .....	27
3.1.2 Placa de desarrollo BASYS 3 .....	27
3.1.3 Interfaz USB de conexión con PC .....	29
3.1.4 Sensor de imagen .....	30
3.1.5 Bloques Funcionales .....	32
3.1.6 Asignación de pines de E/S .....	53
3.2    Módulo Software .....	56
3.2.1 Introducción .....	56
3.2.2 Diagrama General .....	57
3.2.3 Arquitectura basada en clases .....	58
3.2.4 Modelo Neuronal .....	67

Capítulo 4 Pruebas y validación .....	77
4.1 Módulo Hardware .....	77
4.1.1 FT245_IF .....	79
4.1.2 DISP7SEG .....	81
4.1.3 CLOCKGEN .....	82
4.1.4 IMAGE_PIPELINE .....	83
4.1.5 TOP .....	88
4.2 Módulo Software.....	91
4.2.1 Interfaz de usuario en modo DEBUG .....	91
4.2.2 Modelo neuronal con imágenes estáticas .....	92
4.2.3 Recepción de imágenes.....	93
4.2.4 Modelo neuronal con imágenes reales.....	97
Capítulo 5 Conclusiones y líneas futuras.....	99
Capítulo 6 Bibliografía .....	101
Anexo A. Repositorio en GitHub .....	103
Anexo B. Instalación del software.....	104

# Lista de figuras

Figura 1.1 Diagrama de requisitos .....	15
Figura 1.2 Diagrama de casos de uso .....	20
Figura 2.1 Arquitectura general del sistema.....	23
Figura 3.1 Pines PMOD placa de desarrollo BASYS 3.....	27
Figura 3.2 Segmentos display 7-segmentos .....	28
Figura 3.3 Conexiones del display.....	28
Figura 3.4 Módulo UM232H-B.....	29
Figura 3.5 Configuración FTDI desde FTProg [17].....	30
Figura 3.6 Arquitectura hardware del sistema .....	32
Figura 3.7 Circuito sincronizador 2-FF .....	33
Figura 3.8 Arquitectura bloque FT245 TRANSCEIVER.....	34
Figura 3.9 Interfaz FIFO asíncrona FT245 – señales ciclo de escritura .....	36
Figura 3.10 Interfaz FIFO asíncrona FT245 – señales ciclo de lectura .....	36
Figura 3.11 Diagrama de estados de IFWRITE .....	38
Figura 3.12 Diagrama de estados de IFREAD .....	40
Figura 3.13 Arquitectura bloque CLOCK GENERATOR .....	41
Figura 3.14 Arquitectura bloque DISPLAY 7 SEGMENTOS .....	42
Figura 3.15 Arquitectura bloque IMAGE PIPELINE .....	43
Figura 3.16 Reporte utilización FPGA post-implementación en Vivado.....	44
Figura 3.17 Arquitectura e implementación componente FIFO .....	45
Figura 3.18 Delays de propagación de PIXCLK y DOUT.....	46
Figura 3.19 Diagrama temporización de datos de salida MT9V111 .....	47
Figura 3.20 Diagrama de estados de MT9V111_IF.....	49
Figura 3.21 Temporización señales de control de frame .....	50
Figura 3.22 Definiciones en componente SENSOR EMU en VHDL .....	51
Figura 3.23 Diagrama de estados de SENSOR EMULATOR .....	52
Figura 3.24 Reporte utilización FPGA post-implementación .....	55
Figura 3.25 Diagrama de flujo completo de la aplicación .....	57
Figura 3.26 Diagrama de flujo de la clase View .....	60
Figura 3.27 Diagrama de flujo de clase Controller .....	63
Figura 3.28 Diagrama de flujo de clase Image Processor .....	65
Figura 3.29 Archivo yolo_dataset.yaml.....	68
Figura 3.30 Parte del código del archivo train_yolo.py .....	69
Figura 3.31 Resultados entrenamiento del modelo .....	70
Figura 3.32 Matriz de confusión del modelo .....	71
Figura 3.33 Tipos de modelos.....	72
Figura 3.34 Estructura de la red neuronal .....	74
Figura 4.1 Vista de la simulación en Vivado .....	77
Figura 4.2 Simulación ciclo de escritura de IF_WRITE .....	79

Figura 4.3 Simulación ciclo de lectura de IF_READ.....	80
Figura 4.4 Simulación visualización de valores en DISP7SEG .....	82
Figura 4.5 Simulación de generación de reloj 25MHz con CLOCKGEN.....	82
Figura 4.6 Simulación de generación de reloj 12.5MHz con CLOCKGEN.....	83
Figura 4.7 Simulación funcionamiento de FIFO .....	84
Figura 4.8 Simulación captura de imágenes con SENSOR_EMU (parte 1). ....	85
Figura 4.9 Simulación captura de imágenes con SENSOR_EMU (parte 2). ....	86
Figura 4.10 Simulación captura de imágenes con SENSOR_EMU (parte 3). ....	86
Figura 4.11 Simulación captura de imágenes con SENSOR_EMU (parte 4). ....	87
Figura 4.12 Simulación captura de imágenes con SENSOR_EMU (parte 5). ....	87
Figura 4.13 Simulación sistema completo (TOP) a 25MHz y frame a color.....	88
Figura 4.14 Simulación sistema completo (TOP) a 25MHz y frame en escala de grises. ....	89
Figura 4.15 Simulación sistema completo (TOP) a 12.5MHz y frame a color.....	89
Figura 4.16 Simulación sistema completo (TOP) al recibir datos desde PC.....	90
Figura 4.17 Vista interfaz de usuario DEBUG comunicación con FPGA.....	91
Figura 4.18 Vista interfaz de usuario DEBUG tras reconocimiento de señal.....	92
Figura 4.19 Envío imágenes sintéticas a color al PC .....	93
Figura 4.20 Envío imágenes sintéticas en escala de grises al PC .....	94
Figura 4.21 Envío de imágenes reales a color al PC.....	95
Figura 4.22 Envío de imágenes reales en escala de grises al PC .....	96
Figura 4.23 Detección y reconocimiento de señal de tráfico en tiempo real.....	97
Figura 0.1 Proceso de instalación del software .....	104
Figura 0.2 Vista inicial de la aplicación .....	105

# Lista de tablas

Tabla 1.1 Requisitos No Funcionales del proyecto .....	16
Tabla 1.2 Requisitos Funcionales del proyecto.....	18
Tabla 1.3 Actores externos .....	20
Tabla 3.1 Requisitos de temporización FIFO asíncrona .....	36
Tabla 3.2 Estados y señales de IFWRITE .....	38
Tabla 3.3 Estados y señales de IFREAD .....	40
Tabla 3.4 Estados y señales de MT9V111_IF .....	49
Tabla 3.5 Tiempos de frame .....	51
Tabla 3.6 Estados y señales de SENSOR EMULATOR.....	52
Tabla 3.7 Asignación de pines PMOD y señales internas del diseño.....	53

# Lista de acrónimos

- ADAS** – Advanced Driver-Assistance Systems
- APS** – Active Pixel Sensor
- BRAM** – Block RAM
- CMOS** – Complementary Metal-Oxide-Semiconductor
- CPU** – Central Processing Unit
- DP** – Decimal Point
- EEPROM** – Electrically Erasable Programmable Read-Only Memory
- FIFO** – First In, First Out
- FPS** – Frames Per Second
- FPGA** – Field-Programmable Gate Array
- FSM** – Finite State Machine
- GB** – Gigabyte
- Gb** – Gigabit
- GPIO** – General-Purpose Input/Output
- GUI** – Graphical User Interface
- HDL** – Hardware Description Language
- IOB** – Input/Output Block
- KB** – Kilobyte
- kBps** – Kilobytes per second
- LDO** – Low Dropout Regulator
- LUT** – Look-Up Table
- LUTRAM** – Look-Up Table RAM
- MB** – Megabyte
- Mb** – Megabit
- MHz** – Megahertz
- mAP** – mean Average Precision
- MVC** – Model-View-Controller
- NPU** – Neural Processing Unit
- PC** – Personal Computer
- PLL** – Phase-Locked Loop
- PMOD** – Peripheral Module
- QSPI** – Quad Serial Peripheral Interface
- RAM** – Random Access Memory
- RGB** – Red, Green, Blue
- SoC** – System on Chip
- UART** – Universal Asynchronous Receiver-Transmitter
- USB** – Universal Serial Bus
- VGA** – Video Graphics Array
- YOLO** – You Only Look Once

# Capítulo 1 Requisitos y casos de uso

## 1.1 Introducción

El objetivo de este capítulo es extraer, estudiar y definir las necesidades del sistema de reconocimiento de señales de tráfico. Se detallarán los requisitos del usuario final y cómo se ha realizado el proyecto para cumplirlos.

## 1.2 Estado de la tecnología

En la última década, los sistemas de asistencia avanzada a la conducción (*Advanced Driver-Assistance Systems, ADAS*) han incorporado de forma progresiva la detección y reconocimiento de señales de tráfico como una funcionalidad clave para aumentar la seguridad en carretera. Fabricantes como Tesla, BMW, Mercedes o Toyota entre otros, integran en sus vehículos cámaras frontales que, en combinación con algoritmos de visión artificial, permiten advertir al conductor de las limitaciones de velocidad o incluso ajustar automáticamente el control de crucero adaptativo.

La evolución tecnológica ha conducido a que, actualmente, los modelos de *deep learning* basados en detección en tiempo real como YOLO (*You Only Look Once*), en sus diferentes versiones (YOLOv3, YOLOv5, YOLOv8), se hayan convertido en el estándar de facto para tareas de detección de objetos en entornos de conducción. Estos modelos ofrecen un equilibrio muy adecuado entre precisión y velocidad de inferencia, permitiendo su integración en sistemas embebidos de bajo consumo [1].

Sin embargo, la implementación práctica en vehículos comerciales suele realizarse sobre SoCs especializados (por ejemplo, NVIDIA Jetson) o procesadores de propósito general, lo que limita su eficiencia energética en escenarios donde se requiere un procesamiento en tiempo real y continuo. En este sentido, las FPGA (*Field-Programmable Gate Array*) representan una alternativa competitiva al permitir paralelizar operaciones, reducir la latencia y adaptar la arquitectura hardware a las necesidades específicas del sistema [2]. La combinación de FPGA para la adquisición y preprocesamiento de imágenes junto con un PC (*Personal Computer*) para la inferencia del modelo neuronal constituye un enfoque híbrido que aprovecha las ventajas de ambos mundos.

La motivación de este proyecto se centra precisamente en ese enfoque híbrido. Por un lado, la FPGA permite capturar y transmitir imágenes en tiempo real desde un sensor digital, garantizando un flujo de datos continuo y eficiente. Por otro, un modelo YOLOv8 preentrenado y posteriormente ajustado (fine-tuning) procesa las imágenes en un PC de manera rápida y con alta fiabilidad. Este planteamiento lo hace especialmente adecuado para aplicaciones en simuladores de conducción, entornos académicos y como paso intermedio para futuras implementaciones en ADAS reales, donde los requisitos de robustez y latencia son críticos.

### 1.3 Objetivos

Este proyecto tiene como finalidad principal el diseño de un sistema, que, a partir de una serie de componentes hardware y de una aplicación software de escritorio, sea capaz de mostrar imágenes en tiempo real, detectando y reconociendo las señales de tráfico correspondientes a la velocidad que aparecen.

## 1.4 Requisitos del proyecto

### 1.4.1 Diagrama de requisitos

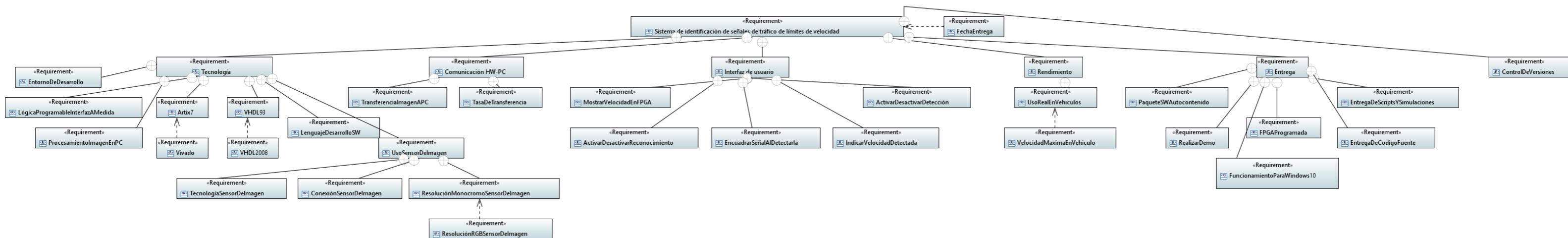


Figura 1.1 Diagrama de requisitos

Como requisitos principales, el sistema debe:

- Capturar imágenes al menos en formato VGA (*Video Graphics Array*), esto es, 680x480.
- Establecer comunicación del hardware con el PC.
- Alcanzar una tasa superior a 10FPS (*Frames Per Second*).
- Poder detener y reanudar la recepción de imágenes a petición del usuario desde una interfaz desde el PC.
- Detectar y reconocer las señales de tráfico relacionadas con los límites de velocidad que aparezcan en la imagen recibida.
- Ser autocontenido: el software debe ser portable y compatible para plataforma Windows 10 o superior. Así como, el hardware también se entregará al usuario completamente funcional, sin necesidad de manipularlo.

A continuación, se detallan otros requisitos derivados, que se dividen en: no funcionales y funcionales.

### 1.4.2 Requisitos No Funcionales

A continuación, se listan los requisitos no funcionales del proyecto.

ID	Nombre	Prioridad	Precedencia
R1.1	LógicaProgramableInterfazAMedida	Fundamental	-
R1.2	Artix7	Fundamental	R1.1
R1.3	Vivado	Fundamental	R1.1, R1.2
R1.4	VHDL93	Fundamental	R1.1
R1.5	VHDL2008	Opcional	R1.1, R1.4
R2.1	ProcesamientoImagenEnPC	Fundamental	-
R2.2	LenguajeDesarrolloSW	Deseable	R2.1
R2.3	FuncionamientoParaWindows10	Fundamental	R2.1
R2.4	EntornoDeDesarrollo	Opcional	R2.1
R8	PaqueteSWAutocontenido	Fundamental	-
R9	FPGAProgramada	Fundamental	-
R10	EntregaDeScriptsSimulaciones	Fundamental	-
R11	EntregaDeCódigoFuente	Fundamental	-
R12	RealizarDemoDelSistema	Fundamental	-
R13	ControlDeVersiones	Fundamental	-
R14	FechaEntrega	Fundamental	-

Tabla 1.1 Requisitos No Funcionales del proyecto

#### R1.1 LógicaProgramableInterfazAMedida

Uso de lógica programable para realizar un interfaz a medida.

#### R1.2 Artix7

Se usará tecnología disponible y de bajo coste. Para ello, se empleará la placa de desarrollo BASYS3, que integra una FPGA Artix7-Xilinx.

#### R1.3 Vivado

El entorno para programar la FPGA (*Hardware Description Language, HDL*) y realizar las requeridas simulaciones será Vivado.

#### R1.4 VHDL93

Como mínimo se usará la versión 93' del lenguaje VHDL (*Very High Speed Integrated Circuit Hardware Description Language*).

#### R1.5 VHDL2008

Opcionalmente, se podrá emplear la versión 2008 del lenguaje VHDL para aprovechar características de una versión posterior.

## R2.1 ProcesamientoImagenEnPC

La unidad de procesamiento de imagen se implementará en el PC.

## R2.2 LenguajeDesarrolloSW

Preferiblemente, se desarrollará el software de la interfaz de usuario del PC en el lenguaje C/C++.

## R2.3 FuncionamientoParaWindows10

El software desarrollado debe garantizar su funcionamiento para Windows10.

## R2.4 EntornoDeDesarrollo

Uso de plataforma Qt para desarrollo de la interfaz de usuario. Está integrado con librerías OpenCV y FTDI (*Future Technology Devices International*).

## R8 PaqueteSWAutocontenido

Se debe realizar un paquete entregable autocontenido y autoejecutable a través de instalador. Abstrayendo al usuario de instalar dependencias.

## R9 FGAProgramada

La FPGA ya debe entregarse programada (*bitstream* en flash) para que el usuario únicamente tenga que alimentarla.

## R10 EntregaDeScriptsSimulaciones

Entrega de scripts de simulaciones compatibles con Vivado.

## R11 EntregaDeCódigoFuente

Entrega tanto del código VHDL como del código fuente de la aplicación SW de escritorio.

## R12 RealizarDemoDelSistema

Realizar una demostración del correcto funcionamiento del sistema completo.

## R13 ControlDeVersiones

Crear un repositorio para gestionar las versiones de los distintos módulos desarrollados, tanto para la parte hardware como software.

## R14 FechaEntrega

La fecha de entrega del proyecto será en septiembre de 2025.

### 1.4.3 Requisitos Funcionales

A continuación, se listan los requisitos funcionales del proyecto.

ID	Nombre	Prioridad	Precedencia
R3.1	UsoSensorImagen	Fundamental	-
R3.2	TecnologíaSensorDeImagen	Fundamental	R3.1
R3.3	ConexiónSensorDeImagen	Fundamental	R3.1
R3.4	ResoluciónMonocromo	Fundamental	R3.1
R3.5	ResoluciónRGB	Opcional	R3.1
R4.1	TransferencialImagenAPC	Fundamental	-
R4.2	TasaDeTransferencia	Fundamental	R4.1
R5	MostrarVelocidadEnFPGA	Fundamental	-
R6.1	ActivarDesactivarReconocimiento	Fundamental	-
R6.2	ActivarDesactivarDetección	Fundamental	-
R6.3	EncuadrarSeñalAlDetectarla	Fundamental	-
R6.4	IndicarVelocidadDetectada	Fundamental	-
R7.1	UsoRealEnVehículos	Opcional	-
R7.2	VelocidadMáximaEnVehículo	Opcional	R7.1

Tabla 1.2 Requisitos Funcionales del proyecto

#### R3.1 UsoSensorImagen

Se necesita usar un sensor de imagen para el proceso de captura de las imágenes de señales de tráfico.

#### R3.2 TecnologíaSensorDeImagen

Uso de sensor de imagen de vídeo con tecnología CMOS (*Complementary Metal-Oxide-Semiconductor*).

#### R3.3 ConexiónSensorDeImagen

El sistema debe permitir la conexión con un sensor de imagen a través de la FPGA.

#### R3.4 ResoluciónMonocromo

El sensor deberá tener resolución en escala de grises (hasta 30FPS de tasa de transferencia con este formato).

#### R3.5 ResoluciónRGB

Formato RGB (hasta 30FPS). Puede ser necesario para detectar áreas de color.

#### R4.1 TransferencialImagenAPC

Transmisión por USB de las imágenes captadas por el sensor desde la FPGA al PC.

#### R4.2 TasaDeTransferencia

Se deberá garantizar una tasa de transferencia mínima de entre 10 imágenes por segundo (FPS). Para dar sensación al usuario de vídeo en tiempo real.

## R5 MostrarVelocidadEnFPGA

Mostrar en un display conectado a la FPGA, la velocidad reconocida en todo momento.

### R6.1 ActivarDesactivarReconocimiento

Desde el interfaz de usuario en el PC, se podrá activar y desactivar el reconocimiento de la velocidad que se muestra en la señal detectada.

### R6.2 ActivarDesactivarDetección

Desde el interfaz de usuario en el PC, se podrá activar y desactivar la detección de la señal de velocidad.

### R6.3 EncuadrarSeñalAlDetectarla

Se debe dibujar un rectángulo alrededor de la señal detectada en el interfaz de usuario del PC.

### R6.4 IndicarVelocidadDetectada

Se debe indicar la velocidad reconocida en kilómetros por hora (km/h) de la señal de tráfico detectada.

## R7.1 UsoRealEnVehículos

Uso de este sistema en un vehículo real en movimiento.

### R7.2 VelocidadMáximaEnVehículo

Indicar la velocidad máxima a la que puede ir el vehículo para que el sistema sea capaz de funcionar correctamente, sin dejarse ninguna señal sin detectar y reconocer.

## 1.5 Casos de uso

Los actores externos que interactúan se muestran en la siguiente tabla:

Nombre	Descripción
Usuario	Persona que interactúa con el sistema.
Sistema	Tecnología que permite la detección y reconocimiento de señales de tráfico relacionadas con los límites de velocidad.

Tabla 1.3 Actores externos

Las acciones funcionales que el usuario inicia o espera del sistema son las que se muestran en el siguiente diagrama de casos de uso.

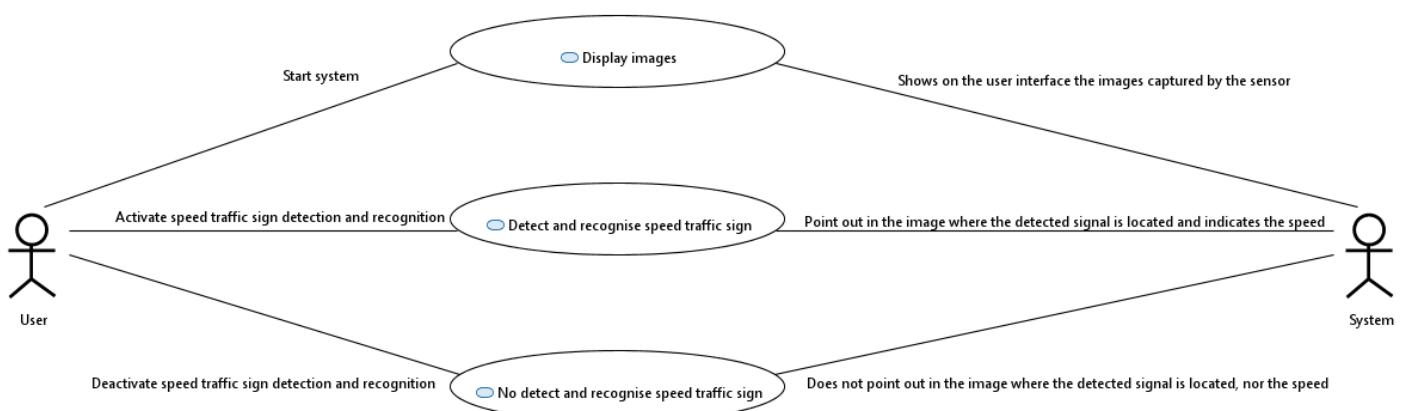


Figura 1.2 Diagrama de casos de uso

A continuación, se realiza una descripción textual de los casos de uso (Figura 1.2):

### C1. Display images

Por defecto, se visualizarán las imágenes capturadas por la cámara a una velocidad de transmisión suficiente para dar sensación al usuario de tiempo real.

- **Actor Principal:** Sistema.
- **Participantes y objetivos:** Ambos actores participan en este caso de uso. El usuario pone en marcha el sistema iniciando la aplicación y conectando correctamente el dispositivo al PC. El sistema, captura las imágenes a través del sensor, las envía al PC y las muestra en el interfaz de usuario.
- **Precondiciones:** Aplicación “SpeedTrafficSignRecognitionApp” instalada en el PC, placa de desarrollo de la FPGA alimentada, conectada al PC mediante dispositivo UM232H-B, configurada con el *bitstream* y el sensor de imagen correctamente conectado.
- **Garantías mínimas:** Las imágenes se muestran al menos a 10FPS, para dar sensación al usuario de que se trata de un vídeo en tiempo real.
- **Escenario de éxito principal:** El PC solicita imágenes a la FPGA, la cual inicia el proceso de captura y envío de datos. Conforme la aplicación de escritorio las

recibe, sólo si la opción de detección y reconocimiento está activada, se procesan.

- **Escenarios secundarios:** No llega ninguna imagen al PC o llegan corruptas y no se visualizan correctamente.

### **C2. Detect and recognise speed traffic sign**

El usuario desde la interfaz habilita la detección y reconocimiento de señales de tráfico de velocidad. Una vez hecho esto, en la aplicación se aprecia la imagen con la señal detectada encuadrada y, junto a esta, un número que indica el límite de velocidad en km/h.

- **Actor Principal:** Usuario.
- **Participantes y objetivos:** Ambos actores participan en este caso de uso. El usuario activa la detección de señales de tráfico de velocidad. El sistema, muestra las imágenes capturadas en la interfaz e indica al usuario dónde se encuentra la señal detectada dibujando un recuadro alrededor de la misma y el valor numérico de la velocidad.
- **Precondiciones:** Transmisión de imágenes al PC funciona correctamente.
- **Garantías mínimas:** Con una tasa de al menos entre 10FPS, el sistema es capaz de detectar todas las señales de tráfico relacionadas con los límites de velocidad.
- **Escenario de éxito principal:** El sistema encuadra correctamente las señales detectadas e indica correctamente el valor numérico (km/h), correspondiente al límite de velocidad que hay en la señal de tráfico.
- **Escenarios secundarios:** A pesar de que el usuario ha habilitado esta opción, el sistema no es capaz de detectar ni reconocer las señales.

### **C3. No detect and recognise speed traffic sign**

El usuario desde la interfaz deshabilita la detección y reconocimiento de señales de tráfico de velocidad. Una vez hecho esto, en la aplicación se muestra la imagen original, sin ningún recuadro o dibujo alrededor de la misma, ni ningún tipo de información acerca de la señal de tráfico detectada.

- **Actor Principal:** Usuario.
- **Participantes y objetivos:** Ambos actores participan en este caso de uso. El usuario desactiva la detección de señales de tráfico de velocidad. El sistema, muestra las imágenes capturadas en la interfaz sin indicar al usuario dónde se encuentra la señal detectada, ni tampoco indica la velocidad.
- **Precondiciones:** Transmisión de imágenes al PC y la detección y reconocimiento de señales funcionan correctamente.
- **Garantías mínimas:** El sistema responde inmediatamente a la petición del usuario y detiene la detección y reconocimiento de señales de tráfico en las imágenes capturadas.

- **Escenario de éxito principal:** El sistema muestra las imágenes originales recogidas por el sensor, sin ningún tipo de dibujo o figura que ubique las señales de tráfico detectada sobre la misma, ni ninguna información.
- **Escenarios secundarios:** A pesar de que el usuario ha deshabilitado esta opción, el sistema no es capaz de detener la detección y reconocimiento de señales.

# Capítulo 2 Descripción del Sistema

El sistema desarrollado tiene como objetivo principal la detección y reconocimiento automático de señales de tráfico relacionadas con los límites de velocidad. Para ello, se requiere de procesamiento de imágenes en tiempo real, adecuado para integrarse en entornos vehiculares o simuladores de conducción, donde se necesita monitorizar visualmente el entorno de manera continua y eficiente.

El sistema se compone de dos módulos principales:

- **Módulo hardware:** responsable de la adquisición de imágenes mediante un sensor digital y su posterior transmisión al módulo software.
- **Módulo software:** encargado de recibir las imágenes, procesarlas mediante un modelo neuronal y presentar los resultados a través de una interfaz gráfica de usuario (*Graphical User Interface, GUI*).

El usuario puede controlar el funcionamiento del sistema desde la propia aplicación: establecer la conexión con el hardware, visualizar las imágenes capturadas, y activar o desactivar la detección y reconocimiento de señales.

Gracias a su arquitectura modular y su portabilidad, este sistema es válido tanto para fines académicos o de prueba, como para una futura integración en sistemas avanzados de asistencia a la conducción ADAS.

## 2.1 Descripción General

El sistema propuesto se basa en una arquitectura modular compuesta por una parte hardware y otro software, cuya integración entre ambas se realiza a través del puerto USB (*Universal Serial Bus*) del PC, como se muestra en la Figura 2.1.

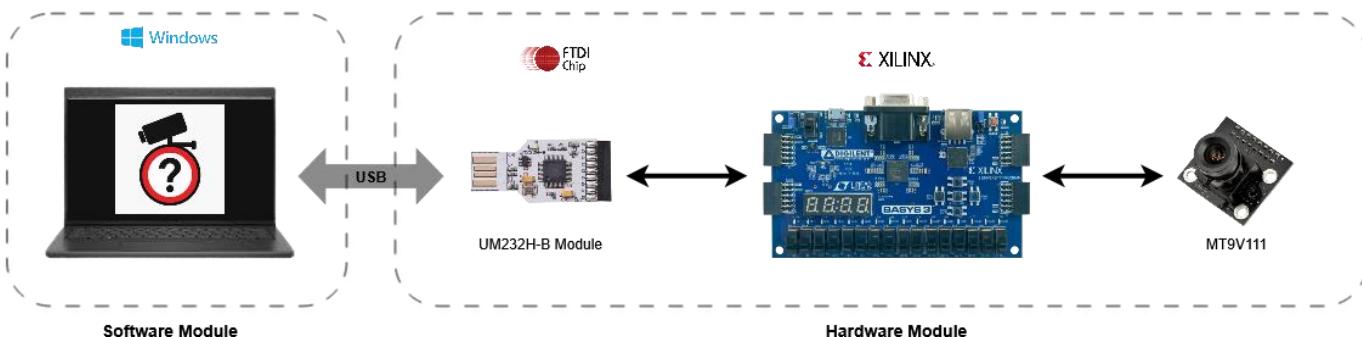


Figura 2.1 Arquitectura general del sistema

La arquitectura del sistema consta de los siguientes elementos:

- Ordenador con sistema operativo Windows.
- Aplicación de escritorio denominada “*SpeedTrafficSignRecognitionApp*”, compatible con sistema operativo Windows.
- Módulo UM232H-B del fabricante FTDI para la comunicación USB.
- Placa de desarrollo BASYS 3 con FPGA Artix-7 de Xilinx para adquisición de imágenes.
- SoC (*System on Chip*) MT9V111 con salida digital en formato YUV 4:2:2.

**Nota:** los detalles específicos de cada uno de estos componentes se describen en el Diseño e Implementación.

## 2.2 Descripción Software

El módulo software está desarrollado en **Python 3.11.11**, haciendo uso de bibliotecas como **PySide6** para la interfaz gráfica y **Ultralytics YOLO** para el reconocimiento de señales de tráfico mediante un **modelo neuronal** previamente entrenado.

La aplicación hace uso de ejecución **multithread (PySide6 QThread)** para mantener la interfaz gráfica responsiva mientras se realizan tareas de adquisición y procesamiento de imágenes.

La interfaz gráfica permite:

- Establecer conexión con la FPGA.
- Visualizar en tiempo real las imágenes capturadas.
- Activar o desactivar la detección y reconocimiento de señales.
- Mostrar los resultados del procesamiento, incluyendo la clase detectada (límite de velocidad) y el nivel de confianza asociado.

## 2.3 Descripción Hardware

El módulo hardware se implementa en la **FPGA Artix-7** de la placa BASYS 3. Su función principal es adquirir las imágenes desde el sensor MT9V111 y enviarlas hacia el PC mediante una interfaz USB controlada por el chip **FT232H**.

### 2.3.1 Módulo UM232H-B

El chip FT232H se encarga de adaptar la comunicación paralela de la FPGA al protocolo USB. Aunque es compatible con múltiples modos, se ha configurado en modo **FT245 FIFO asíncrono**, mediante la utilidad “**FTProg**”, para lograr una interfaz simple y eficiente.

Este modo emplea señales como **RD#**, **WR#**, **RXF#** y **TXE#** para sincronizar la transmisión de datos entre la FPGA y el PC. Aunque el chip admite velocidades de hasta **480 Mbps** (USB 2.0), en modo **FT245 FIFO asíncrono** la tasa de transferencia máxima es de **8 MBps**, lo cual resulta más que suficiente para las necesidades de transmisión del sistema [3].

### 2.3.2 Sensor MT9V111

El sensor de imagen **MT9V111** es un SoC CMOS que proporciona imágenes en resolución VGA ( $640 \times 480$ ) con un flujo de datos digital. Este sensor forma parte esencial del sistema y se conecta directamente a la FPGA, proporcionando señales de sincronización (**PIXCLK**, **LINE\_VALID**, **FRAME\_VALID**) y datos en formato **YUV 4:2:2**.



# Capítulo 3 Diseño e Implementación

Este capítulo, detalla tanto el proceso de diseño e implementación hardware como el software.

## 3.1 Módulo Hardware

### 3.1.1 Introducción

El diseño hardware, como se ha visto en la arquitectura del sistema, tiene como núcleo la placa de desarrollo BASYS 3, fabricada por Digilent. La cual, integra múltiples componentes, siendo la FPGA Artix 7 de Xilinx el principal, complementada con una memoria flash QSPI (*Quad Serial Peripheral Interface*) externa, interfaces estándar como USB y VGA, pulsadores mecánicos, *switches*, leds y pines de expansión PMOD Figura 3.1.

La alimentación de la BASYS 3 es suministrada mediante el conector USB J4, que, además, es el que se emplea para programar la FPGA.

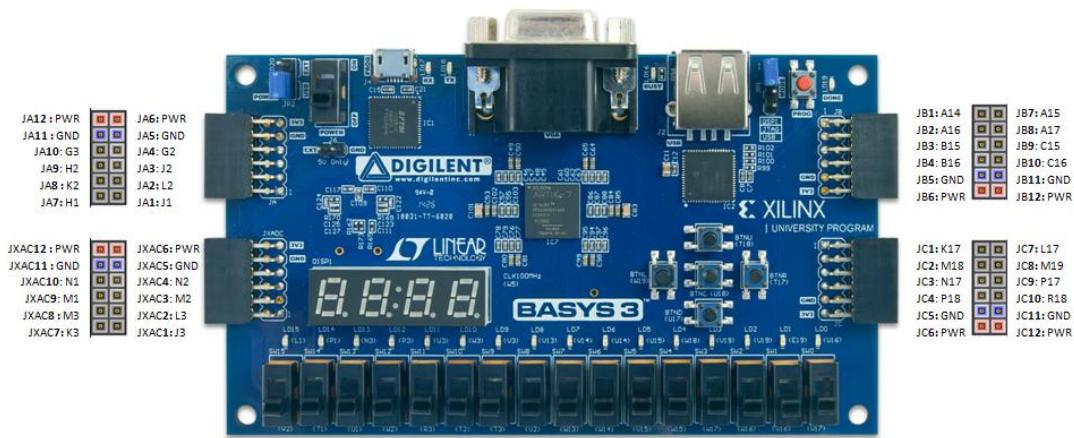


Figura 3.1 Pines PMOD placa de desarrollo BASYS 3

### 3.1.2 Placa de desarrollo BASYS 3

#### Oscilador

Todo diseño digital necesita una señal de reloj. Para este diseño, la BASYS 3 cuenta con un oscilador de **100MHz** que es la **entrada de reloj del sistema**. Se utiliza para sincronizar los distintos bloques lógicos del hardware descrito en **VHDL**.

Esta entrada, está conectada internamente al **pin W5** de la FPGA Artix-7, y así se indica en el archivo de restricciones (\*.xdc), donde se nombra para poder referenciarla en el diseño (**Máster Clock, MCLK**). A partir de esta, se pueden generar relojes derivados mediante divisores o bloques PLL si el diseño lo requiere.

## Display 7 segmentos

La placa BASYS 3 incorpora un *display* de 4 dígitos de ánodo común, donde cada dígito contiene 7 leds que pueden activarse individualmente, dando lugar a muchas combinaciones o patrones posibles (Figura 3.2). En este caso, cada 7 segmentos, se usará para representar números del 0 al 9. Además de estos, el *display* cuenta con un octavo segmento denominado **DP** que representa el punto decimal, aunque en este proyecto no tiene uso [4].

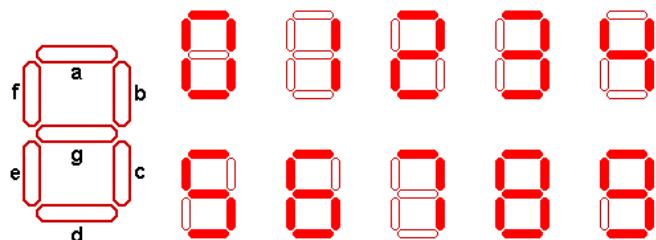


Figura 3.2 Segmentos display 7-segmentos

Los ánodos de cada dígito están controlados mediante las señales **ANx** y los cátodos comunes de los segmentos están conectados desde **CA** hasta **CG** (Figura 3.3). Esto permite una configuración multiplexada, donde los segmentos son compartidos por todos los dígitos, pero solo hay uno activo en cada instante.

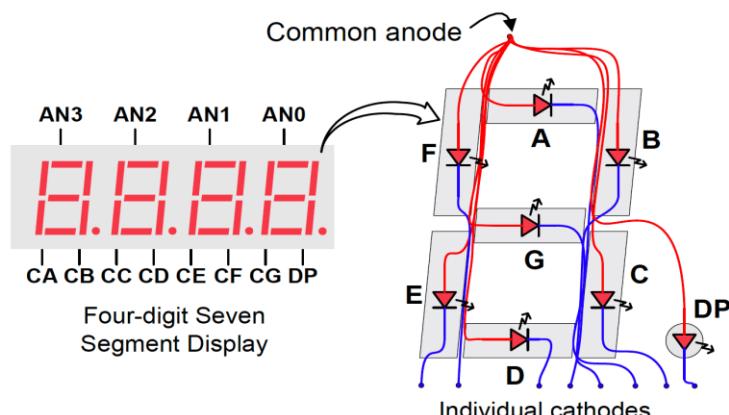


Figura 3.3 Conexiones del display

Para que todos los dígitos parezcan encendidos de forma simultánea, se recomienda activar cada uno de ellos y apagando el resto con una frecuencia de refresco alta (> 60Hz). Si la frecuencia de refresco baja de 45Hz pueden apreciarse el parpadeo de los segmentos al conmutar de dígito.

Las señales de activación están invertidas, de forma que, tanto los ánodos (**ANx**) como los cátodos (**CA** a **CG**) se activan a nivel bajo ('0').

## Conectores PMOD

Estos conectores permiten conectar periféricos como los usados en este proyecto: **UM232H-B** y **MT9V111**. Cada conector, proporciona 12 pines, de los cuales 8 son multipropósito, un par de alimentación (PWR) y el otro par de tierra (GND).

## Conexión BASYS 3

Esta placa de desarrollo cuenta con un conector tipo microUSB conectado a un controlador JTAG que va directo a la FPGA, permitiendo la programación *in-circuit*. Gracias a esta conexión, se consigue alimentar, programar y depurar el sistema al mismo tiempo. Para este proyecto que requiere que el sistema arranque sin necesidad de programación, se deberá poner el jumper JP1 en la posición QSPI, de forma que cuando la FPGA se alimente, automáticamente cargará el *bitstream* previamente almacenado en la memoria flash SPI en el proceso de producción.

### 3.1.3 Interfaz USB de conexión con PC

Como interfaz de conexión entre PC y BASYS 3 se usará el protocolo USB a través del módulo UM232H-B.



Figura 3.4 Módulo UM232H-B

Este es un módulo de desarrollo del fabricante FTDI que integra el IC (*Integrated Circuit*) FT232H, que es un interfaz de un solo canal, que puede configurarse para convertir datos entre USB y protocolos como UART, SPI, I<sup>2</sup>C, 245 FIFO, entre otros.

El sistema no impone requisitos específicos sobre la versión del protocolo USB utilizada, ya que el chip FT232H es compatible con los controladores UHCI, OHCI y EHCI, lo que garantiza su funcionamiento en puertos USB 1.1, 2.0 y 3.0 (aunque no sea capaz de aprovechar al máximo el rendimiento de este).

A diferencia de otros chips FTDI, este es capaz de operar en modos síncronos y asíncronos, lo que lo hace ideal para sistemas donde se requiere comunicación rápida y flexible [5].

Para este proyecto, se va a configurar a través de su utilidad “**FTProg**”, programando su EEPROM para trabajar en modo FT245 FIFO asíncrona (Figura 3.5).

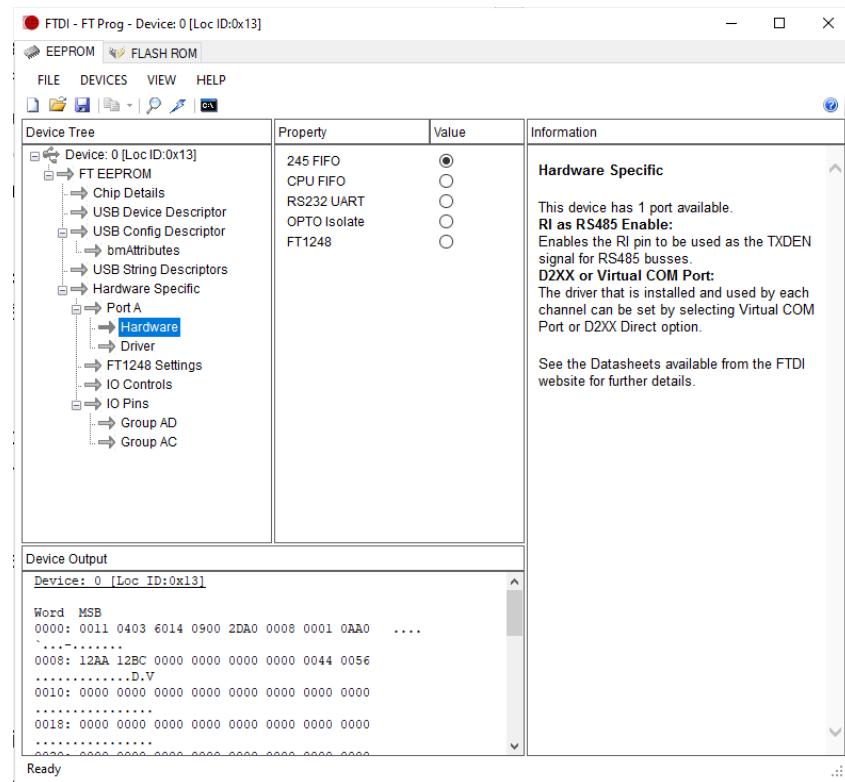


Figura 3.5 Configuración FTDI desde FTProg [17]

Este es un protocolo simple de comunicación paralela de 8 bits, en el que los datos se transmiten mediante señales de control: **RD#**, **WR#**, **RXF#** y **TXE#**.

Se ha decidido emplear este módulo por su facilidad de integración a través de librerías estándar con soporte en varios lenguajes de programación. Además, desde el punto de vista hardware, por su compatibilidad con los PMOD de la placa de desarrollo, además de su salida de alimentación a 5V, que podría servir en el futuro para alimentar la FPGA directamente sin necesidad de cable microUSB, haciendo el sistema aún más portable.

### 3.1.4 Sensor de imagen

El componente más importante y que condicionará en gran parte el rendimiento del sistema, es el sensor de imagen CMOS, integrado sobre un SOC conocido como MT9V11, cuyo fabricante es *Micron*. Entre los puntos a destacar [6]:

#### Resolución de imagen

- El sensor ofrece una resolución VGA de 640x480 píxeles.
- Emplea una matriz Active Pixel Sensor (APS), que proporciona buena sensibilidad a la luz y bajo consumo.
- Soporta también modos de ventana (*windowing*) para capturar regiones parciales de la imagen a mayor velocidad.

## Frecuencia de imagen

- El sensor es capaz de capturar imágenes a una tasa de hasta 30 *frames* por segundo a resolución completa.
- Esta tasa de fotogramas variará en función de la configuración de exposición, tamaño de ventana activo y la frecuencia de reloj de entrada (**XCLK**).

## Tamaño de píxel y formato óptico

- Cada píxel tiene un tamaño de  $6.0\mu\text{m} \times 6.0\mu\text{m}$ , lo cual proporciona buena sensibilidad para una iluminación estándar.
- El formato óptico del sensor es de aproximadamente **1/4"**, lo que corresponde a una diagonal activa de unos 4.8 mm, siendo un estándar de la industria que indica la compatibilidad con ópticas compactas.

## Interfaz de datos

- El sensor entrega la imagen digital a través de un bus de datos paralelo de 8 bits (**DATA [7:0]**).
- La sincronización de los datos transmitidos se gestiona a través de las siguientes señales:
  - **PCLK**: reloj de píxel.
  - **LVAL** (o HREF): indica líneas activas.
  - **FVAL** (o VSYNC): indica fotogramas activos.

## Configuración

- Este SoC permite configurar varios parámetros a través de un bus I<sup>2</sup>C.
- El sensor requiere de una señal de reloj externa para funcionar (**XCLK**).
- Típicamente, se emplea una frecuencia de 27MHz, que es la frecuencia máxima de entrada admitida para sacar el máximo rendimiento. En nuestro caso, por no poder llegar exactamente a esta frecuencia, con la señal de reloj de sistema de 100MHz (**MCLK**), se trabajará a 25MHz.
- Este reloj se usará internamente para generar todas las señales de sincronización mencionadas anteriormente.

### 3.1.5 Bloques Funcionales

A continuación, se muestra el diagrama de bloques detallado que representa la implementación hardware desarrollada en VHDL. En él se observan las conexiones entre los distintos bloques lógicos, así como las señales de control y datos relevantes que permiten el funcionamiento del sistema en tiempo real.

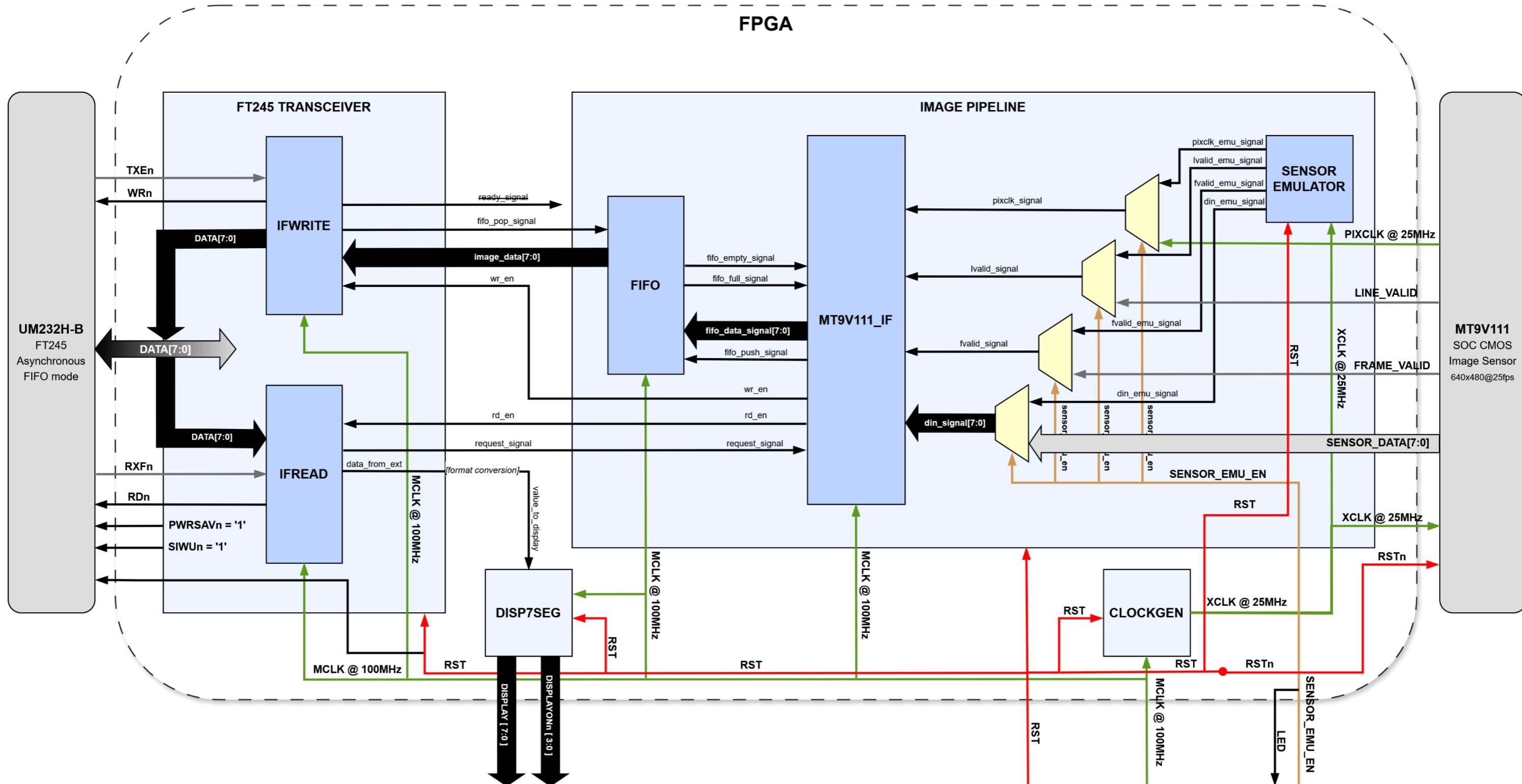


Figura 3.6 Arquitectura hardware del sistema

Todas las señales terminadas en la letra “n” son activas a nivel bajo (**TXEn**, **RXF<sub>n</sub>**, **DISPLAYON<sub>n</sub>**, etc.). El sistema presenta un solo dominio de reloj o **Máster Clock (MCLK)** de **100MHz** generado por la FPGA. Todo el sistema debe ser síncrono a esta entrada de reloj, pero al contar con señales de control asíncronas como **RXF<sub>n</sub>** y **TXEn** del módulo UM232H-B, es necesaria una proceso de sincronización de estas con el reloj del sistema. Con el objetivo de minimizar el riesgo de **metaestabilidad**, que puede producir errores de captura y comportamiento impredecible en lógica sincrónica, se implementa un **sincronizador de dos etapas** como el de la Figura 3.7, formado por un registro de desplazamiento con dos **Flip-Flops** para sincronizar cada señal asíncrona y así usar siempre sus señales síncronas y estables (**sync\_signal**).

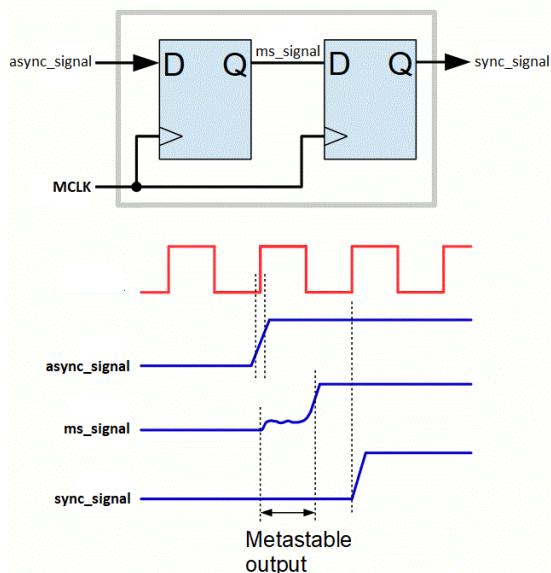


Figura 3.7 Circuito sincronizador 2-FF

El sistema cuenta con una señal de reinicio global denominada **RST**, activa a nivel alto y que se emplea para inicializar el sistema al completo a petición del usuario a través del botón **BTNC** de la BASYS 3. También, se utiliza una versión invertida de esta señal, **RST<sub>n</sub>**, activa a nivel bajo destinada principalmente al sensor de imagen MT9V111.

## FT245 TRANSCEIVER

Este bloque funcional comprende los interfaces de lectura (**IFREAD**) y escritura (**IFWRITE**) del UM232H-B.

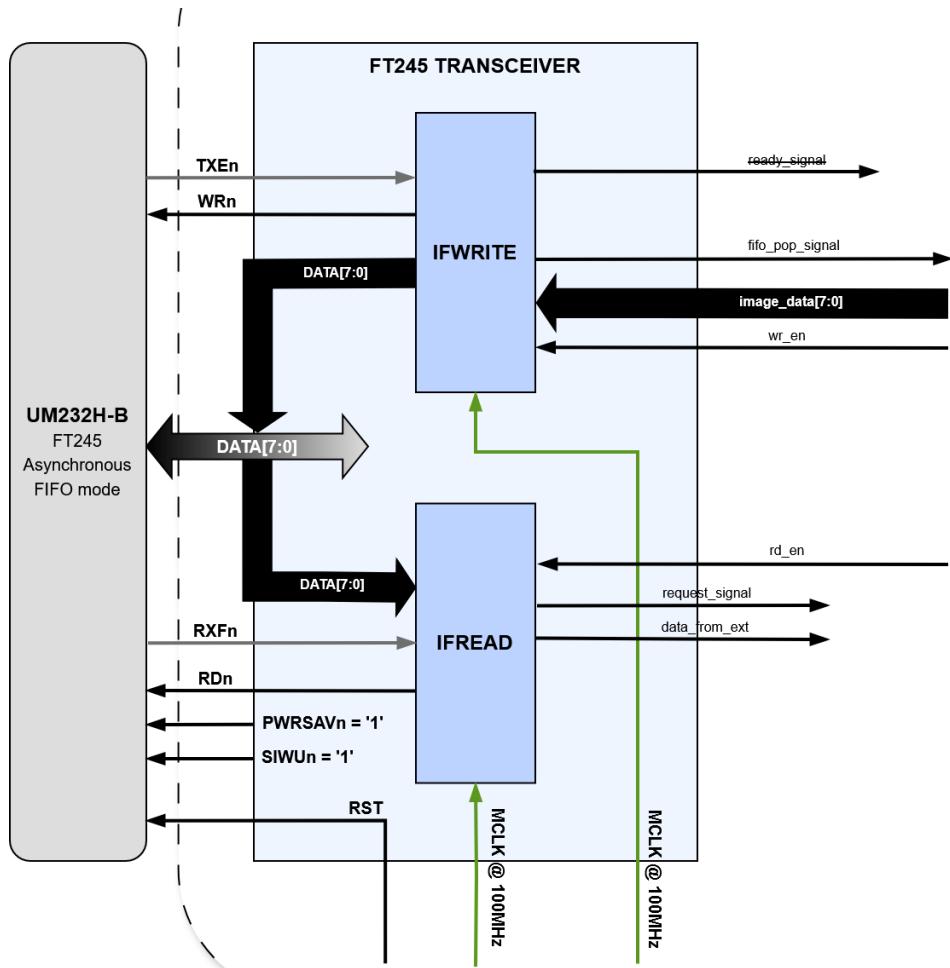


Figura 3.8 Arquitectura bloque FT245 TRANSCEIVER

**Ambos comparten un bus de datos DATA de 8 bits bidireccional o inout (de entrada/salida), por el que se intercambian datos entre PC y FPGA. El resto de las señales de control son independientes y se describen a continuación:**

- **IFREAD**

Este componente se encarga de realizar la lectura de datos procedentes del módulo UM-232H-B cuando el PC envía información hacia la FPGA.

- **Entradas:**

**RXFn:** Flag activo a nivel bajo. Indica que existen datos disponibles para la lectura desde el módulo UM232H-B (estado de su FIFO RX).

- **0:** hay datos en el buffer.

- **1:** buffer vacío.
- **Salidas:**
    - RDn:** Bit de control activo a nivel bajo. Se activa para realizar la lectura del dato disponible en la FIFO RX del USB.
      - **0:** se realiza la lectura del dato disponible.
      - **1:** no se realiza la lectura.
    - PWRSAVn:** activo a nivel bajo. Señal que indica al módulo USB que no entre en modo ahorro de energía.
      - **0:** para que no entre en modo ahorro de energía.
      - **1:** para que entre en modo ahorro de energía.
  - En este diseño, se fija esta señal a '**0**', de forma que el dispositivo nunca entra en modo ahorro.
  - SIWUn:** activo a nivel bajo. Mantiene el chip despierto, impidiendo que se suspenda la interfaz USB.
    - **0:** fuerza USB a modo '*'sleep'*'.
    - **1:** operación normal.
  - En este diseño, se fija esta señal a '**1**' para que el USB opere normalmente.
- **IFWRITE**

Este componente se encarga de gestionar el envío de datos desde la FPGA al PC, a través del módulo UM232H-B. Controla cuándo se puede escribir y genera la señal de escritura.

    - **Entradas:**
      - TXEn:** Flag activo a nivel bajo. Indica que el buffer interno del UM-232H-B tiene espacio para aceptar nuevos datos (estado de su FIFO TX).
        - **0:** se puede escribir.
        - **1:** buffer lleno o dispositivo no listo.
    - **Salidas:**
      - WRn:** Bit de control activo a nivel bajo. Activa la escritura del dato actual de la FIFO TX al bus USB.
        - **0:** escribir datos.
        - **1:** no escribir datos.

Estos dos interfaces de control de lectura y escritura descritas siguen las correspondientes temporizaciones que indica el fabricante del dispositivo UM232H-B configurado en modo FIFO asíncrona [3]:

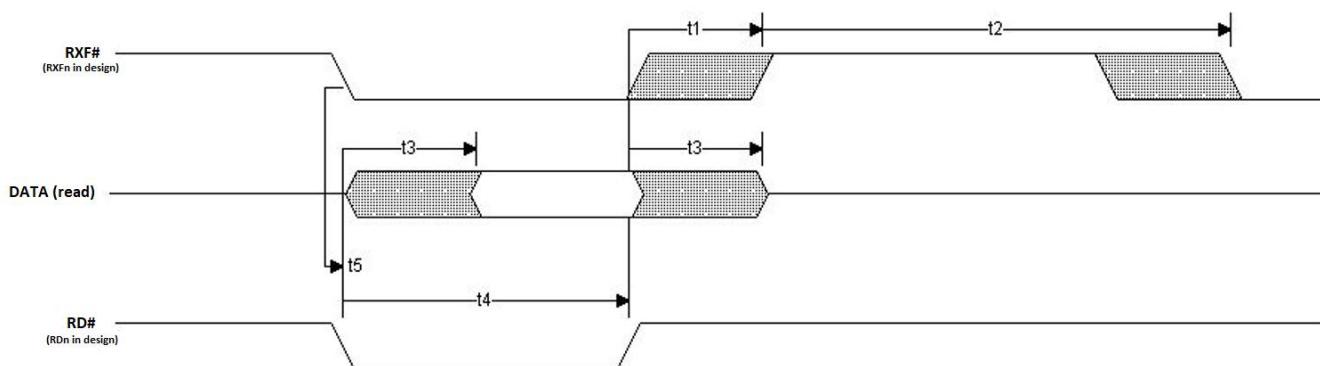


Figura 3.10 Interfaz FIFO asíncrona FT245 – señales ciclo de lectura

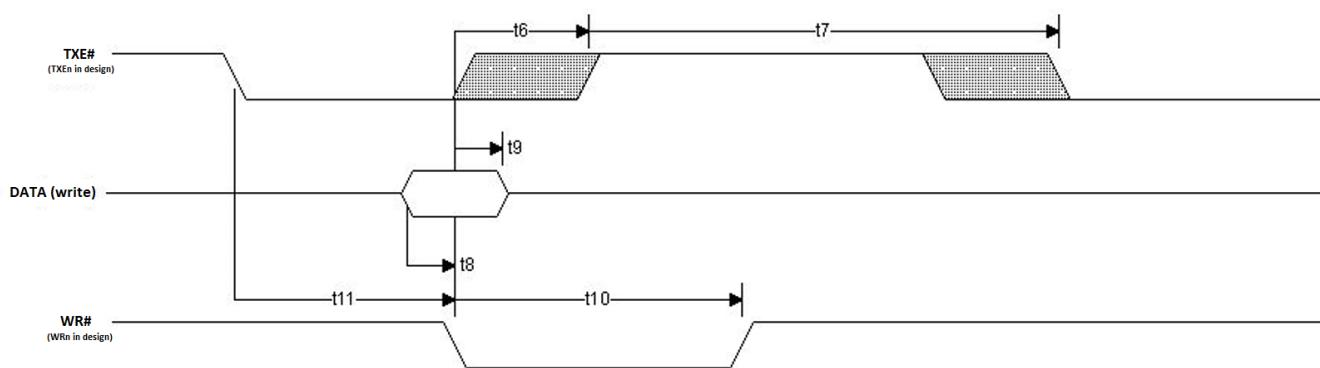


Figura 3.9 Interfaz FIFO asíncrona FT245 – señales ciclo de escritura

Time	Description	Min	Max	Units
T1	RD# inactive to RXF#	1	14	ns
T2	RXF# inactive after RD# cycle	49		ns
T3	RD# to DATA	1	14	ns
T4	RD# active pulse width	30		ns
T5	RD# active after RXF#	0		ns
T6	WR# active to TXE# inactive	1	14	ns
T7	TXE# active to TXE# after WR# cycle	49		ns
T8	DATA to WR# active setup time	5		ns
T9	DATA hold time after WR# goes active	5		ns
T10	WR# active pulse width	30		ns
T11	WR# active after TXE#	0		ns

Tabla 3.1 Requisitos de temporización FIFO asíncrona

Antes de describir la implementación de este componente, se recuerda brevemente el papel de las principales señales involucradas. **TXEn** es la señal de control generada por el FTDI para indicar a la FPGA que el buffer de escritura está disponible para recibir un nuevo byte. La señal **WRn** es controlada por la FPGA e indica al FTDI que el dato en el bus **DATA** es válido y puede ser capturado.

Siguiendo estos requisitos de temporización (Tabla 3.1), se describe a continuación el proceso de escritura de un dato en **IFWRITE** (Figura 3.9):

1. En el estado inicial (**IDLE**) del interfaz, las señales **TXEn** y **WRn** están a nivel lógico '1' y, por tanto, inactivas. En este momento, se indica al sistema que la interfaz está lista para comenzar la transmisión de un dato a través de la señal de salida **READY** a '1'. Solo cuando desde el sistema se habilita la escritura, es decir, la entrada **WR\_EN** está a '1', se comienza el ciclo de escritura.
2. Cuando esto ocurre, se pasa al estado **WAIT\_FOR\_TXE** y se indica, que el interfaz de escritura está ocupado, poniendo la señal de salida **READY** a '0'. Como el nombre de este estado indica, se espera a que la versión síncrona de la señal de entrada **TXEn** (**TXEn\_sync**) esté a '0'. Cuando lo está, se pide al sistema un nuevo dato a transmitir a través de la activación de la señal de salida **WRREQ** a '1'. Además, una vez listo el dato **DIN**, se bajará la señal **WRn** a '0' registrándose el dato en el siguiente flanco de MCLK (10ns después), para cumplir con el requisito de tener el dato preparado al menos 5ns antes del flanco de bajada de **WRn** (**T8**). Se avanza al estado **WRITE\_DATA**.
3. Una vez en este estado, se desactiva **WRREQ** a '0' y con **WRn** ya a nivel bajo se comienza la escritura del dato, que dura **NWCYCLES** de reloj. Esta constante se ha definido en 4 (esto es 40ns), para garantizar que la señal **WRn** permanece un mínimo de 30ns activa para completar el proceso de escritura (**T10**). Además, con esto también se cumple el requisito de que el dato después del comienzo de la escritura se mantenga al menos 5ns más estable en el bus (**T9**).
4. Una vez concluye el ciclo de escritura, se pone de nuevo **WRn** a '1'. La señal **TXEn** deberá permanecer inactiva durante al menos 49ns (**T7**) antes de volver a activarse para transmitir un dato. Tras este tiempo, habría dos transiciones posibles, si la escritura sigue habilitada a través de la entrada **WR\_EN = '1'**, se pasa directamente al estado **WAIT\_FOR\_TXE**, en caso contrario se volvería al estado inicial o **IDLE**.

Este proceso de escritura da lugar a la definición de la **FSM** de **IFWRITE**, que describe el proceso de comunicación desde la FPGA al PC. Teniendo en cuenta el tiempo de ciclo o período del reloj del sistema, **MCLK**, es de 10ns (100MHz), se diseña el diagrama de estados de esta FSM:

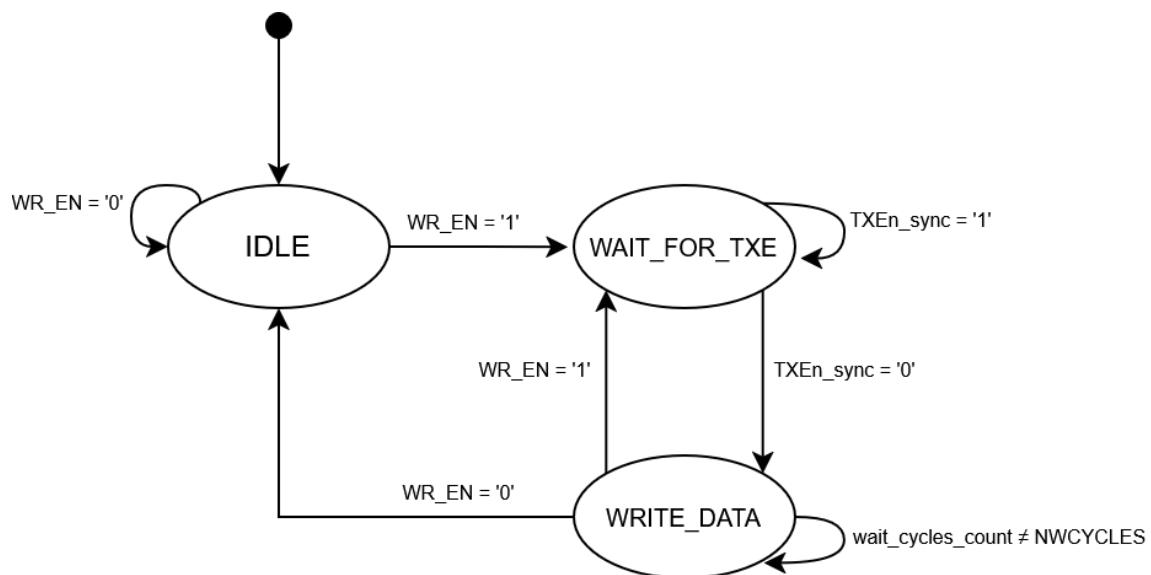


Figura 3.11 Diagrama de estados de IFWRITE

State	WRn	WRREQ	READY	DOUT	Comment
IDLE	'1'	'0'	'1'	—	Espera de orden de escritura
WAIT_FOR_TXE	'1'	'0' → '1'	'0'	—	Espera de disponibilidad del FTDI (TXEn_sync = '0')
WRITE_DATA	'0'	'0'	'0'	DIN	WRn activo durante NWCYCLES, DOUT estable
(Fin ciclo)	'1'	'0'	'1' (WR_EN='0')	—	Retorno a estado IDLE o espera próximo ciclo de escritura (WAIT_FOR_TXE)

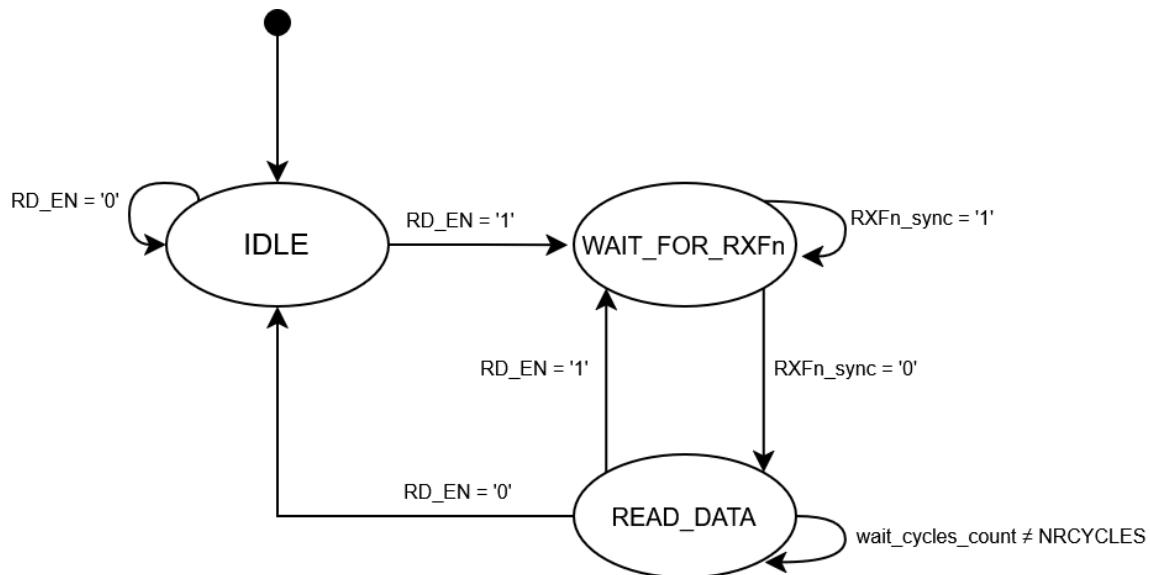
Tabla 3.2 Estados y señales de IFWRITE

Antes de describir la implementación del componente **IFREAD**, se recuerda brevemente el papel de las principales señales involucradas. **RXF<sub>n</sub>** es la señal de control generada por el FTDI para indicar a la FPGA que hay un nuevo byte disponible para su lectura. La señal **RD<sub>n</sub>** es controlada por la FPGA e indica al FTDI que el proceso de lectura está activo.

Siguiendo los requisitos de temporización para estas señales (Tabla 3.1), se describe a continuación el proceso de lectura de un dato en **IFREAD** (Figura 3.10):

1. En el estado inicial (**IDLE**) del interfaz, las señales **RXF<sub>n</sub>** y **RD<sub>n</sub>** están a nivel lógico '1' y, por tanto, inactivas. En este momento, se indica al sistema que la interfaz no ha recibido ningún dato aún, a través de la señal de salida **RDREQ** a '0'. Solo cuando desde el sistema se habilita la lectura, es decir, la entrada **RD\_EN** está a '1', se comienza el ciclo de lectura.
2. Cuando esto ocurre, se pasa al estado **WAIT\_FOR\_RXFn**. Como su nombre indica, se espera a que la versión síncrona de la señal de entrada **RXF<sub>n</sub>** (**RXF<sub>n</sub>\_sync**) esté a '0'. Cuando lo esté, se bajará la señal **RD<sub>n</sub>** a '0' en el siguiente flanco de MCLK (10ns después), aunque no hay tiempo mínimo requerido entre el flanco de bajada de **RXF<sub>n</sub>\_sync** y el de **RD<sub>n</sub>** según la Tabla 3.1 (**T5** es 0ns). Se avanza al estado **READ\_DATA**.
3. Una vez en este estado, con **RD<sub>n</sub>** ya a nivel bajo se comienza la lectura del dato, que dura **NRCYCLES** de reloj. Esta constante se ha definido en 4 (esto es 40ns), para garantizar que la señal **RD<sub>n</sub>** permanece un mínimo de 30ns activa para completar el proceso de lectura (**T4**). En el último ciclo de espera de **NRCYCLES**, se activa **RDREQ** a '1' y se captura el dato de entrada **DIN**.
4. Una vez concluye el ciclo de lectura, se pone de nuevo **RD<sub>n</sub>** a '1', lo que indica al FTDI que la lectura a finalizado. A partir de aquí, se deben cumplir dos requisitos temporales según la tabla del fabricante. El primero es **T1** (entre 1 y 14ns), que corresponde al tiempo desde que **RD<sub>n</sub>** se desactiva hasta que **RXF<sub>n</sub>** puede volver a activarse. El segundo, afecta a la señal **RXF<sub>n</sub>**, que deberá permanecer inactiva durante al menos 49ns (**T2**) antes de volver a activarse para indicar que hay un nuevo dato disponible para ser leído. Tras este tiempo, habría dos transiciones posibles, si la lectura sigue habilitada a través de la entrada **RD\_EN = '1'**, se pasa directamente al estado **WAIT\_FOR\_RXFn**, en caso contrario se volvería al estado inicial o **IDLE**.

Este proceso de lectura da lugar a la definición de la **FSM** de **IFREAD**, que describe el proceso de comunicación desde el PC a la FPGA. El diagrama de estados de esta FSM es el siguiente:



*Figura 3.12 Diagrama de estados de IFREAD*

State	RDn	RDREQ	DIN	Comment
IDLE	'1'	'0'	—	Estado inicial. RXFn aún no activa. Espera RD_EN = '1'.
WAIT_FOR_RXFn	'1'	'0'	—	Espera a que RXFn_sync = '0'. Se activa RDn en el siguiente ciclo.
READ_DATA	'0'	'0' → '1' en ciclo 1	DIN capturado en wait_cycles_count = 1	RDn se mantiene activa durante NRCYCLES, DIN se captura, se genera RDREQ.
(Fin de ciclo)	'1'	'0'	—	Se desactiva RDn. Transición a WAIT_FOR_RXFn o IDLE según RD_EN.

*Tabla 3.3 Estados y señales de IFREAD*

## CLOCK GENERATOR

Este componente se encarga de generar la salida de reloj **XCLK** que será la entrada del SoC MT9V111, como se ha mencionado anteriormente. Este generador, consiste en un divisor de frecuencia, implementado mediante un contador binario que cuenta ciclos (constante **NCYCLES + 1**) del reloj de entrada. En este caso, la señal de reloj de entrada es de 100MHz (**MCLK**) y la objetivo, por ejemplo, 25MHz. Tendremos que dividir entre 4 (**DIVISOR**) nuestra entrada, por lo que, siguiendo la fórmula:

$$2 * (\text{NCYCLES} + 1) = \text{DIVISOR}$$

Habrá que definir la constante **NCYCLES = 1**.

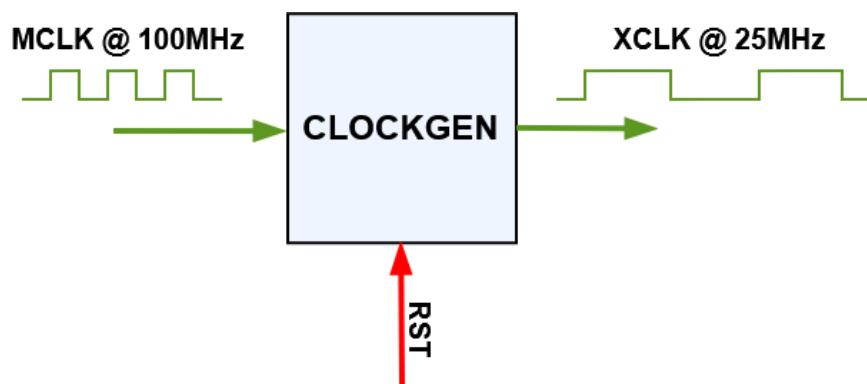


Figura 3.13 Arquitectura bloque CLOCK GENERATOR

Alternativamente, la FPGA dispone de bloques dedicados **MMCM/PLL**, que permiten generar relojes con mayor precisión en frecuencia, duty cycle y fase, además de distribuirlos por la red global de reloj. En este caso, dado que la aplicación únicamente requiere dividir por un valor entero sencillo, se optó por un divisor lógico por simplicidad de implementación, aunque el uso de un **MMCM** sería la opción más recomendable en un diseño de producción.

## DISPLAY 7 SEGMENTOS

Este bloque implementa el control del *display 7 segmentos* de 4 dígitos que incorpora la BASYS 3, descrito en el apartado 3.1.2 Placa de desarrollo BASYS 3.

Aunque estén disponibles cuatro dígitos, únicamente se representarán números naturales de 3 cifras, esto es, desde 0 a 999.

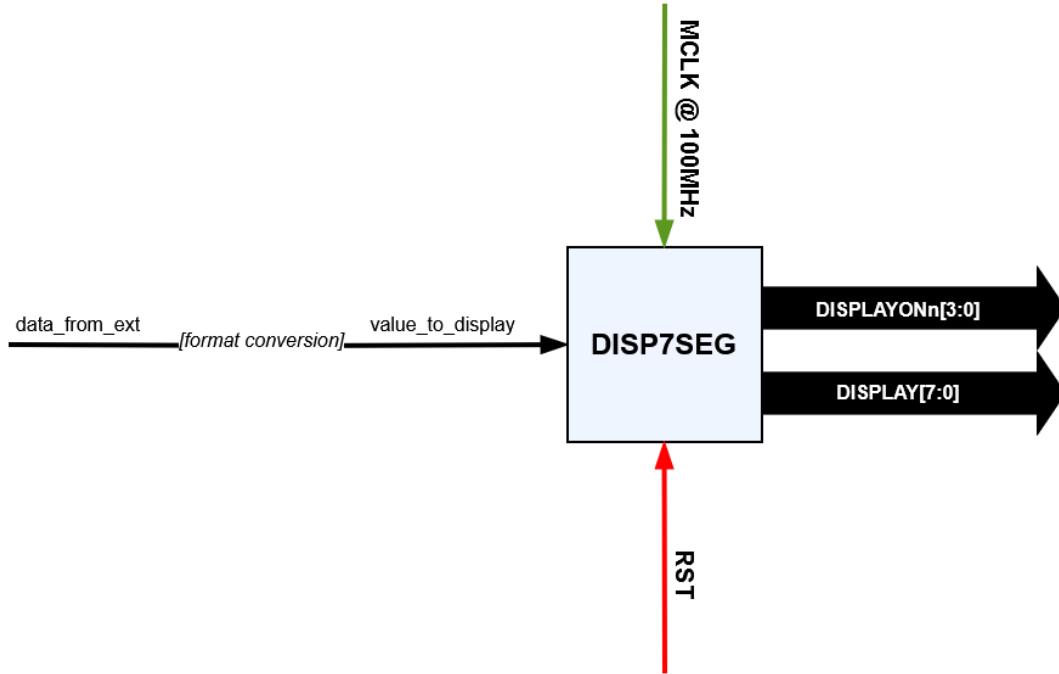


Figura 3.14 Arquitectura bloque DISPLAY 7 SEGMENTOS

Esencialmente, este componente, descompone el número **value\_to\_display** en centenas, decenas y unidades. Posteriormente, activa cíclicamente uno de los 3 dígitos, esto es, cada ánodo correspondiente al dígito (**DISPLAYONn**) a una velocidad suficientemente alta como para que el ojo humano perciba los tres encendidos a la vez. Además, genera las señales necesarias para activas los cátodos (**DISPLAY**) correspondientes a los segmentos que forman cada número. Cuando se activa la señal externa de **RST**, el *display* muestra "000".

## IMAGE PIPELINE

Este bloque funcional se encarga de la adquisición de imágenes reales capturadas por el sensor de imagen **MT9V111** o emuladas desde el componente **SENSOR EMULATOR**. Al recibirlas a través del interfaz **MT9V111\_IF**, se almacenan en forma de bytes que contienen la información de píxel antes de ser enviadas al PC a través del bloque **FT245 TRANSCEIVER** descrito anteriormente. Este almacenamiento se realiza para que no se pierda ninguna imagen o *frame*, ni se corrompan al perderse información dentro de un mismo *frame*.

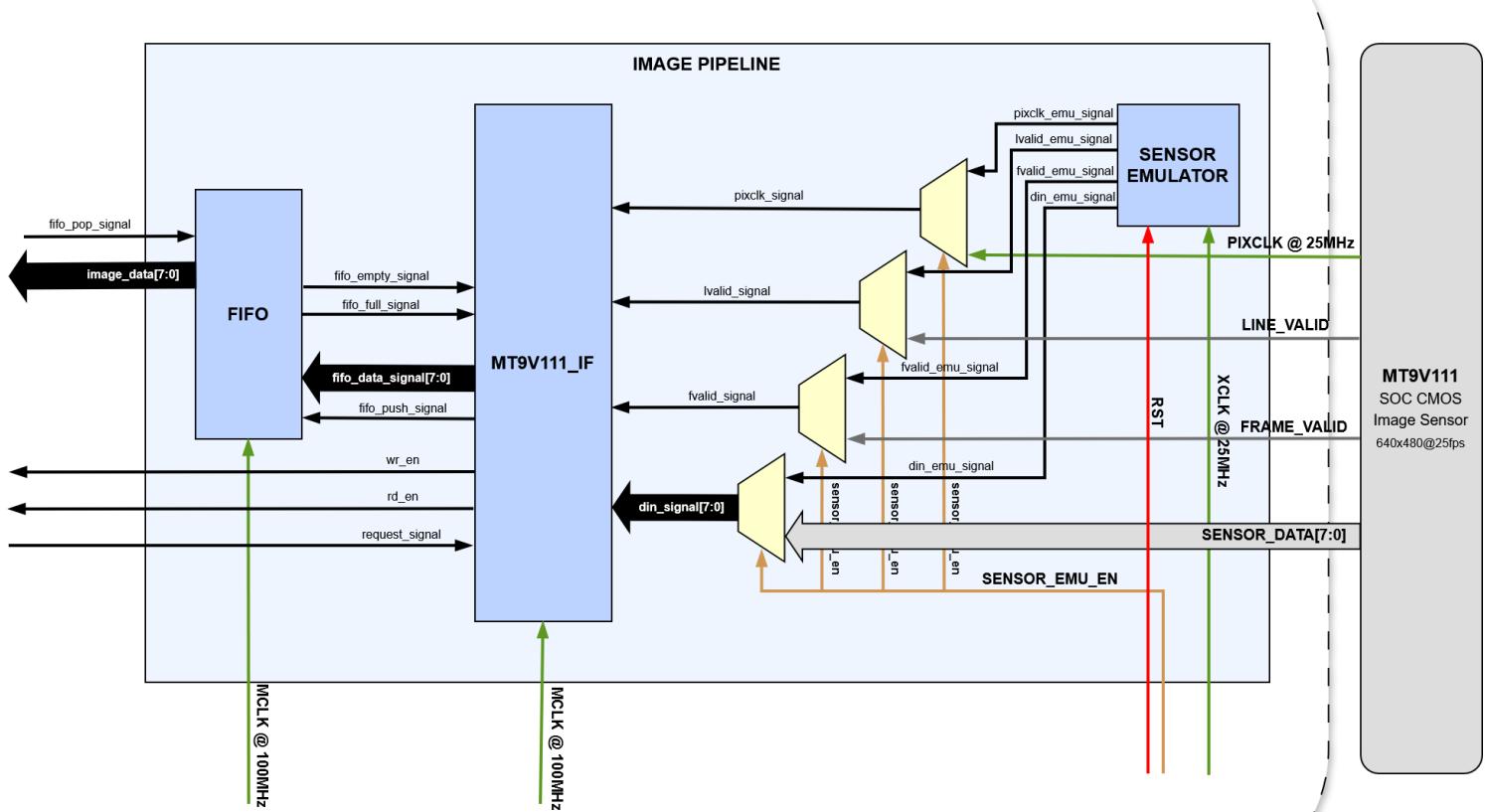


Figura 3.15 Arquitectura bloque IMAGE PIPELINE

## FIFO

La lectura de la imagen desde el sensor es más rápida que la transmisión de esta al PC, por lo que es necesario implementar en el sistema una memoria RAM de tipo FIFO, que permita almacenar los datos recibidos desde el sensor de imagen y garantizar que estos no se pierdan ni haya problemas de integridad en los datos mientras son enviados al PC.

Las FPGA modernas, como la Artix-7 (BASYS 3) disponen de bloques de memoria dedicados llamados **Block RAM (BRAM)**, diseñados precisamente para almacenar grandes cantidades de datos de forma síncrona y eficiente. De forma automática, el sintetizador, al procesar el diseño RTL de la FIFO detecta el patrón y utiliza BRAM o LUTRAM (a menos que se especifique lo contrario con atributos o pragmas, que no es el caso).

La FIFO ha sido implementada mediante memoria interna de este tipo, específicamente con **16 bloques RAMB36E1** (memorias de 36Kbits). Tal y como se indica en el reporte de utilización de la FPGA en Vivado:

Resource	Utilization	Available	Utilization %
LUT	663	20800	3.19
FF	264	41600	0.63
BRAM	16	50	32.00
IO	43	106	40.57
BUFG	2	32	6.25

Figura 3.16 Reporte utilización FPGA post-implementación en Vivado

Esto ha sido realizado automáticamente por el sintetizador de Vivado, al detectar la configuración de **FIFO** con **8 bits** de ancho de bus de datos o tamaño de palabra (**DATA\_WIDTH**) y **16 bits** de ancho de bus de direcciones (**ADDR\_WIDTH**). Esto se traduce en una capacidad de almacenamiento de:

$$2^{16} = 65536 \text{ palabras} * 8 \text{ bits/palabra} * 1 \text{ byte/8 bits} * 1 \text{kbyte/1024 bytes} = 64\text{KB}$$

La dimensión de esta FIFO se estableció en base a la simulación de un ciclo de escritura en caso peor. Ya que se debe garantizar que mientras se está escribiendo, la FIFO debe ser capaz de almacenar los datos que no se puedan enviar a través de **IFWRITE** en su caso peor, es decir, su ciclo de escritura más largo. Para ello, se recopilan los tiempos en caso peor de la tabla de requisitos de temporización del fabricante vista anteriormente, en el que el FTDI solo te permite escribir una vez cada:

$$T_{\text{total}} = T_{11} + T_6 + T_7 = 20\text{ns} + 14\text{ns} + 49\text{ns} = 83\text{ns}$$

**Nota:** Aunque  $T_{11}$  mínimo según *datasheet* sea de 0ns, se añaden 20ns debido a la latencia de dos ciclos de **MCLK** que introduce el sincronizador de dos etapas por el que pasa la señal de entrada **TXEn**.

Teniendo en cuenta que el sistema funciona con períodos de **MCLK** de **10ns**, se ajusta esta suma de tiempos:

$$T_{\text{total}} = T_{11} + T_6 + T_7 = 20\text{ns} + 20\text{ns} + 50\text{ns} = 90\text{ns}$$

Añadiendo un margen de seguridad del 10% para mayor robustez:

$$T_{\text{worst}} = T_{\text{total}} * 1.1 = 99\text{ns} \approx 100\text{ns}$$

Se realiza una simulación del sistema completo con ciclos de **TXEn** de **100ns**. Para ello se emplea el componente **SENSOR EMULATOR**, observando que la FIFO nunca llega a llenarse y que tampoco se excede demasiado cuando su capacidad se establece en **64KB**. En el capítulo de pruebas y verificaciones se comentará aún más esta simulación.

Para este bloque de memoria, de dos puertos (Dual-Port RAM), se implementa una lógica de control basada en dos señales: **POP** y **PUSH**. Estas indican cuando un dato debe ser liberado (**DOUT**) o almacenado (**DIN**) respectivamente. La lógica de estado de la FIFO depende del valor del contador de palabras almacenadas, quedando **EMPTY** activa cuando no hay datos almacenados (**WORD\_COUNTER\_REG = 0**) y **FULL** activa cuando la memoria está llena (**WORD\_COUNTER\_REG = 2<sup>16</sup>**). Su principio de funcionamiento es el de una memoria circular como la de la Figura 3.17. En la implementación de este proyecto, además de las señales de control referentes a los punteros (**RPOINTER\_REG** y **WPOINTER\_REG**), se ha usado una señal de control extra denominada **WORD\_COUNTER\_REG**, para saber exactamente el grado de ocupación de la memoria y validar su dimensión como se ha visto anteriormente.

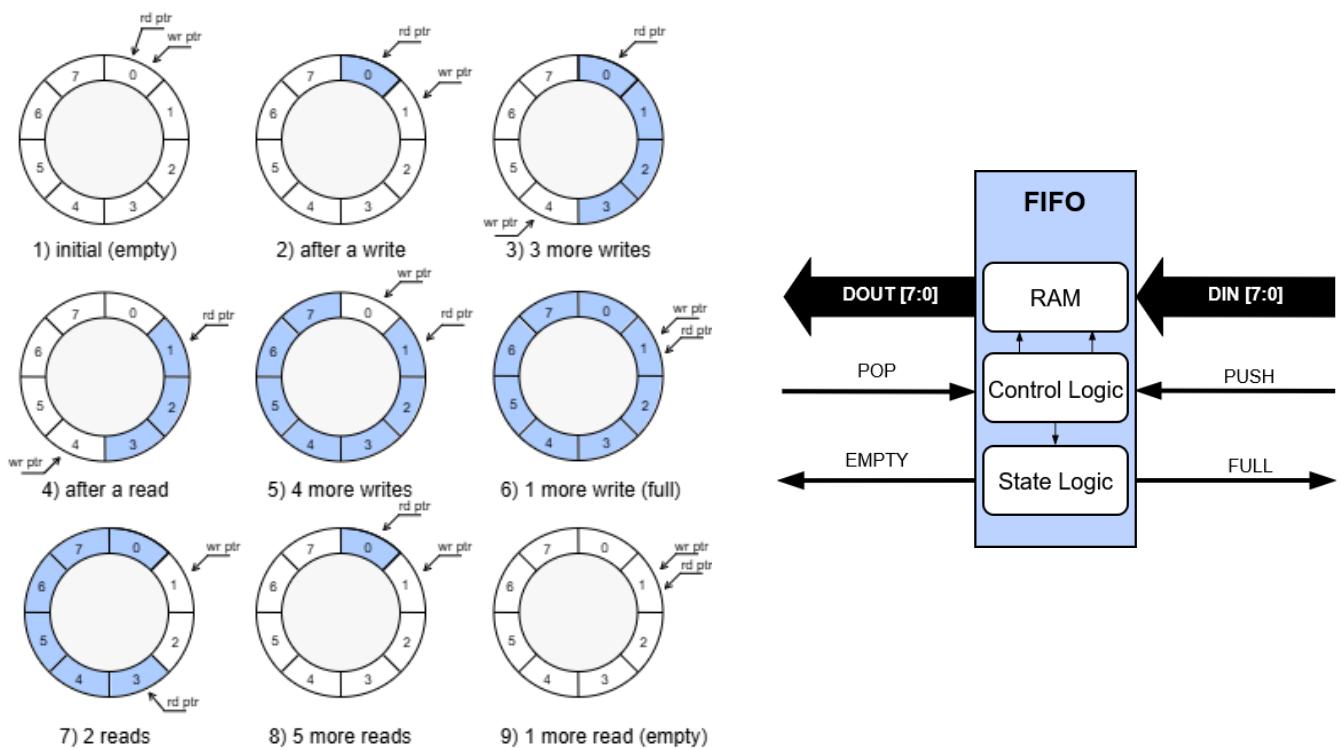


Figura 3.17 Arquitectura e implementación componente FIFO

### MT9V111\_IF

Este bloque actúa de interfaz con el sensor MT9V111 como su nombre indica, su objetivo es recibir datos de imagen procedentes del sensor y transferirlos a la FIFO del sistema, respetando las señales de control y evitando pérdidas de información. Este sensor, opera de forma síncrona con los flancos de bajada de su reloj **PIXCLK**. Este reloj no es más que una copia desfasada del reloj de entrada **CLKIN** (renombrado como **XCLK** en este diseño), generado por el componente **CLOCKGEN** desde la FPGA, con un ligero retardo (como indican los cronogramas de la Figura 3.18).

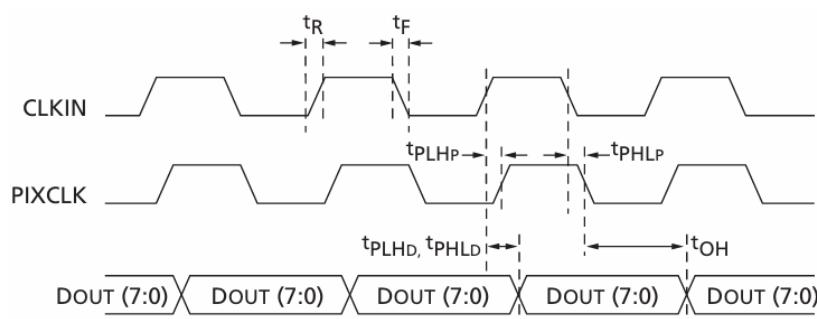


Figura 3.18 Delays de propagación de PIXCLK y DOUT

El funcionamiento de este componente se basa en una máquina de estados finita (FSM) que gestiona el flujo de datos desde el sensor real o el emulado (componente **SENSOR EMULATOR**) y controla el acceso a la FIFO de almacenamiento intermedio antes de ser enviados al PC.

Para ello, primero sincroniza la señal de reloj del sensor **PIXCLK** con el reloj del sistema **MCLK**, mediante un sincronizador de dos etapas. Los datos de imagen son enviados por el sensor en formato YUV4:2:2, aunque se trate cada byte individualmente.

Este formato es una representación de color utilizada en vídeo digital, se basa en separar la información de luminancia (Y) que representa el brillo, de la información de crominancia (U o Cb y V o Cr) que representa el color (tonos azul y rojo respectivamente). Este tipo de formato permite comprimir mejor el color sin afectar gravemente a la calidad percibida, porque el ojo humano es más sensible al brillo que al color. La notación usada 4:2:2 hace referencia al submuestreo: 4 muestras Y, 2 muestras U (Cb) y 2 muestras V (Cr). Esto significa que, en sensores como este, los datos salen como una secuencia de bytes:

**Cb0, Y0, Cr0, Y1, Cb2, Y2, Cr2, Y3...**

donde, cada grupo de 4 bytes representa 2 píxeles:

Byte	Contenido	Descripción
1	U0	Crominancia azul
2	Y0	Luminancia píxel 0
3	V0	Crominancia roja
4	Y1	Luminancia píxel 1

Tal y como muestra el diagrama de temporización de la salida de datos del *datasheet* del sensor [6]:

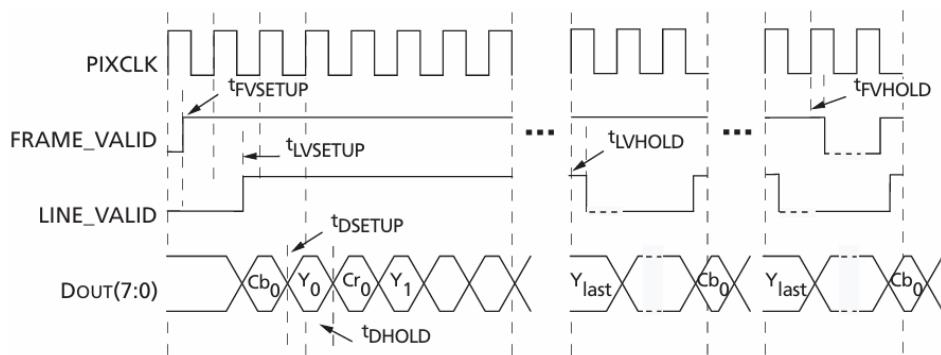


Figura 3.19 Diagrama temporización de datos de salida MT9V111

Esto significa que, si se quiere enviar la imagen en escala de grises, únicamente se deberán capturar los bytes pares correspondientes a la luminancia. Así, disminuye la información a transmitir y por tanto aumentará el número de *frames* por segundo (FPS) transmitidos al PC. En este caso, se han probado ambos, imágenes a color e imágenes en formato de escala de grises. Aunque finalmente se ha optado por enviarlas en formato YUV, pues el modelo de procesado de imágenes así lo requiere como ya se explicará posteriormente.

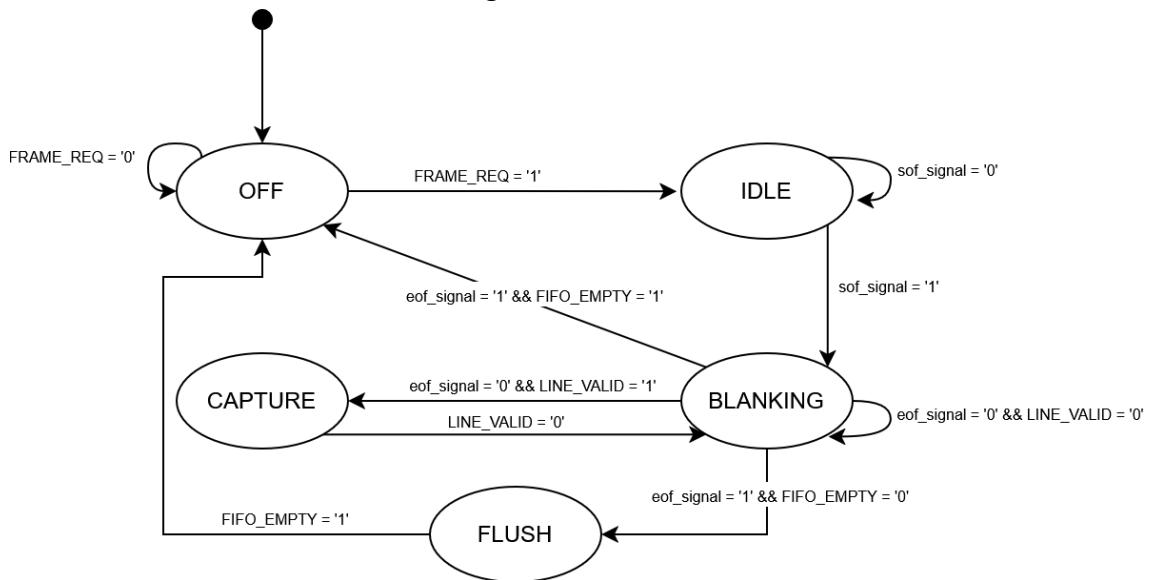
La lógica de control de esta FSM controla señales clave ya mencionadas como **WR\_EN**, **RD\_EN** y **FIFO\_PUSH**.

A continuación, se describe el proceso de captura de una imagen desde esta interfaz **MT9V111\_IF**:

1. El interfaz parte de su estado inicial u **OFF**. Este estado indica que el proceso de captura está detenido a la espera de que el sistema (en concreto, el software desde el PC) le pida una nueva captura de imagen, esto es, le active la señal de entrada **FRAME\_REQ = '1'**.

2. Una vez recibe la petición de un nuevo *frame*, la FSM pasa al estado **IDLE**, donde espera el flanco de subida de la señal de entrada **FRAME\_VALID** que viene desde el sensor real o emulado. Esta señal indica que se inicia el envío de un nuevo *frame*, internamente se traduce en la señal **sof\_signal='1'**.
3. Tras detectar el inicio de envío de una nueva imagen, se pasa al estado **BLANKING**. En este estado, se procesan datos recibidos que no traen información de la imagen, si no que se usan como períodos de sincronización en el proceso de envío por parte del sensor. En este caso, se espera a que la señal de entrada **LINE\_VALID** desde el sensor se active a '1'. Indicando que se va a transmitir la primera línea de píxeles.
4. Una vez activa esta señal, ya si se pasa al estado de captura (**CAPTURE**), pues ya si se reciben datos válidos correspondientes a la imagen transmitida. Es en este punto, cuando se activa la señal **WR\_EN** (bloque **FT245\_TRANSCEIVER**) mientras la FIFO no esté vacía (**FIFO\_EMPTY = '0'**), además de activar **FIFO\_PUSH** para almacenar el byte recibido (**DIN**) en la FIFO. Esto se hace en cada flanco de reloj **MCLK**, hasta que se detecte que **LINE\_VALID = '0'**, indicando que se ha terminado de transmitir la primera fila de la imagen.
5. Tras esto, se vuelve al estado **BLANKING**, donde de nuevo se espera a que se active **LINE\_VALID** y de nuevo pasar al estado **CAPTURE** hasta que se vuelva a desactivar.
6. Finalmente, se repite este proceso hasta que, en el estado de **BLANKING** se detecta el flanco de bajada de **FRAME\_VALID**, indicando el final del *frame*, internamente denominada **eof\_signal = '1'**. Es entonces, cuando se vuelve al estado **OFF** si la FIFO está vacía (**FIFO\_EMPTY = '1'**) o al estado **FLUSH** si la FIFO tiene aún datos. En ese caso, se activa **WR\_EN** y se envían los datos sin activar la señal **FIFO\_PUSH**, hasta que la FIFO este completamente vacía. Entonces, se vuelve al estado **OFF** y se inicia de nuevo el proceso.

A continuación, se muestra el diagrama de estados de esta FSM descrita.



*Figura 3.20 Diagrama de estados de MT9V111\_IF*

State	FIFO_PUSH	WR_EN	RD_EN	IMAGE_DATA	Comment
OFF	'0'	'0'	'1'	(others => '0')	Espera que FRAME_REQ = '1' para pasar a IDLE.
IDLE	'0'	'0'	'1'	Último valor	Espera que sof_signal = '1' para comenzar captura.
BLANKING	'0'	not FIFO_EMPTY	'0'	(others => '0')	Espera entre líneas. Si LINE_VALID = '1', pasa a CAPTURE.
CAPTURE	data_valid_signal	not FIFO_EMPTY	'0'	DIN	Captura de datos mientras LINE_VALID = '1'. En cada flanco de subida de pixclk_sync, data_valid_signal vale '1'.
FLUSH	'0'	not FIFO_EMPTY	'0'	DIN	Vacia la FIFO tras final de frame (eof_signal) si hay datos pendientes.

*Tabla 3.4 Estados y señales de MT9V111\_IF*

## SENSOR EMULATOR

Este componente se ha desarrollado para emular el funcionamiento del sensor de imagen MT9V111, permitiendo verificar el correcto funcionamiento del interfaz (**MT9V111\_IF**), así como todo el proceso de envío al PC sin necesidad de hardware real, todo a través de simulaciones. Esta emulación se basa en la generación de las señales **PIXCLK**, **FRAME\_VALID**, **LINE\_VALID** e **IMAGE\_DATA** de forma totalmente controlada. Para habilitar este componente, se debe activar o mover hacia arriba el switch **SW0** de la BASYS 3. Si no lo está, el sistema funcionará con el sensor real.

Su funcionamiento se basa en los parámetros de temporización indicados en el *datasheet* correspondientes a los períodos de imagen activa y de *blanking* (reposo). Permitiendo obtener un entorno determinista y reproducible.

El comportamiento del emulador consiste en una máquina de estados finita (FSM) con 5 estados:

- **SOFBLANKING**  
Blanking inicial de *frame*. Activa la señal **FRAME\_VALID**.
- **ACTIVE**  
Envío de datos válidos. Activa **LINE\_VALID** y transmite información de píxeles.
- **HBLANKING**  
Blanking o pausa entre líneas activas. **LINE\_VALID** = '0'.
- **EOFBLANKING**  
Blanking de final de *frame*. Últimos ciclos con **FRAME\_VALID** = '1'.
- **VBLANKING**  
Espera entre *frames*. **FRAME\_VALID** = '0'.

El cambio entre estos estados está controlado por contadores que simulan con precisión los tiempos reales del sensor [6]:

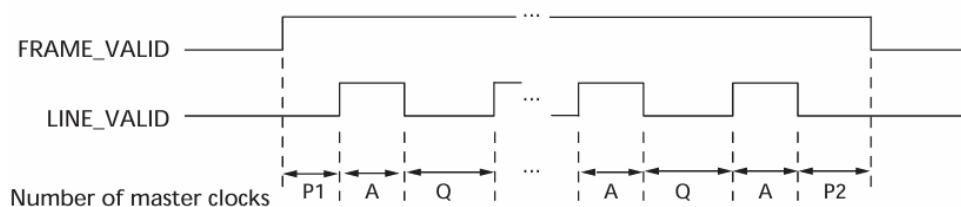


Figura 3.21 Temporización señales de control de frame

Parameter	Name	Equation (Master Clocks)	Default Timing At 12 MHz
A	Active Data Time	$(\text{Reg}0x04 - 7) \times 2$	= 1,280 pixel clocks = 1,280 master clocks = 106.7us
P1	Frame Start Blanking	$(\text{Reg}0x05 + 112) \times 2$	= 300 pixel clocks = 300 master clocks = 25.0us
P2	Frame End Blanking	14 CLKS	= 14 pixel clocks = 14 master clocks = 1.17us
Q	Horizontal Blanking	$(\text{Reg}0x05 + 121) \times 2$ (MIN Reg0x05 value = 9)	= 318 pixel clocks = 318 master clocks = 26.5us
A + Q	Row Time	$(\text{Reg}0x04 + \text{Reg}0x05 + 114) \times 2$	= 1,598 pixel clocks = 1,598 master clocks = 133.2us
V	Vertical Blanking	$(\text{Reg}0x06 + 9) \times (A + Q) + (Q - P1 - P2)$	= 20,778 pixel clocks = 20,778 master clocks = 1.73ms
Nrows x (A + Q)	Frame Valid Time	$(\text{Reg}0x03 - 7) \times (A + Q) - (Q - P1 - P2)$	= 767,036 pixel clocks = 767,036 master clocks = 63.92ms
F	Total Frame Time	$(\text{Reg}0x03 + \text{Reg}0x06 + 2) \times (A + Q)$	= 787,814 pixel clocks = 787,814 master clocks = 65.65ms

Tabla 3.5 Tiempos de frame

El sensor permite una frecuencia máxima de **PIXCLK** de **27MHz**, pero como el reloj del sistema, **MCLK**, es de **100MHz**, no se puede llegar a esa frecuencia con divisores. **La frecuencia máxima configurable por el sistema es de 25MHz.**

Este emulador permite simular distintos valores de frecuencia para **PIXCLK**, simplemente ajustando los contadores para alinearse con los tiempos que se ven en la Tabla 3.5:

```
architecture Behavioral of SENSOR_EMULATOR is

-- Pixel clocks counter signals to emulate real sensor parameters --
-- Need 2 bytes to send data of 1 pixel (NUM_PIXELS * 2) -- PIXCLK clocks | 12MHz(DEFAULT) | 25MHz |
constant ACTIVE_DATA_COUNT : NATURAL := 1280;    -- 1280 clks | 106.7us | 51.2us
constant FRAME_START_BLANKING : NATURAL := 300;    -- 300 clks | 25us | 12us
constant FRAME_END_BLANKING : NATURAL := 14;       -- 14 clks | 1.17us | 0.56us
constant HORIZONTAL_BLANKING : NATURAL := 318;     -- 318 clks | 26.5us | 12.72us
constant VERTICAL_BLANKING : NATURAL := 20778;    -- 20778 clks | 1.73ms | 0.83112ms
-- ===== Total Frame Time: 787814 clks | 65.65ms | 31.512ms |

-- Image Resolution (640x480)
constant NUM_PIXELS : NATURAL := 640;      -- Number of pixels per line
constant NUM_LINES : NATURAL := 480;        -- Number of lines per frame

-- IMAGE PATTERN TEST values
constant IMAGE_GRADIENT_PATTERN_STEP : NATURAL := 256;
constant IMAGE_PATTERN_FRAME_START_BYTE : STD_LOGIC_VECTOR(7 downto 0) := "14";
constant IMAGE_PATTERN_FRAME_END_BYTE : STD_LOGIC_VECTOR(7 downto 0) := "28";
constant IMAGE_PATTERN_LINE_START_BYTE : STD_LOGIC_VECTOR(7 downto 0) := "97";
constant IMAGE_PATTERN_LINE_END_BYTE : STD_LOGIC_VECTOR(7 downto 0) := "17";
constant IMAGE_PATTERN_HBLANKING_BYTE : STD_LOGIC_VECTOR(7 downto 0) := "A5";
constant IMAGE_PATTERN_VBLANKING_BYTE : STD_LOGIC_VECTOR(7 downto 0) := "45";
```

Figura 3.22 Definiciones en componente SENSOR EMU en VHDL

Manteniendo los números de clocks de **PIXCLK** a contar en cada caso, aumentando la frecuencia de este, se disminuye el tiempo que se tarda en

alcanzar dichos clocks. Por lo que, el tiempo de frame total se ve disminuido desde 65.65ms a 12MHz (frecuencia por defecto), hasta 31.512ms cuando la frecuencia se sube al máximo que se ha mencionado, esto es, 25MHz.

La imagen sintética a transmitir por este emulador, con resolución 640x480, se basa en un patrón creciente o gradiente con un paso de 256 píxeles (**IMAGE\_GRADIENT\_PATTERN\_STEP**). Para facilitar la identificación en simulación de los distintos estados de **BLANKING**, se han definido distintos valores constantes a transmitir en cada uno de ellos, para así diferenciarlos. Así como para el inicio y final de **FRAME** e inicio y final de **LINE**.

A continuación, se muestra el diagrama de estados de este componente:

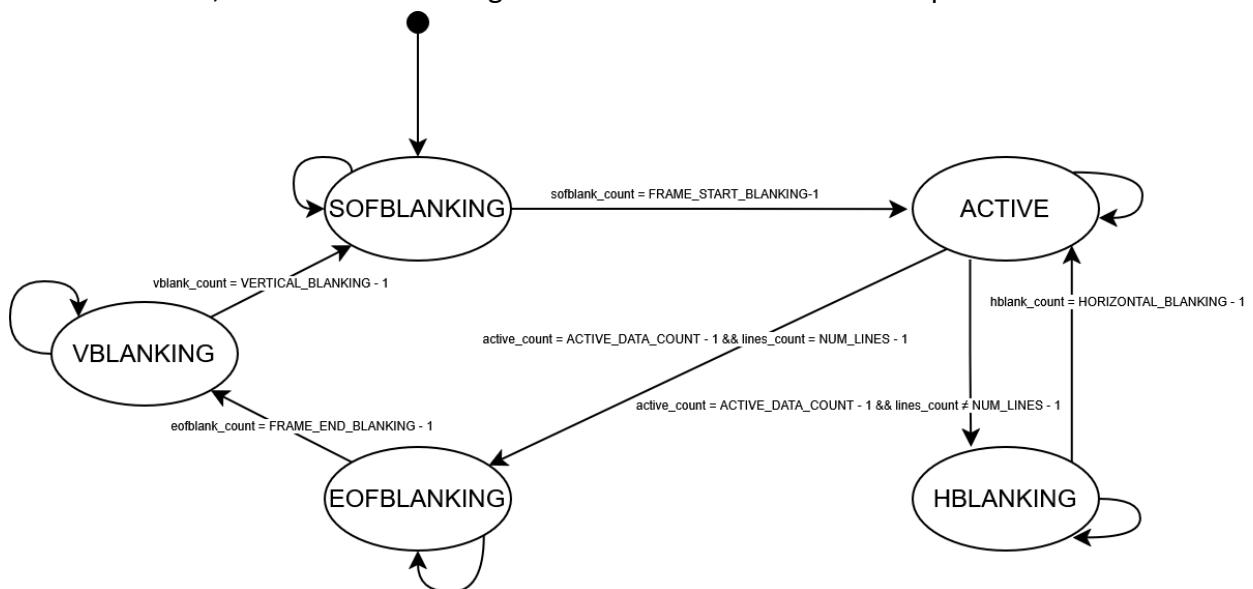


Figura 3.23 Diagrama de estados de SENSOR EMULATOR

State	FRAME_VALID	LINE_VALID	IMAGE_DATA	Comments
SOFBLANKING	'1'	'0'	IMAGE_PATTERN_FRAME_START_BYTE	Preámbulo de inicio de frame (blanking inicial).
ACTIVE	'1'	'1'	GET_HEX_PIXEL_VALUE(active_count)	Envío de datos de imagen sintética.
HBLANKING	'1'	'0'	IMAGE_PATTERN_HBLANKING_BYTE	Tiempo entre líneas (horizontal blanking).
EOFBLANKING	'1'	'0'	IMAGE_PATTERN_FRAME_END_BYTE	Indica el final de transmisión del frame.
VBLANKING	'0'	'0'	IMAGE_PATTERN_VBLANKING_BYTE	Tiempo entre frames consecutivos.

Tabla 3.6 Estados y señales de SENSOR EMULATOR

### 3.1.6 Asignación de pines de E/S

Una parte importante del proceso de diseño e implementación del módulo hardware es la definición de las conexiones de entrada y salida de la FPGA. Para ello, Xilinx proporciona un archivo **.xdc** (*Xilinx Design Constraints*), que es un archivo de restricciones de diseño que define cómo se conectan las señales lógicas internas del proyecto VHDL, en concreto del **TOP**, con los pines físicos de la FPGA.

Se trata de un archivo de texto con sintaxis basada en TCL, donde se indican:

- Las asignaciones de pines (*PACKAGE\_PIN*).
- La dirección de la señal (entrada o salida).
- La norma eléctrica del pin (*IOSTANDARD*).
- En algunos casos, restricciones temporales (*create\_clock*, *set\_input\_delay*, etc.).

En concreto, estas señales de entrada salida están disponibles en la placa de desarrollo BASYS 3, accesibles mediante los conectores **PMOD** descritos anteriormente. Aquí, se definirán, por tanto, las señales lógicas con los pines donde se conecta lo siguiente: sensor de imagen real (**MT9V111**), módulo FTDI (**UM232H-B**), el **display 7 segmentos**, el **botón de reset** y un **switch** para habilitar/deshabilitar el emulador de sensor.

Cada pin PMOD de cada puerto está asociado internamente a un pin de la FPGA y a su vez, a cada uno de ellos, se le ha asignado una señal lógica interna del sistema:

PMOD JA	FPGA Pin	FTDI Signals	PMOD XDAC	FPGA Pin	FTDI Signals	PMOD JB	FPGA Pin	MT9V111 Signals	PMOD JC	FPGA Pin	MT9V111 Signals
JA1	J1	RXFn	JXADC1	J3	DATA[0]	JB1	A14	RSTn	JC1	K17	SENSOR_DATA[6]
JA2	L2	RDn	JXADC2	L3	DATA[1]	JB2	A16	SENSOR_DATA[0]	JC2	M18	XCLK
JA3	J2	SIWUn	JXADC3	M2	DATA[2]	JB3	B15	SENSOR_DATA[2]	JC3	N17	LINE_VALID
JA4	G2	n/a	JXADC4	N2	DATA[3]	JB4	B16	SENSOR_DATA[4]	JC4	P18	SDA
JA7	H1	TXEn	JXADC7	K3	DATA[4]	JB7	A15	n/a	JC7	L17	PIXCLK
JA8	K2	WRn	JXADC8	M3	DATA[5]	JB8	A17	SENSOR_DATA[1]	JC8	M19	SENSOR_DATA[7]
JA9	H2	PWRSAVn	JXADC9	M1	DATA[6]	JB9	C15	SENSOR_DATA[3]	JC9	P17	FRAME_VALID
JA10	G3	n/a	JXADC10	N1	DATA[7]	JB10	C16	SENSOR_DATA[5]	JC10	R18	SCL

Tabla 3.7 Asignación de pines PMOD y señales internas del diseño

En este archivo, también se crea la señal de reloj en el pin correspondiente, con las siguientes líneas:

```
## Clock signal
set_property PACKAGE_PIN W5 [get_ports MCLK]
set_property IOSTANDARD LVCMOS33 [get_ports MCLK]
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {MCLK}]
```

Además, de conectar los distintos componentes integrados en la propia BASYS 3 con los pines de la FPGA:

```
## Switches (most of them without use on this application)

set_property PACKAGE_PIN V17 [get_ports SENSOR_EMU_EN]
set_property IOSTANDARD LVCMOS33 [get_ports SENSOR_EMU_EN]

## LEDs

set_property PACKAGE_PIN U16 [get_ports LED]
set_property IOSTANDARD LVCMOS33 [get_ports LED]

## Buttons (most of them without use on this application)

set_property PACKAGE_PIN U18 [get_ports RST]
set_property IOSTANDARD LVCMOS33 [get_ports RST]

## 7 segment display

# Cathodes
set_property PACKAGE_PIN W7 [get_ports {DISPLAY[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[0]}]
set_property PACKAGE_PIN W6 [get_ports {DISPLAY[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[1]}]
set_property PACKAGE_PIN U8 [get_ports {DISPLAY[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[2]}]
set_property PACKAGE_PIN V8 [get_ports {DISPLAY[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[3]}]
set_property PACKAGE_PIN U5 [get_ports {DISPLAY[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[4]}]
set_property PACKAGE_PIN V5 [get_ports {DISPLAY[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[5]}]
set_property PACKAGE_PIN U7 [get_ports {DISPLAY[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[6]}]
set_property PACKAGE_PIN V7 [get_ports {DISPLAY[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[7]}]

# Anodes
set_property PACKAGE_PIN U2 [get_ports {DISPLAYONn[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAYONn[0]}]
set_property PACKAGE_PIN U4 [get_ports {DISPLAYONn[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAYONn[1]}]
set_property PACKAGE_PIN V4 [get_ports {DISPLAYONn[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAYONn[2]}]
set_property PACKAGE_PIN W4 [get_ports {DISPLAYONn[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAYONn[3]}]
```

El LED se emplea para indicar si está habilitado o no el emulador del sensor.

Tras completar el diseño y este archivo de restricciones, se deben completar los siguientes pasos para conseguir un archivo binario para cargarlo en la FPGA desde la descripción hardware hecha en VHDL. Este archivo se denomina **bitstream** y estos son los pasos que seguir:

- **Síntesis**

Este proceso compila nuestro diseño RTL descrito en VHDL, generando una red de puertas lógicas, *FlipFlops*, otras primitivas como BRAMs e interconexiones abstractas (*netlist*). Comprueba que la descripción del hardware es sintáctica y semánticamente correcta. Optimiza el diseño según la lógica requerida y las opciones seleccionadas.

- **Implementación**

Este proceso es el que sigue a la síntesis y asigna o mapea la *netlist* generada a los recursos físicos de la FPGA: **LUTs** (Look-Up Tables), **Flip-Flops** (FFs), **BRAMs** y **IOBs** (bloques de E/S). Además, también ubica los componentes en la matriz física del chip (*placement*), conecta estos bloques o componentes físicamente (*routing*) y comprueba que el diseño cumple con las restricciones temporales (*timing analysis*). Este es el reporte que se obtiene tras esta etapa, de la ocupación de recursos de la FPGA:

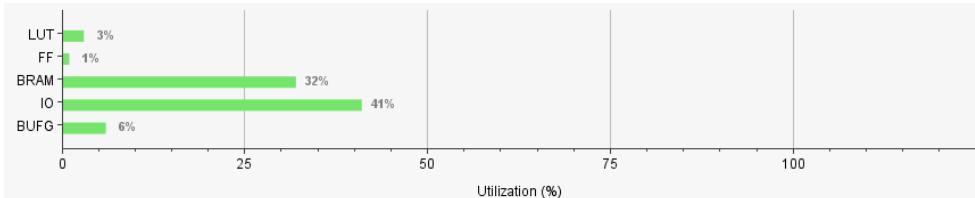


Figura 3.24 Reporte utilización FPGA post-implementación

- **Generación del bitstream**

A partir del diseño ya implementado, se genera el archivo **.bit**. Este archivo contiene toda la información binaria que configura la FPGA internamente. Este **bitstream** se almacena en la memoria no volátil externa, que en el caso de la BASYS 3 es una memoria **QSPI Flash**. Esto permite que la FPGA Artix-7 se comunique con esta memoria durante el arranque, cargando el diseño automáticamente, sin necesidad de reprogramar manualmente desde Vivado cada vez [7].

## 3.2 Módulo Software

### 3.2.1 Introducción

El diseño e implementación del software desarrollado en este proyecto permite interactuar de forma sencilla con el módulo hardware. Para ello, se ha creado una aplicación íntegramente en **Python**, utilizando **PySide6** —la vinculación oficial de Qt6 para Python— para la implementación de la **interfaz gráfica de usuario (GUI)**. Esta aplicación se ha desarrollado en **Visual Studio Code** (VS Code), debido a su flexibilidad como IDE y su integración con Python y Conda. Esta, hace uso tanto del API de Qt a través de la biblioteca PySide6 como de un **designer**, que es una herramienta gráfica que permite diseñar interfaces visualmente, facilitando la creación y organización de elementos como botones, menús y ventanas.

Como se ha comentado anteriormente, se hace uso de ejecución **multithread (PySide6 QThread)** para mantener la interfaz gráfica responsive mientras se realizan tareas de adquisición y procesamiento de imágenes.

La arquitectura del software se basa en varias clases principales, entre las que se encuentran el **Controlador**, la **Vista (GUI)**, el **Procesador de imágenes** y un **Mediador** que gestiona la comunicación entre ellas. Además de código propio, se ha hecho uso de librerías de terceros como **ftd2xx** para el control del dispositivo **UM232H-B**, y herramientas de procesamiento de imágenes para ejecutar el modelo neuronal entrenado integrado en el sistema.

El desarrollo se ha llevado a cabo en el entorno **Visual Studio Code**, utilizando un entorno virtual gestionado con **Conda** para facilitar la instalación y aislamiento de las dependencias del proyecto (PySide6, ftd2xx, etc.). Asegurando así la portabilidad y control sobre sus versiones [8].

Para más detalles sobre como crear este entorno o todas las dependencias necesarias, véase el [README](#) del módulo software del proyecto en GitHub.

La guía de instalación de la aplicación **SpeedTrafficSignRecognitionApp** se encuentra en el **Anexo B. Instalación del software** al final de este documento.

### 3.2.2 Diagrama General

La siguiente figura muestra el diagrama del flujo de ejecución de la aplicación desarrollada.

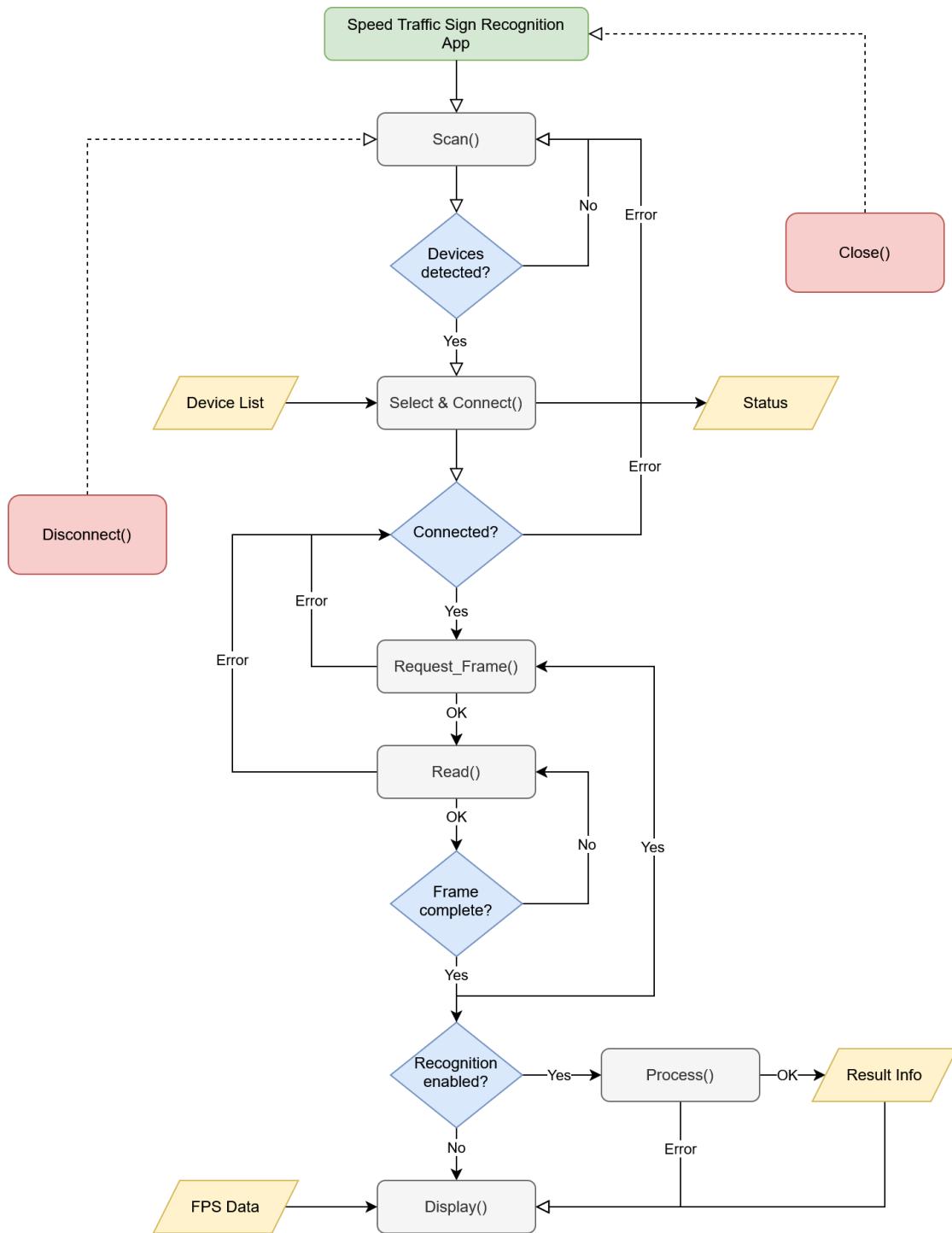


Figura 3.25 Diagrama de flujo completo de la aplicación

El flujo comienza con la inicialización de la app y el escaneo o detección de dispositivos conectados al PC. Si se detecta correctamente el módulo hardware, vía el chip FTDI (UM232H-B), aparece en la lista de dispositivos, permitiendo así seleccionarlo y establecer conexión con él.

Una vez establecida la conexión, el sistema desde uno de sus hilos de ejecución realiza la lectura de bytes recibidos desde la FPGA. Cuando se completa un *frame*, se comprueba si el usuario ha habilitado el reconocimiento de señales de tráfico relacionadas con los límites de velocidad. Si esto es así, se procede a procesarla desde otro hilo de ejecución (*multithread*), pasando la imagen por el modelo neuronal entrenado. Si esto sale mal porque no reconoce ninguna señal, realmente no la hay o no se ha habilitado el reconocimiento, se muestra la imagen original recibida y no se muestra ningún resultado del procesado. Si en cambio, si que se detecta, se muestra la información correspondiente a la señal reconocida junto con la imagen editada ubicando la señal detectada. En cualquier caso, siempre que el proceso de lectura de *frames* vaya bien, se mostrará un indicador de la tasa de *frames* por segundo recibidos.

En cualquier momento del flujo de ejecución, el usuario puede pulsar el botón de desconectar (**disconnect**), lo que fuerza volver al punto de escaneo de dispositivos conectados al PC por los puertos serie. Al igual que podría directamente cerrar la aplicación (**close**).

El diagrama también muestra puntos de salida alternativos para errores de conexión, lectura o procesamiento, lo que permite mantener una ejecución robusta ante fallos.

### 3.2.3 Arquitectura basada en clases

Cada una de las acciones descritas, son llevadas a cabo por una clase o módulo distinto: **Mediador**, **Controlador**, **Procesador de Imágenes** y **Vista (GUI)**.

#### Mediador

Esta clase coordina actúa como coordinador central de la aplicación. Su propósito es gestionar las peticiones de la interfaz gráfica (**View**), el controlador (**Controller**) y el procesador de imágenes (**Image Processor**). Gracias a este enfoque, cada componente puede operar de forma independiente, facilitando el mantenimiento y la escalabilidad del software.

Al instanciar la clase Mediador (**Mediator**), se le pasa los objetos correspondientes al controlador, procesador y vista. Esta clase mediator, recibe los eventos de conexión, desconexión, escritura y lectura del controlador. Notificando de ellos tanto a la vista, para actualizar el interfaz en cada momento, como al procesador para que procese los datos leídos. Por parte de la Vista recibe los eventos de usuario, y actúa con el resto de los módulos en base a ellos. Sólo coordina llamadas entre las distintas clases propias.

El uso del patrón de diseño **Modelo-Vista-Controlador (MVC)** combinado con el patrón **Mediator (mediador)** permite centralizar la lógica de comunicación entre componentes y reducir el acoplamiento, mejorando la modularidad del software. Este enfoque favorece futuras ampliaciones, por ejemplo, sustituir el controlador por otra fuente de datos o incluso cambiando la vista o GUI fácilmente.

## Vista

La clase **View** es responsable de la interfaz gráfica de usuario (GUI) de la aplicación. Se encarga de mostrar la información visual, recibir eventos del usuario y actualizar los elementos gráficos o *widgets* (etiquetas, botones, imágenes, etc.). Hereda de la *QMainWindow* y del formulario generado por Qt Designer (*Ui\_MainWindow*) y utiliza PySide6 como *framework* gráfico. Se comunica con el Mediador únicamente, a través de *callbacks* que le son registradas (**setConnectionCallback**, **setCloseCallback**, etc.). Utiliza señales y slots para actualizar la GUI desde otros componentes [9]. A continuación, se muestra su diagrama de flujo:

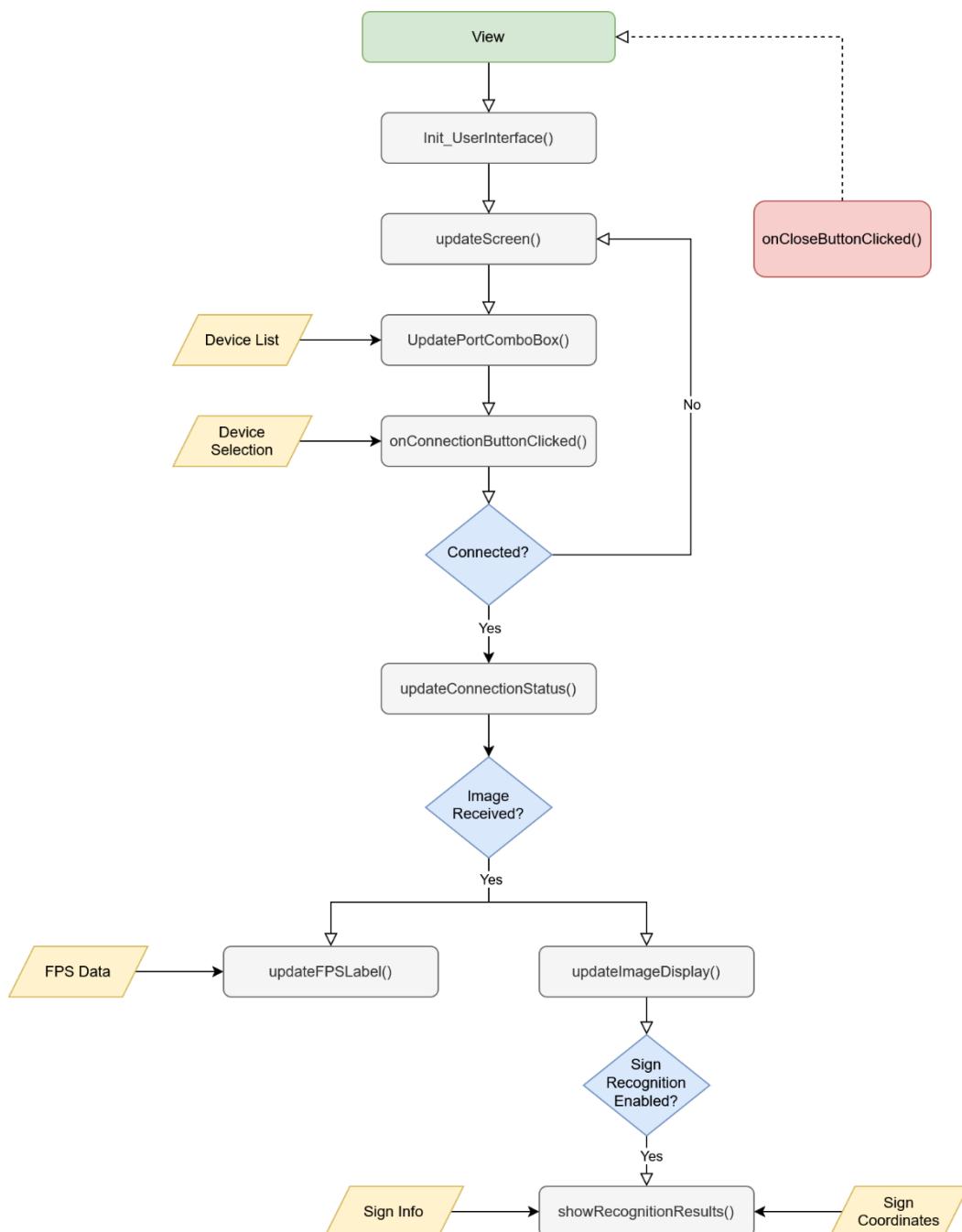


Figura 3.26 Diagrama de flujo de la clase **View**

En primer lugar, en el constructor de la clase, se inicializa la GUI (**Init\_UserInterface**). Aquí, se carga la interfaz, esto es el archivo **.ui** generado a partir del siguiente comando en el terminal:

```
pyside6-uic GUI/SpeedTrafficSignRecognitionApp_release.ui -o GUI/userInterface_release.py
```

Siendo **userInterface\_release.py** el nombre del fichero generado a través del *designer* de Qt.

Esto se hace a través de la llamada a **setupUi**, que es un método generado automáticamente por **Qt Designer** cuando exportas la interfaz a un archivo **.py**. Básicamente, esta función crea y coloca todos los widgets definidos en el archivo **.ui** dentro del objeto actual, que en este caso es una subclase de **QMainWindow**.

Para la vista de usuario se usará el interfaz **"\_release.py"**, sin embargo, se ha diseñado otro interfaz **"\_debug.py"**, que es como una vista de desarrollador, para poder hacer diversas pruebas sobre el sistema. Para poder cargar una interfaz u otra sólo hace falta actuar sobre la etiqueta **\_DEBUG\_COMPILATION**:

```
if _DEBUG_COMPILATION:  
    from GUI.userInterface_debug import Ui_MainWindow  
else:  
    from GUI.userInterface_release import Ui_MainWindow
```

Esto demuestra la potencia de modular el software, permitiendo de forma fácil cargar cualquier GUI.

Posteriormente, se aplica el estilo de la interfaz, esto es, por ejemplo, la fuente de los textos, la forma del cuadro donde irá la imagen, se centra la ventana de la app, entre otras cosas. También se registran y conectan los botones y widgets a funciones internas o *callbacks* externas.

Una vez se ha inicializado la interfaz, se actualiza el widget **comboBox** con la lista de dispositivos detectados que indica el Mediator, que a su vez le ha indicado el controlador. Si el usuario pulsa el botón **connect**, activando el evento **onConnectionButtonClicked**, se establece la conexión a través de una llamada al Mediator desde una *callback*. Si se ha conectado correctamente se actualiza la interfaz, mostrando el estado de conexión a través de **updateConnectionStatus**. Aquí, el interfaz se queda en espera y una vez recibida una imagen por parte del Mediator, este se la pasará a esta clase a través de la función **updateImageDisplay** junto con el valor de FPS para que muestre la información en un widget tipo *label*. Si el usuario ha habilitado el **checkBox** correspondiente al reconocimiento de señales, además, sobre la imagen se dibujará un rectángulo en las coordenadas donde se haya detectado la señal y si se ha detectado (*Sign Coordinates*), junto con información sobre esta justo a la derecha. Indicando en una etiqueta la velocidad en km/h, la confianza del



reconocimiento en forma de porcentaje y en otra si se ha reconocido o no alguna señal (*Sign Info*). Pues por defecto si está habilitada siempre se mostrará la información de la última señal reconocida.

## Controlador

La clase “**FTDIController**” es el componente encargado de gestionar toda la comunicación entre la aplicación y el chip FTDI (UM232H-B). Su función es abstraer la lógica de bajo nivel necesaria para escanear, conectar, leer y escribir datos a través del puerto USB del PC, proporcionando una interfaz limpia y transparente para el mediador.

Esta clase depende de la biblioteca **ftd2xx**, que permite el acceso directo a los dispositivos FTDI mediante su *driver D2XX*, el cual hay que instalar en el PC [10]. También se apoya en **serial.tools.list\_ports** para detectar puertos disponibles y en **PySide6.QtCore** para la gestión de señales (**Signals**) y temporizadores (**QTimer**). Estas dependencias son necesarias para poder realizar las acciones que se comentan en el siguiente diagrama de flujo de esta clase:

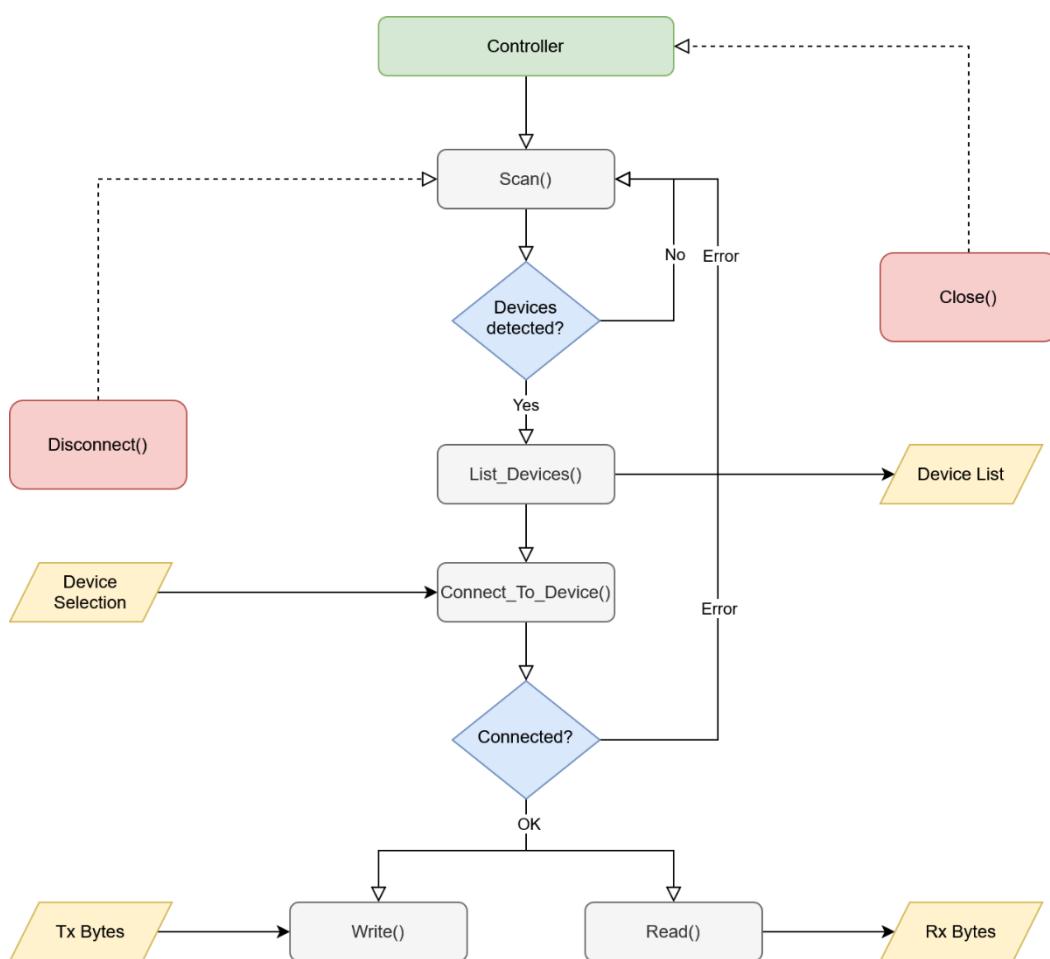


Figura 3.27 Diagrama de flujo de clase Controller

Una vez se construye el correspondiente objeto “controlador” de la clase **“FTDIController”**, se procede a escanear los puertos serie del PC en busca de dispositivos. Este escaneo se realiza con un periodo de 1s (parámetro constante de la clase y no configurable por el usuario), mientras se van listando los dispositivos encontrados, a través de las librerías ***serial.tools.list\_ports*** y ***ftd2xx*** para obtener detalles como VID, PID, número de serie y descripción. Hasta que se reciba el evento externo de conexión junto con el puerto seleccionado.

Internamente, esta función **`connectToDevice`**, llama a las funciones de la librería FTDI, abriendo conexión con el dispositivo y configurándole diversos parámetros de comunicación, entre ellos los *timeouts* y los tamaños de los buffers de lectura y escritura, entre otros. En este caso, no haría falta ponerlo en modo FIFO 245 asíncrona, pues ya se hizo a través de su propia app, guardándolo directamente en su EEPROM. Una vez se ha establecido la conexión correctamente, el controlador está disponible para atender las peticiones externas de escritura y/o lectura que llegarán desde el **Mediator**. La acción de escritura, **`Write`**, necesita como entrada los bytes a transmitir y, por otro lado, la de lectura, **`Read`**, devolverá los bytes recibidos. En cualquier momento, puede llegar un evento externo de desconexión (**`Disconnect`**), que hará que se reanude de nuevo el flujo, activando el escaneo periódico.

## Image Processor

La clase “**ImageProcessor**” es la encargada del procesamiento de imágenes recibidas desde la FPGA, incluyendo la conversión de datos brutos o bytes en imágenes visualizables en el GUI, así como la ejecución de un modelo neuronal para el reconocimiento automático de señales de tráfico relacionadas con los límites de velocidad.

Está implementada como un componente con capacidad de trabajar en segundo plano mediante su propio hilo de procesamiento. Este módulo recibe *frames* completos desde la clase Mediator y una vez pasados por el modelo, emite una señal (**Qt Signal**) **imgReadyToDisplay** hacia el Mediator y este, a su vez hacia la Vista. Esta última actualiza la imagen a visualizar y los resultados del modelo. No accede directamente a la GUI ni al controlador y además trabaja sobre un modelo previamente cargado. A continuación, se muestra el diagrama de flujo de esta clase:

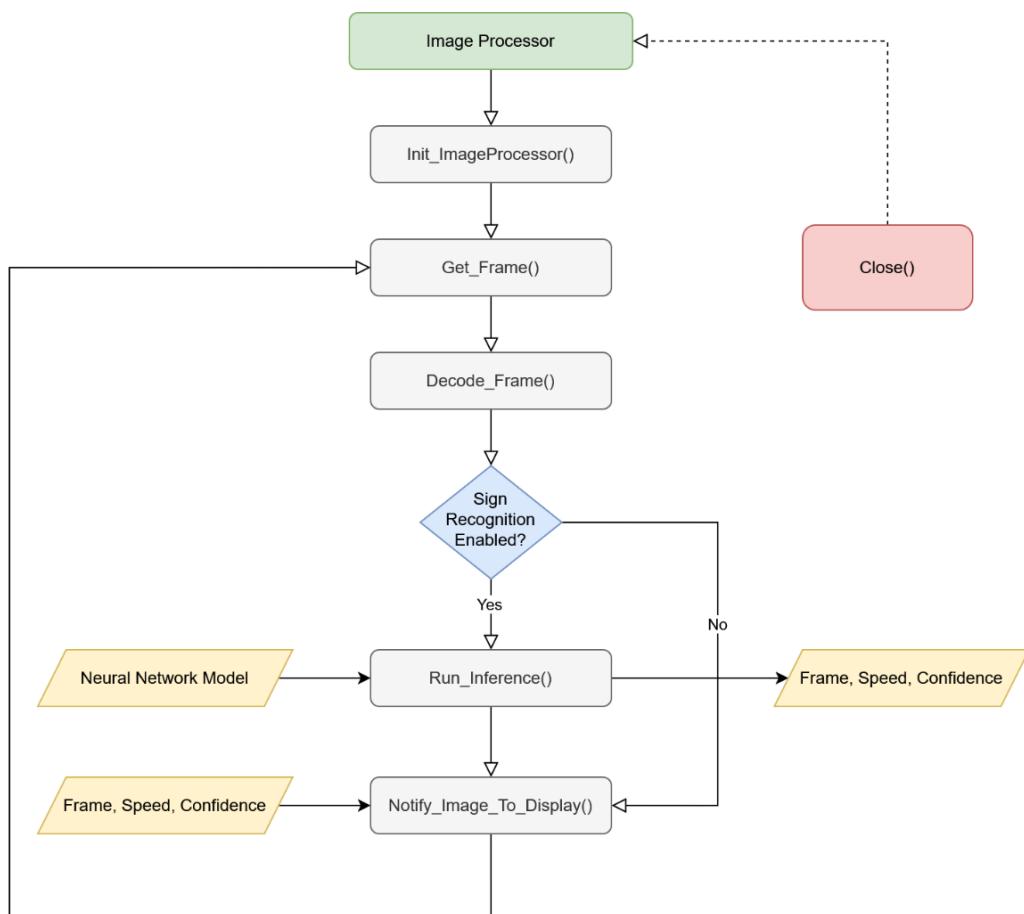


Figura 3.28 Diagrama de flujo de clase *Image Processor*

En primer lugar, al inicializar la clase, se carga un modelo neuronal YOLO (best.pt) previamente entrenado. Este proceso de entrenamiento se detallará más tarde.

Una vez cargado, se pone en marcha el hilo “**ProcessorWorker**”, que una vez recibe un frame completo, ejecuta la inferencia del modelo (`_runInference`) sobre este para detectar y reconocer señales de tráfico. Para desacoplar el flujo de adquisición de imágenes del proceso de inferencia, se emplea una cola tipo FIFO (queue.Queue) con longitud limitada (`FRAME_QUEUE_LEN = 16`). Esta cola actúa como buffer intermedio, permitiendo añadir imágenes sin bloquear la adquisición, descartar frames antiguos si el procesamiento está más lento de lo esperado, evitando saturaciones o desbordamientos, además de adaptarse dinámicamente al tiempo de inferencia gracias al parámetro `MAX_PROCESS_FRAMES_STEP`.

Los datos de imagen que llegan a la cola están codificados en formato **YUV4:2:2**, correspondiente al formato de salida del sensor de imagen (real o emulado) conectado a la FPGA . Antes de pasarle la imagen al modelo de inferencia, el frame debe ser decodificado correctamente y convertido al formato compatible con **PyTorch** y con **OpenCV** (RGB888 o Grayscale 8-bit).

Este proceso de decodificación (`Decode_Frame`), dependiendo del formato de frame configurado, color (`FRAME_COLOR_SIZE`) o escala de grises (`FRAME_GRAY_SIZE`), espera una determinada cantidad de bytes u otra. En este caso, **el modelo únicamente es compatible con imágenes en formato color**, ya que fue entrenado de esta forma. Se convierte la imagen o *frame* recibido en formato RGB compatible con **OpenCV**. Posteriormente, se convierte a un objeto tipo **QPixmap** para poder mostrarla en la interfaz PySide. Además de esto, la inferencia con **YOLOv8**, requiere como parámetros de entrada, la redimensión de la imagen a **512x512**, formato tipo matriz **numpy uint8** en formato **RGB**, un valor de confianza mínima (en este caso 0.85), umbral bajo el cual se descartarán las detecciones. Todo esto debido a que las imágenes durante el entrenamiento presentaban este formato [11].

Una vez se lleva a cabo la inferencia si el usuario ha habilitado el reconocimiento, se filtran los resultados, seleccionando la detección con mayor valor de confianza por encima del umbral establecido (0.85), la cual se guarda y envía al GUI.

### 3.2.4 Modelo Neuronal

El núcleo funcional del módulo software es un modelo neuronal entrenado para detectar y reconocer señales de tráfico relacionadas con los límites de velocidad.

A continuación, se detallan las fases para su obtención:

#### 1. Selección y filtrado del conjunto de datos

El dataset inicial utilizado es el de **GTSRB** (*German Traffic Sign Recognition Benchmark*), ampliamente conocido en el ámbito del reconocimiento de señales de tráfico [12]. Sin embargo, este conjunto incluye más de 40 clases de señales, lo que complica el entrenamiento cuando el objetivo del proyecto es detectar únicamente las señales de límites de velocidad.

Por tanto, se lleva a cabo un proceso de filtrado y remapeo de clases y conversión del dataset. Permitiendo reducir la complejidad del modelo, el tiempo de entrenamiento y centrar los recursos computacionales en el objetivo real del sistema.

#### 2. Conversión al formato YOLO

Una vez filtradas, se reorganizan las imágenes y se generan las anotaciones en *class\_mapping.txt*, con el formato requerido con **YOLO**, junto con la estructura de carpetas estándar:

```
yolo_dataset_filtered/
├── images/
│   ├── train/
│   └── test/
└── labels/
    ├── train/
    └── test/
```

Esto se lleva a cabo de forma automatizada con el script [python convertDatasetIntoYOLOformat\\_new.py](#).

El resultado es un dataset listo para ser procesado por el *framework* de **Ultralytics**.

### 3. Configuración del dataset

Una vez hecho esto, se genera el archivo *yolo\_dataset.yaml* con las rutas relativas de las imágenes y las clases finales:

```
train: yolo_dataset_filtered/images/train  
val: yolo_dataset_filtered/images/test  
  
nc: 9 # Número de clases (límites de velocidad + clase inválida)  
names: ["20", "30", "50", "60", "70", "80", "Invalid", "100", "120"]
```

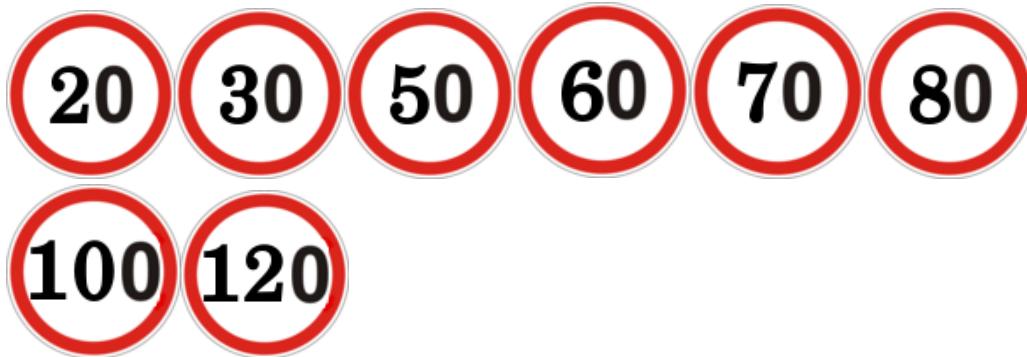
Figura 3.29 Archivo *yolo\_dataset.yaml*

La clase inválida se ha decidido incluir ya que se corresponde con esta señal:



que podría inducir a error al sistema, por lo que se decide añadir al modelo para que sea capaz de detectarla y diferenciarla de las demás también.

Por lo que, actualmente, las clases que el sistema es capaz de reconocer son las siguientes:



#### 4. Entrenamiento del modelo

Se selecciona la versión **YOLOv8n** (nano), por su equilibrio entre precisión y eficiencia para ejecución en hardware embebido (posible caso futuro).

El entrenamiento se realizó mediante el script *train\_yolo.py*.

```
model = YOLO('yolov8n.pt')
model.train(
    data='yolo_dataset.yaml',
    epochs=20,
    imgsz=512,
    batch=4,
    verbose=True
)
```

Figura 3.30 Parte del código del archivo *train\_yolo.py*

Se usa el modelo pre-entrenado *yolov8n.pt*. Se entrena durante 20 épocas (**epochs**), con un tamaño de imagen de entrada de 512x512 (**imgsz**), y **batch** 4, es decir, el número de imágenes que se procesan simultáneamente antes de actualizar los pesos del modelo durante el entrenamiento. Con este valor, se busca un equilibrio entre precisión y rendimiento para su despliegue en sistemas con recursos limitados (modelos ligeros como YOLOv8n).

Este proceso genera automáticamente el archivo final *best.pt* del modelo entrenado.

## 5. Evaluación y resultados

Finalizado el entrenamiento, se evalúa el modelo automáticamente sobre el conjunto de validación (`model.val`). En la Figura 3.31 se presentan las curvas de entrenamiento y validación del modelo.

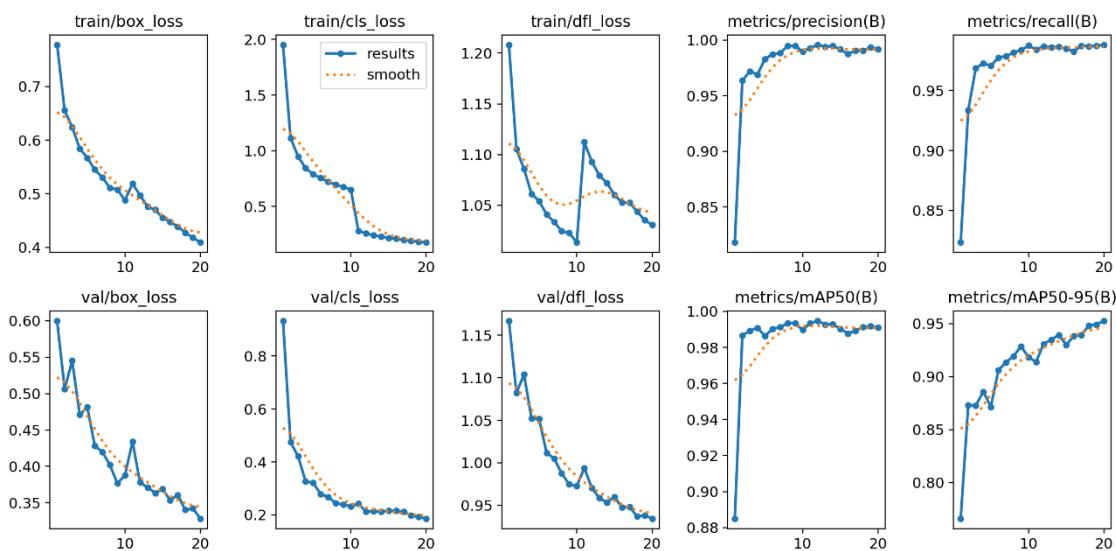


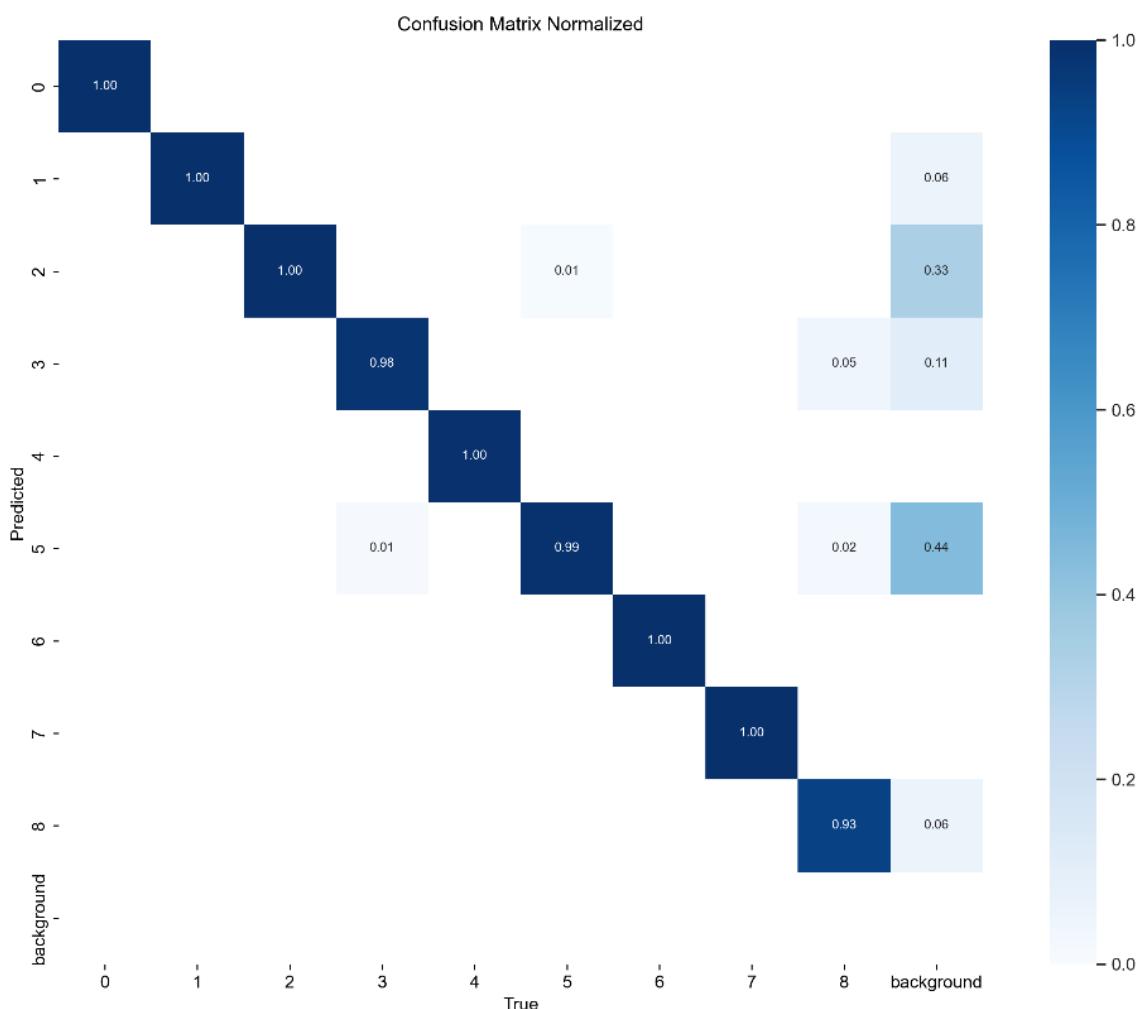
Figura 3.31 Resultados entrenamiento del modelo

Las gráficas de ***box\_loss***, ***cls\_loss*** y ***dfl\_loss*** muestran la evolución de las funciones de pérdida asociadas a la localización de los cuadros delimitadores, la clasificación de las señales y la regresión de las coordenadas, respectivamente. En todos los casos se observa una disminución progresiva de la pérdida a medida que avanza el número de épocas, lo que indica que el modelo mejora su capacidad predictiva con el entrenamiento.

Por otro lado, las métricas de evaluación (***precision***, ***recall*** y ***mAP***) reflejan la calidad del modelo sobre el conjunto de validación. La precisión y el recall alcanzan valores próximos a 1, lo que significa que el modelo apenas genera falsos positivos ni falsos negativos. Asimismo, la métrica ***mAP@50*** y ***mAP@50-95***, que combinan precisión y recall en distintos umbrales de intersección sobre la unión (IoU), presentan una tendencia creciente hasta valores muy elevados (**>0.95**), confirmando que el detector generaliza correctamente.

En conjunto, estas curvas muestran que el entrenamiento fue exitoso: el modelo converge, las pérdidas se reducen y las métricas de evaluación alcanzan valores que evidencian un rendimiento muy alto.

La Figura 3.32 muestra la matriz de confusión normalizada del modelo evaluado sobre el conjunto de validación.



*Figura 3.32 Matriz de confusión del modelo*

En el eje horizontal se representan las clases reales y en el eje vertical las clases predichas por el modelo. Los valores en la diagonal indican las predicciones correctas, mientras que los valores fuera de ella representan errores de clasificación.

Se observa que la mayoría de los valores de la diagonal están muy próximos a 1, lo que significa que cada clase de señal de tráfico es identificada con gran precisión. Los valores fuera de la diagonal son reducidos, aunque en algunos casos existe cierta confusión entre señales de límites cercanos, lo cual es esperable dado su parecido visual.

Estos resultados confirman que el modelo no solo presenta métricas globales altas, sino que también logra un comportamiento consistente clase por clase,

manteniendo la tasa de error muy baja incluso en señales con características similares.

También, se puede visualizar de forma interactiva estas gráficas a través de la propia página web de **TensorBoard**, lanzada con el script [13]:

```
from tensorboard import program
tb = program.TensorBoard()
tb.configure(argv=[None, '--logdir', 'runs/detect/train'])
tb.launch()
```

## 6. Formato del modelo

En este proyecto, se trabaja directamente con el archivo “**best.pt**”, esto es, en formato **PyTorch**.

Para otras aplicaciones, se podría necesitar exportar el modelo a otro formato de los existentes:

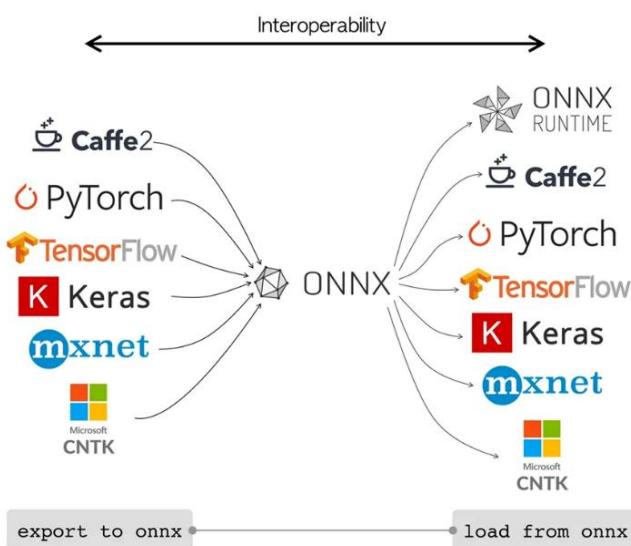


Figura 3.33 Tipos de modelos

Por ejemplo, el formato que ofrece mayor interoperabilidad es **onnx** (Open Neural Network Exchange), que se obtiene con:

```
model.export(format='onnx')
```

## 7. Arquitectura del modelo

La arquitectura empleada en este proyecto corresponde al modelo YOLOv8n (nano), una versión ligera y optimizada para tareas de detección en tiempo real con recursos computacionales limitados. Su diseño sigue un enfoque jerárquico y modular, dividiendo el flujo de datos en diferentes bloques funcionales. La siguiente figura representa el grafo computacional del modelo cargado desde *best.pt* [14]:

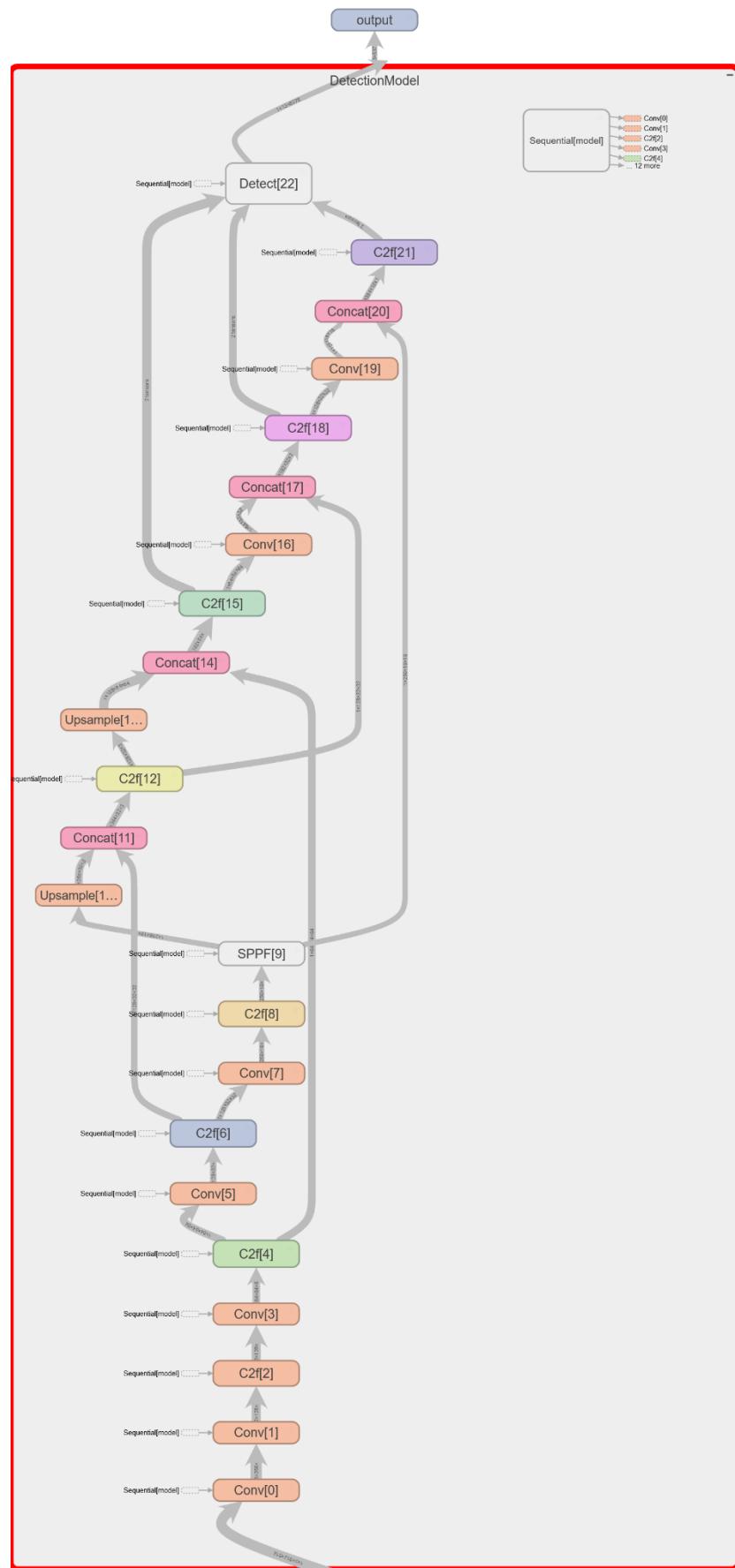


Figura 3.34 Estructura de la red neuronal

## 1. Estructura general

- **Input (entrada)**

La imagen de entrada (normalizada y redimensionada a 512x512) es inyectada en la parte inferior del modelo.

- **Output (salida)**

El bloque superior (Detect[22]) genera como salida:

- Coordenadas de las cajas delimitadoras (*bounding boxes*).
- Clases detectadas.
- Valores de confianza.

## 2. Bloques funcionales principales

- **Conv (Convoluciones)**

Extraen características espaciales básicas como bordes, esquinas o texturas.

- **C2f (Cross Stage Partial)**

Mejoran la eficiencia del entrenamiento dividiendo y fusionando flujos de información.

- **Concat (Concatenación)**

Fusionan información de distintas etapas del modelo para mejorar la capacidad de detección, especialmente de objetos pequeños.

- **Upsample (Subida de resolución)**

Permiten recuperar información de alta resolución perdida durante el *downsampling*, vital para detectar objetos pequeños.

- **SPPF (Spatial Pyramid Pooling-Fast)**

Mezcla información de múltiples escalas espaciales para proporcionar contexto global. Mejora la robustez del modelo frente a variaciones de tamaño y ubicación.

- **Detect**

Es el bloque final de detección (Detect[22]). Combina todas las características extraídas y genera las predicciones finales (*boxes, class, confidence*).

## 3. Ventajas del diseño jerárquico

Este tipo de arquitectura con múltiples resoluciones en paralelo permite detectar objetos pequeños (con capas de alta resolución) y grandes (con capas más profundas y resumidas), además de, minimizar pérdida de información gracias a las conexiones tipo **Concat**.

Se utilizó el paquete **Ultralytics** para la carga, entrenamiento y evaluación del modelo:

```
pip install ultralytics
```

Tras una primera fase de pruebas del modelo, se vio que funcionaba correctamente con imágenes con ruido, ya que había sido entrenado con un *dataset* (GTSRB) pensado para situaciones reales de conducción. Por este motivo, se optó por realizar un **proceso de fine-tuning** empleando un *dataset* (custom\_dataset) adicional compuesto por imágenes más nítidas, con el fin de mejorar la capacidad del modelo para reconocer señales en condiciones de alta calidad visual y sin degradación [15]. De esta forma se consiguió un equilibrio entre robustez frente a ruido y precisión en contextos más controlados.

La documentación completa del proceso de entrenamiento y **fine-tuning** del modelo (*best\_finetune.pt*), junto con los *scripts* empleados, están disponibles en la carpeta [Software/Neural Network](#) del repositorio de GitHub.

# Capítulo 4 Pruebas y validación

## 4.1 Módulo Hardware

Este capítulo recoge el proceso de verificación funcional llevado a cabo sobre los distintos bloques hardware desarrollados en VHDL, que componen el sistema completo embebido implementado en la FPGA. El objetivo es asegurar el correcto funcionamiento de cada componente por separado y del sistema completo en condiciones reales y simuladas.

Para ello, se han diseñado una serie de *testbenches* y scripts de simulación TCL que automatizan las tareas de simulación y permiten observar el comportamiento de señales internas en diferentes escenarios. Estas simulaciones se realizan en el entorno **Xilinx Vivado 2019.1**, en concreto, con su herramienta Vivado Simulator (ver figura) integrada en la **suite Vivado HLx WebPack** [16].

Para comenzar una simulación, se selecciona la opción “**Run Behavioral Simulation**”. Esta realiza una simulación a nivel RTL, verificando la lógica y FSM antes de la síntesis:

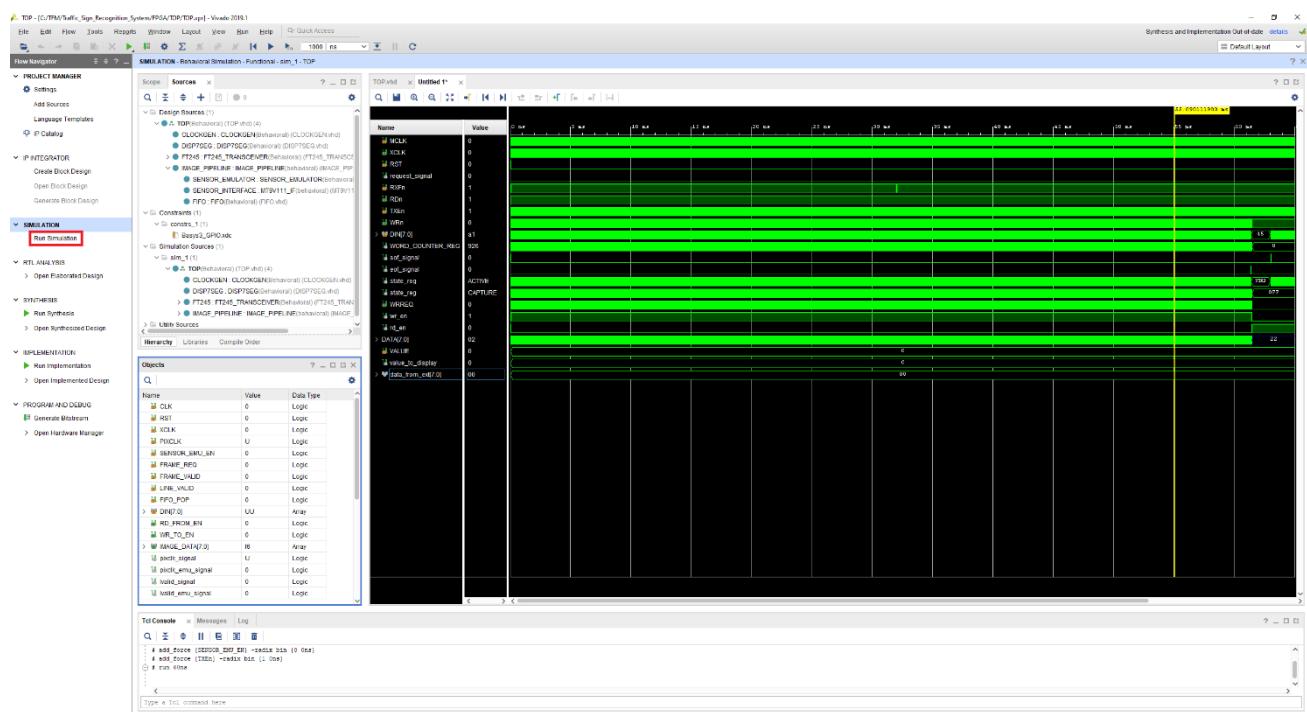


Figura 4.1 Vista de la simulación en Vivado

A continuación, se describen las distintas ventanas disponibles en la vista que aparece al ejecutar este tipo de simulación:

- **Waveform**

Esta es la ventana principal de visualización de señales. Muestra cómo evolucionan las señales en el tiempo. Permite hacer zoom, medir tiempos entre transiciones y detectar problemas como *glitches* o retardos no deseados.

- **Objects**

Lista todas las señales internas y externas del módulo TOP o la jerarquía seleccionada. Desde aquí, se pueden añadir al *waveform* para visualizarlas.

- **Scopes**

Muestra la jerarquía del diseño, útil si el diseño tiene varios niveles como es el caso. Te permite navegar por los submódulos y ver sus señales internas.

- **Console y TCL console**

Muestra mensajes, errores, advertencias o logs durante la simulación. La consola TCL permite lanzar comandos directamente o cargar scripts.

En primer lugar, se realizan pruebas unitarias por bloques, simulando cada una de las partes de cada uno de ellos por separado. Para posteriormente, simular cada bloque completo individualmente.

Las otras opciones de simulación, post-síntesis y post-implementación tienen el objetivo de simular la *netlist* generada, y no el modelo RTL descrito.

#### 4.1.1 FT245\_IF

Esta prueba se centra en validar el correcto funcionamiento del bloque **FT245\_IF**. Como se mencionó en apartados anteriores, este bloque contiene una FSM que controla las señales involucradas en la comunicación entre FPGA y PC por medio del módulo UM232H-B.

Como se comentó en el capítulo de diseño e implementación hardware, este bloque está compuesto por dos componentes **IF\_WRITE** y **IF\_READ**, correspondientes a los interfaces de escritura y lectura respectivamente.

En primer lugar, se presenta la simulación que muestra el comportamiento de **IF\_WRITE** ante un proceso de escritura asíncrona, lo que sería una transmisión desde la FPGA al PC. Para ello se ejecuta el script *FT245\_WRITE\_async\_IF\_sim.tcl*, que automatiza la compilación del diseño, el lanzamiento de la simulación y la visualización de las señales clave.



Figura 4.2 Simulación ciclo de escritura de **IF\_WRITE**

Analizando la imagen, se puede ver la evolución de las señales de control del interfaz (**WR\_EN**, **WRn**, **WRREQ**, **READY**) ante diferentes datos de entrada (**DIN**) y activación o desactivación de la señal **TXEn** (**TXE#** en el chip FTDI), clave para iniciar y terminar el ciclo de escritura. El **FT245** emplea esta señal para indicar si tiene o no el buffer interno listo para recibir datos. Por tanto, se aprecia que la FPGA se queda en el estado **WAIT\_FOR\_TXE**, tras pasar a este estado desde **IDLE** cuando **WR\_EN=‘1’**, hasta que la versión sincronizada de **TXEn** (**TXEn\_sync**) baja a ‘0’, iniciándose el ciclo de escritura, lo cual se refleja en la transición al estado **WRITE\_DATA** y en la activación de la señal **WRn** a nivel bajo. Momento en el cual se activa **WRREQ**, señal que representa una petición de dato hacia un componente externo responsable de suministrar la información a transmitir. Además, se ve como el dato de entrada, **DIN**, sale del interfaz de escritura (**DOUT**) hacia la FPGA. Se aprecia cómo el dato de entrada **0x6e** se pierde por no cumplir los requisitos de timing del FT245, al no coincidir la disponibilidad del dato con el instante en el que el FT245 indica que está listo para transmitir activando **TXEn\_sync**. Sin embargo, el siguiente (**0x1e**) sí que se captura correctamente, al cumplir estas condiciones. La señal **READY** activa, simplemente indica que el interfaz está en reposo, esto es, en estado **IDLE**.

En cada uno de los ciclos se ve como el interfaz cumple con los requisitos que impone el FT245:

- ✓ **T6:** WRn activo tras TXEn inactivo  $\geq 1\text{ns}$  (WRn baja un ciclo después que TXEn\_sync = '0').
- ✓ **T10:** Pulso activo de WRn (duración)  $\geq 30\text{ns}$  (NWCYCLES = 4  $\rightarrow 40\text{ns}$ )
- ✓ **T8:** Setup de datos antes de WRn  $\downarrow \geq 5\text{ns}$  (DIN ya estable cuando WRn se activa).
- ✓ **T9:** Tiempo de hold de datos tras WRn  $\downarrow \geq 5\text{ns}$  (DIN se mantiene a la entrada del interfaz hasta después de WRn  $\uparrow$ ).

Se ha obtenido que cada byte transmitido requiere aproximadamente **90ns**. Esto equivale a una **tasa teórica de transferencia** de alrededor de **11 MB/s** ( $90\text{ns} \times 1 \text{ byte} \rightarrow 1/90\text{ns} \approx 11,1 \times 10^6 \text{ bytes/s}$ ).

No obstante, el sistema no puede alcanzar dicha velocidad en la práctica, ya que el **FT232H**, configurado en modo **FT245 FIFO asíncrono**, presenta una limitación máxima de transferencia de **8 MB/s**, tal y como se especifica en el datasheet del fabricante. Esto significa que, aunque la lógica de la FPGA esté en condiciones de generar un ancho de banda mayor, la restricción real la impone el propio dispositivo USB, fijando el rendimiento máximo de la comunicación.

A partir de esta limitación, la tasa efectiva de imágenes que pueden transmitirse al PC depende del tamaño de cada trama. En el caso del formato color YUV422 ( $640 \times 480$  píxeles, 2 bytes por píxel), cada imagen ocupa **614 400 bytes**, mientras que en el formato monocromo (1 byte por píxel) la imagen ocupa **307 200 bytes**. Dividiendo el ancho de banda máximo (8 MB/s) entre estos tamaños de trama, se obtiene un límite superior aproximado de **13FPS en color** y **26FPS en escala de grises**. Como se verá posteriormente, estos valores coinciden con las medidas obtenidas en las pruebas reales.

A continuación, se presenta la simulación que muestra el comportamiento de **IF\_READ** ante un proceso de lectura asíncrona, lo que sería una transmisión desde el PC a la FPGA. Para ello se ejecuta el script **FT245\_READ\_async\_IF\_sim.tcl**, que automatiza la compilación del diseño, el lanzamiento de la simulación y la visualización de las señales clave.

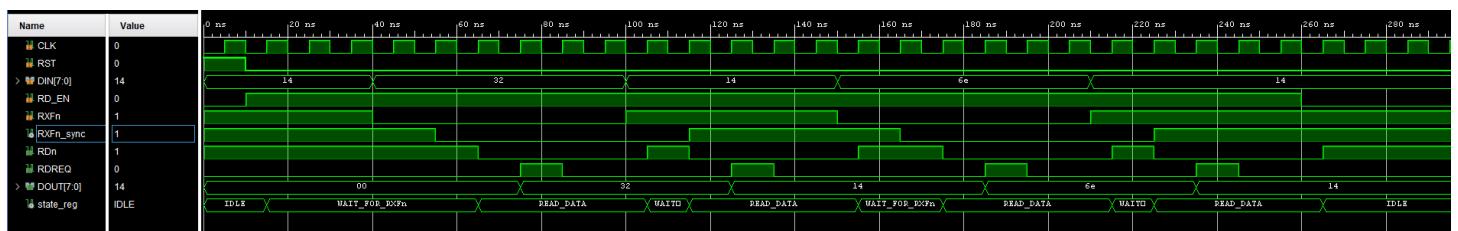


Figura 4.3 Simulación ciclo de lectura de IF\_READ

Aquí, se muestra el comportamiento de las señales de control (**RDn**, **RDREQ**) como la respuesta del interfaz a la señal **RXF<sub>n</sub>** que le indica disponibilidad de datos en el buffer del **FT245** (**RXF<sub>n</sub>**, que es RXF# en el chip FTDI). El flujo de operación de lectura comienza con el interfaz en estado **IDLE** o reposo hasta que se habilita la lectura, esto es, **RD\_EN** = '1'. En ese momento, pasa al estado **WAIT\_FOR\_RXFn**, donde permanece hasta que la versión sincronizada de **RXF<sub>n</sub>** (**RXF<sub>n\_sync</sub>**) se activa a nivel bajo. Simulando así el aviso del FT245 de que tiene datos disponibles para el interfaz. Instante en el que se pasa al estado **READ\_DATA**, activándose **RDn** a nivel bajo, iniciando así el proceso de lectura del dato a la entrada del interfaz (**DIN**). A su vez, se activa a nivel alto **RDREQ**, que es una señal de salida de este componente que indica que el dato **DOUT** está disponible para ser consumido por otro bloque externo. El ciclo de lectura dura **NRCYCLES** = 4 ciclos de reloj CLK, como se puede comprobar midiendo el tiempo que permanece la FSM del interfaz en el estado **READ\_DATA** (tiempo entre la activación y desactivación de **RDn**). Si la lectura sigue habilitada como es el caso (**RD\_EN**), se vuelve al estado de espera de la señal de inicio de lectura **RXF<sub>n\_sync</sub>**. Por lo que vemos, que en la simulación se han realizado 4 ciclos de lectura, donde el segundo y cuarto ciclo permanece menos tiempo esperando la activación de **RXF<sub>n\_sync</sub>**, ya que cuando vuelve a este estado se la encuentra ya activa.

Tras este último ciclo de lectura, finalmente se ha deshabilita, volviendo así la FSM al estado **IDLE**.

En cada uno de los ciclos se ve como el interfaz cumple con los requisitos del FT245 que impone solo a partir del inicio de lectura, no durante la espera:

- ✓ **T5**: RDn activo tras flanco de bajada de RXFn  $\geq$  0ns
- ✓ **T4**: Pulso de RDn activo  $\geq$  30ns (**NRCYCLES** = 4  $\rightarrow$  40ns)
- ✓ **T1**: RDn inactivo antes de nuevo RXFn (flanco de subida de RDn y espera)

#### 4.1.2 DISP7SEG

Este bloque se encarga de mostrar un valor numérico natural de 3 dígitos, esto es, desde el 0 al 999 en el display de 4 dígitos de 7 segmentos de la BASYS 3. Internamente, divide el valor en centenas, decenas y unidades, para activar uno a uno cíclicamente mientras apaga el resto mediante las señales **AN[3 : 0]**, dando sensación de que se muestran a la vez.

Se puede ver en la simulación realizada lanzando el script *DISP7SEG\_sim.tcl*, que, cuando a la entrada se tiene **VALUE = 213**, este valor se descompone, reflejándose el resultado en la señal **digits[2:0] = "2, 1, 3"**.

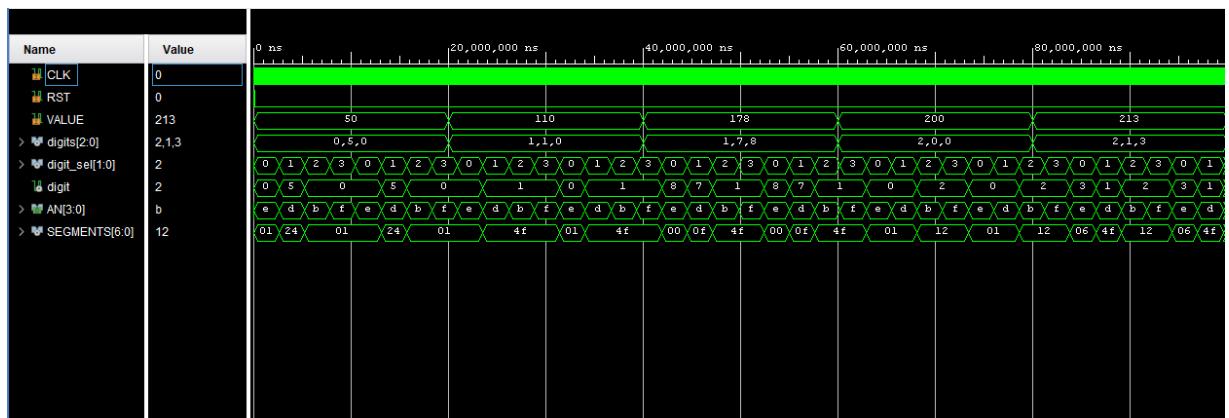


Figura 4.4 Simulación visualización de valores en DISP7SEG

La señal **digit\_sel** va rotando entre **00**, **01** y **10**, seleccionando así el dígito activo en cada ciclo de refresco. La señal **digit** por su parte, contiene el número a mostrar en ese instante (esto es 3, 1, 2 en orden).

Como se ha comentado, **AN[3:0]** controla el encendido del dígito en cada instante, como es un display de ánodo común se activa a nivel bajo. Por tanto, se puede ver como **AN** va rotando entre **1011**, **1101** y **1110**, indicando que el bloque está rotando el encendido de los dígitos 0, 1 y 2. **El cuarto no tiene uso en esta aplicación** (se mantiene apagado a nivel alto, '1').

Por último, **SEGMENTS [7 : 0]** representa los 7 segmentos que forman el número (sin incluir el punto decimal, DP). Cada valor hexadecimal representa una combinación de segmentos activos, se observa, por tanto, cómo cambia el valor correspondiente al dígito actual (por ejemplo, **0x4f** corresponde a los segmentos activos para visualizar el "3").

### 4.1.3 CLOCKGEN

Este módulo tiene como propósito generar una señal de reloj **CLKGEN** de menor frecuencia a partir de una señal de referencia **CLKREF**. Esto se realiza mediante un contador controlado por el parámetro **NCYCLES**.

A continuación, se muestra una captura de la simulación obtenida tras lanzar el script:

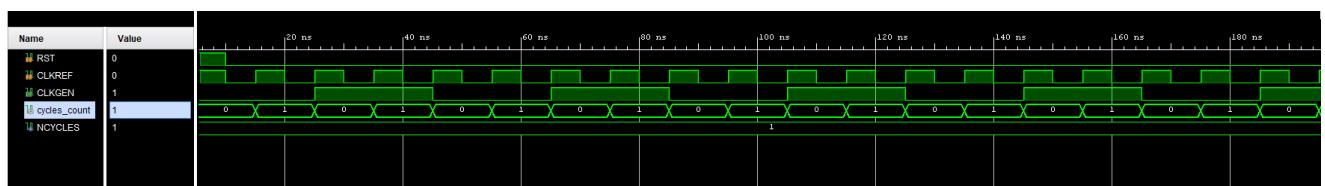


Figura 4.5 Simulación de generación de reloj 25MHz con CLOCKGEN

En esta imagen se muestra la entrada al bloque, **CLKREF**, que es la señal de reloj usada como referencia. En este caso, se define **NCYCLES = 1**, que es el valor máximo que alcanzará el contador **cycles\_count**, que cuenta cada flanco de subida de **CLKREF** hasta llegar a **NCYCLES**, momento en el que se reinicia. Este contador provoca que **CLKGEN** alterne entre '0' y '1' cada dos ciclos de **CLKREF**, produciendo así una señal con  $\frac{1}{4}$  de la frecuencia original. En este caso, la de entrada es la del sistema **100MHz**, y, por tanto, la generada es de **25MHz**.

Finalmente, para el proyecto se genera una señal de **12.5MHz**, lo que significa que **NCYCLES = 3**.

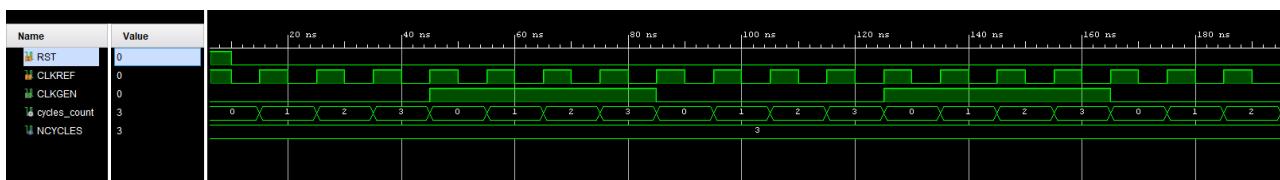


Figura 4.6 Simulación de generación de reloj 12.5MHz con CLOCKGEN

Esta frecuencia se ha seleccionado por ser la necesaria para garantizar la transmisión de tramas completas en **formato color**, requisito indispensable dado que el **modelo neuronal entrenado trabaja exclusivamente con imágenes a color**. Como se detallará en apartados posteriores, el uso de imágenes en escala de grises no es compatible con el modelo de detección desarrollado.

#### 4.1.4 IMAGE\_PIPELINE

Este bloque constituye el núcleo de procesamiento intermedio entre la interfaz del sensor de imagen y los módulos que escriben o procesan los datos. Su función principal es capturar, almacenar temporalmente y transferir los datos de imagen digitalizados de forma segura y sincronizada.

#### FIFO

Uno de los componentes esenciales de este subsistema es la memoria **FIFO** (*First In, First Out*), encargada de almacenar los bytes de imagen capturados para ser consumidos por los siguientes módulos (como el interfaz de transmisión al PC o el procesador de señales). Esta cola permite desacoplar dos dominios temporales distintos (captura y transmisión), y gestiona correctamente los flujos de entrada y salida.

Este componente es una cola de almacenamiento síncrona, parametrizable en ancho de datos (**DATA\_WIDTH**), que es el número de bits que almacena cada posición o celda de la FIFO y el ancho de dirección, que es el número de bits necesarios para direccionar todas las posiciones de la FIFO (**ADDR\_WIDTH**). Lo que se traduce en que si, por ejemplo, **ADDR\_WIDTH** = 2, se tendrá una FIFO que podrá almacenar hasta  $2^2 = 4$  posiciones de datos (profundidad). Sabiendo esto, se simula el funcionamiento de la FIFO a través de un script, obteniendo el siguiente resultado:

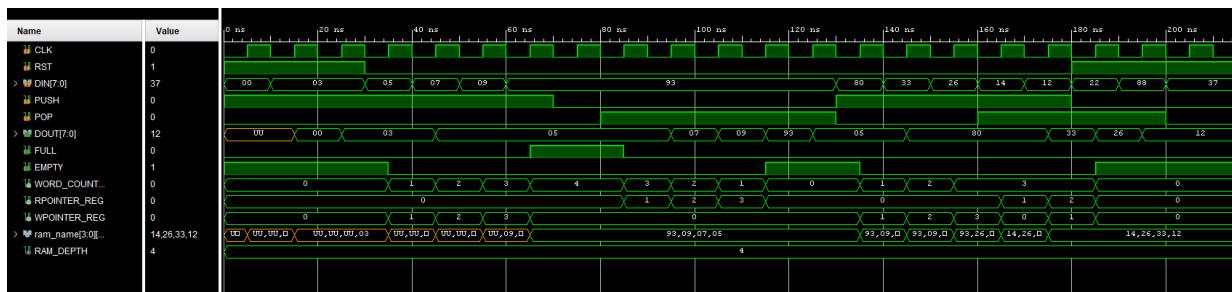


Figura 4.7 Simulación funcionamiento de FIFO

Aquí, se aprecia como se escriben los valores de **DIN**: **0x03**, **0x05**, **0x07** y **0x09** entre 0 y 90ns al activar **PUSH** = '1'. A su vez, el contador **WORD\_COUNTER\_REG** se incrementa hasta 4, indicando el número de palabras o valores almacenados en la **FIFO** en cada instante. La señal **FULL** se activa cuando se alcanza la capacidad máxima (**RAM\_DEPTH** = 4). El puntero de escritura de datos refleja la posición donde se encuentra el último dato guardado, rotando cíclicamente entre 0 y 3, en este caso de ejemplo simulado. Se puede ver como el contenido de **ram\_name** se actualiza correctamente con los nuevos datos que se van escribiendo o guardando.

También, se aprecia como esta FIFO está diseñada de forma que, el dato de salida **DOUT** refleja siempre el valor almacenado en la posición actual del puntero de lectura, aunque no se active la señal **POP** = '1'. Este comportamiento permite observar el próximo dato disponible sin necesidad de consumirlo, evitando ciclos muertos y mejorando la eficiencia y sincronización entre el bloque de captura de imágenes y el que las consume.

Es, tras 100ns de simulación, cuando se comienza el proceso de lectura, mediante la activación de la señal **POP** = '1'. En cada ciclo de lectura, se ve cómo se va reduciendo el valor de **WORD\_COUNTER\_REG**, al ir sacando valores de la **FIFO**, rotando también el puntero **RPOINTER\_REG** y actualizando **DOUT** con el contenido leído en orden (**0x03** → **0x05** → **0x07** → ...). La señal **EMPTY** se activa al consumir el último dato de la FIFO.

A los 160ns, se realizan varias escrituras y lecturas simultáneamente, en el mismo ciclo de reloj. El contador de palabras se mantiene constante (**WORD\_COUNTER\_REG**) ya que entra un dato a la vez que sale otro. Los punteros de escritura y lectura se actualizan ambos en el mismo ciclo. La memoria (**ram\_name**) actualiza correctamente el estado de la FIFO.

Este comportamiento está previsto en el diseño de FIFO sincronizada. La condición esencial de este diseño es que PUSH no debe ocurrir cuando FULL = '1' y POP no debe ocurrir con EMPTY = '1', y eso también se verifica en el resultado de la simulación.

## SENSOR\_EMU

Este bloque implementa el emulador del sensor de imagen MT9V111 real. Esto permite validar todo el proceso de captura sin necesidad de disponer físicamente del sensor.

Para ello, se ejecuta un script de simulación, donde se ve como el emulador genera la señal de reloj **PIXCLK**, que es igual a la presente en su entrada, **XCLK**. Además, genera las señales de control **FRAME\_VALID** y **LINE\_VALID**, presentes en el sensor real. En **IMAGE\_DATA[7 : 0]** se tiene una imagen sintética generada en el propio componente, que es un simple patrón gradiente, de forma que sea fácilmente reconocible cuando se reciba y muestre en el software del PC. Todo este bloque, está controlado por una máquina de estados tal y como se explicó en el apartado .

En esta primera figura, se puede ver como el emulador del sensor inicia la transmisión del primer *frame*, activado la señal **FRAME\_VALID**. Transcurrido el periodo de **SOFBLANKING**, definido en el *datasheet* del MT9V111, este activa la señal de transmisión de su primera fila o línea de la imagen. Lo que significa que ya son datos válidos de la imagen, por eso el paso del estado **SOFBLANKING** a **ACTIVE**.

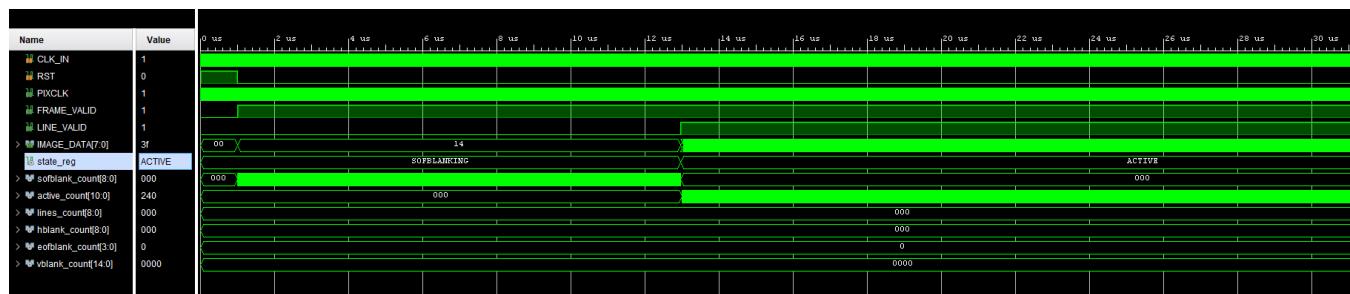


Figura 4.8 Simulación captura de imágenes con SENSOR\_EMU (parte 1).

En esta segunda captura, se puede ver como la FSM del componente comuta del estado **ACTIVE** a **HBLANKING** de forma continua, esto se debe a los periodos de BLANKING entre líneas válidas de la imagen activa. Este periodo se cuenta con su contador **hblank\_count** y durante el mismo la señal **LINE\_VALID** se desactiva.

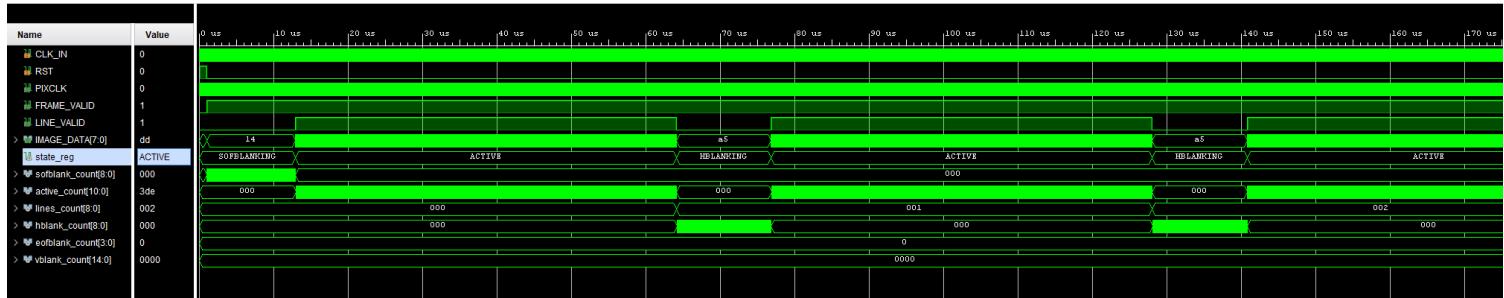


Figura 4.9 Simulación captura de imágenes con SENSOR\_EMU (parte 2).

Estos periodos de blanking en general permite a los consumidores o receptores de la imagen saber dónde empiezan y acaban líneas y tramas, es decir, sincronizarse con el sensor.

Finalmente, al llegar el contador de líneas válidas transmitidas al valor de la resolución de la imagen (**NUM\_LINES = 480**), el envío de la imagen ha concluido y por tanto se pasa al estado de *blanking* de final de trama, **EOFBLANKING**. Cuando se concluye este estado, se da por finalizada la trama actual y así se indica a través de la señal **FRAME\_VALID**, que estuvo activa durante todo el envío.

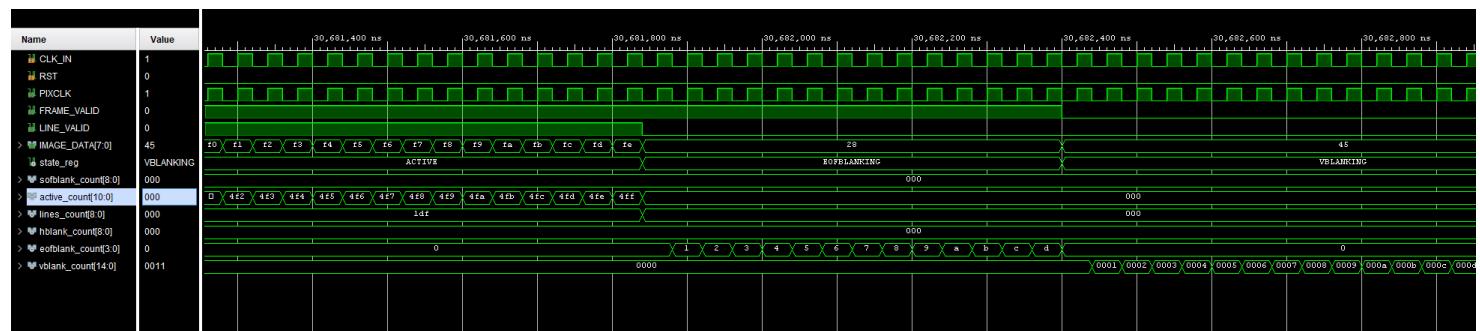


Figura 4.10 Simulación captura de imágenes con SENSOR\_EMU (parte 3).

Ahora, se pasa a otro período de blanking, **VBLANKING**, que es una pausa entre tramas para asegurar la sincronización de estas con el receptor.



Figura 4.11 Simulación captura de imágenes con SENSOR\_EMU (parte 4).

Una vez finalizado este periodo se comienza la transmisión de un nuevo *frame*, comenzando de nuevo por el periodo de *blanking* de inicio de este (SOFBLANKING). Como se ven en las imágenes, cada periodo o estado de la FSM tiene sus propios contadores, que son definidos con valores sacados del *datasheet*, como se indicó en el apartado 3.1.4 Sensor de imagen.



Figura 4.12 Simulación captura de imágenes con SENSOR\_EMU (parte 5).

## MT9V111\_IF

Este bloque implementa una máquina de estado o FSM parecida a la del emulador del sensor, ya que se trata del componente que hace de interfaz con éste y con el sensor real. Además de los diferentes estados de **BLANKING**, tiene un estado adicional **FLUSH**, al cual se llega cuando la FIFO se ha llenado, es decir, **FIFO\_FULL = '1'** o cuando ha concluido la recepción de un *frame*. En este estado, lo que se hace es vaciar la FIFO, asegurando así que no se pierden datos.

### 4.1.5 TOP

Sabiendo esto, se simula el sistema al completo, en primer lugar, se emula la captura de una imagen en formato color y a 25MHz, la frecuencia máxima configurable en este sistema. Todo esto modificando el proceso de captura del archivo **.vhdl** del interfaz del sensor (MT9V111\_IF), además de la frecuencia del reloj de entrada al emulador **XCLK**.

Una vez se ejecuta el script de simulación, se obtiene lo siguiente:

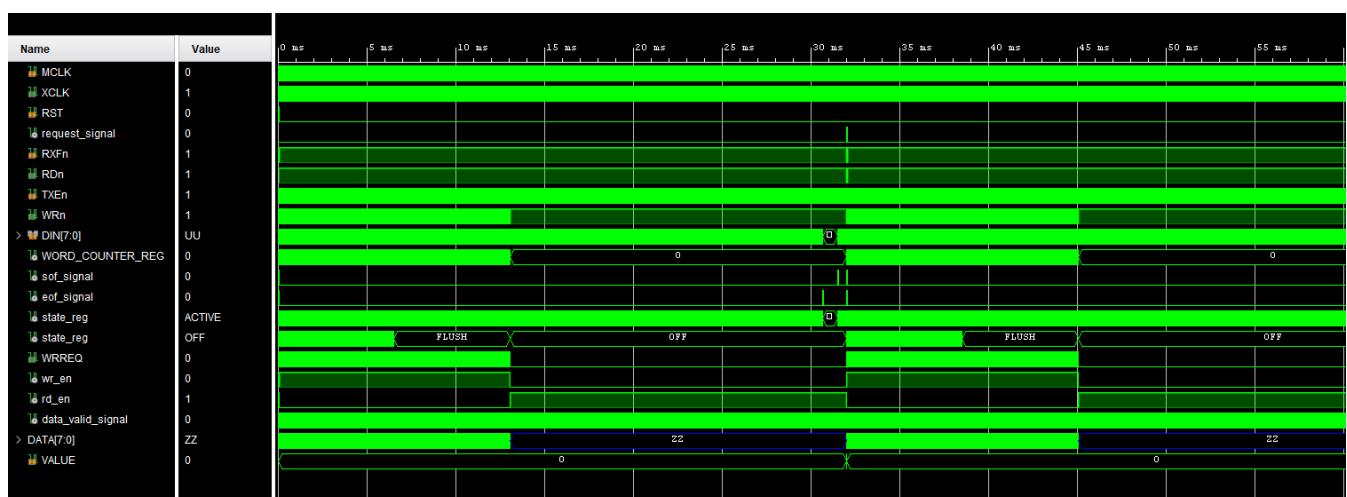


Figura 4.13 Simulación sistema completo (TOP) a 25MHz y frame a color

Aquí, se puede ver como con este ancho de datos y a esta frecuencia, el sistema no es capaz de capturar el frame completo y almacenarlo, entrando el interfaz de recepción en estado **FLUSH** antes de concluir el frame (indicado mediante la activación de **eof\_signal**).

A modo de prueba, se decide mantener la frecuencia, pero disminuir los datos a capturar, esto es, capturando la imagen en escala de grises en vez de a color.



Figura 4.14 Simulación sistema completo (TOP) a 25MHz y frame en escala de grises

Ahora sí, se ve como el sistema es capaz de llevar a cabo la captura de un *frame* completo sin saturarse.

Sin embargo, esto no nos sirve, pues el modelo neuronal entrenado y usado para este proyecto fue entrenado con imágenes a color por lo que no interpretará las imágenes en este formato.

Se decide por tanto dejarlo en formato color (YUV4:2:2), pero disminuir la frecuencia a la mitad, esto es, 12.5MHz. Cambiando esto, se obtiene el mismo resultado, se ve como el sistema ahora si es capaz de capturar el *frame*.



Figura 4.15 Simulación sistema completo (TOP) a 12.5MHz y frame a color

Calculando de la simulación y cotejándolo con el *datasheet* del sensor, se ve como el tiempo de *frame* será de 65ms aprox. Por lo que el sistema, en caso ideal tendrá una tasa máxima de 15FPS. En escala de grises es aún mayor, ya que el tiempo de frame a 25MHz, es de 32ms como se vio en la Figura 4.14, lo que se traduce en una tasa de 32FPS idealmente.

Únicamente, cuando el bloque interfaz de sensor ha finalizado la captura del frame y está en reposo, es cuando se permite la lectura desde el PC (señal `rd_en`). En esta imagen de una simulación de captura en escala de grises (tiempo de frame 32ms), se ha simulado un envío asíncrono del PC a la FPGA, una vez la captura y envío del frame al PC ha concluido y la FSM del interfaz del sensor está en estado IDLE. La petición de envío de dato desde el PC se refleja en el flanco de la señal `RXFn` en el instante 32ms. La señal `VALUE` es lo que le llega al *display 7 segmentos*, es decir se muestra el valor 80 (0x50 en hexadecimal, como se ve en el *buffer DATA*):



Figura 4.16 Simulación sistema completo (TOP) al recibir datos desde PC

Esta misma escritura de dato se interpreta en el sistema como una nueva petición de frame, de esta forma una vez procesa el PC el frame actual, reconoce la velocidad y la envía a la FPGA para visualizarla, al mismo tiempo le dice a esta que está listo para que le envíe el siguiente frame.

Para más información acerca de las simulaciones de los distintos módulos hardware, así como la localización de los scripts de simulación correspondientes a cada bloque, se remite al repositorio del proyecto en GitHub, donde se incluye la documentación completa en el archivo [README](#) de la carpeta **Hardware/FPGA\_Modules**.

## 4.2 Módulo Software

Una vez desarrollada la parte software, se realizaron distintas pruebas para verificar su correcto funcionamiento junto con la parte hardware, validando así el sistema completo. Estas pruebas permiten validar la fiabilidad de la interfaz gráfica (GUI), la comunicación con el módulo FTDI y el rendimiento del modelo neuronal para el reconocimiento de señales de tráfico.

### 4.2.1 Interfaz de usuario en modo DEBUG

Para ilustrar estas pruebas, se han capturado imágenes en distintos casos de prueba. En primera instancia, se empleó esta interfaz de debug (*SpeedTrafficSignRecognitionApp\_debug.ui*), que no es la que tendrá el usuario pero que ayudaba a validar los distintos bloques hardware y software por separado. Desde esta, se probó a hacer envíos tanto únicos (“*Manual Send Mode*”) como periódicos cada segundo (“*Auto Send Mode*”) desde el PC a la FPGA a través del control señalado, verificando que a esta le llegaba, mostrando el dato en binario encendiendo los leds de la BASYS 3 (**LED0 a LED7**), además de mostrarlo en el *display*.

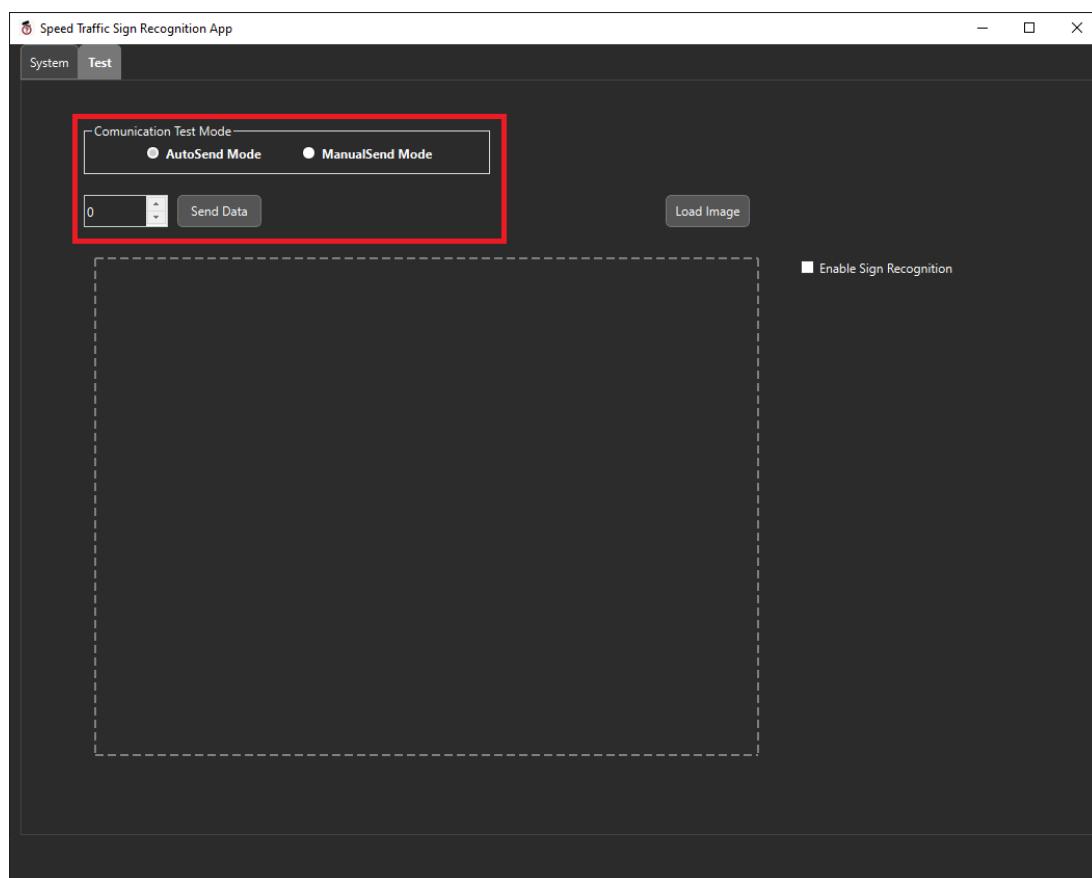


Figura 4.17 Vista interfaz de usuario DEBUG comunicación con FPGA

Posteriormente, se comprobó la recepción desde la FPGA, implementando un sencillo contador free-running en la misma y viendo por la consola de Visual Studio Code, como llegaba a la aplicación.

#### 4.2.2 Modelo neuronal con imágenes estáticas

Una vez validada la comunicación, se procede a comprobar el correcto funcionamiento del modelo neuronal, cargando una imagen del *dataset* de prueba desde el sistema de archivos del PC, pulsando en el botón “*Load Image*”. Se observó que los resultados eran correctos y con gran valor de confianza, tal y como indicaban que pasaría los datos sobre el rendimiento tras el proceso de entrenamiento.

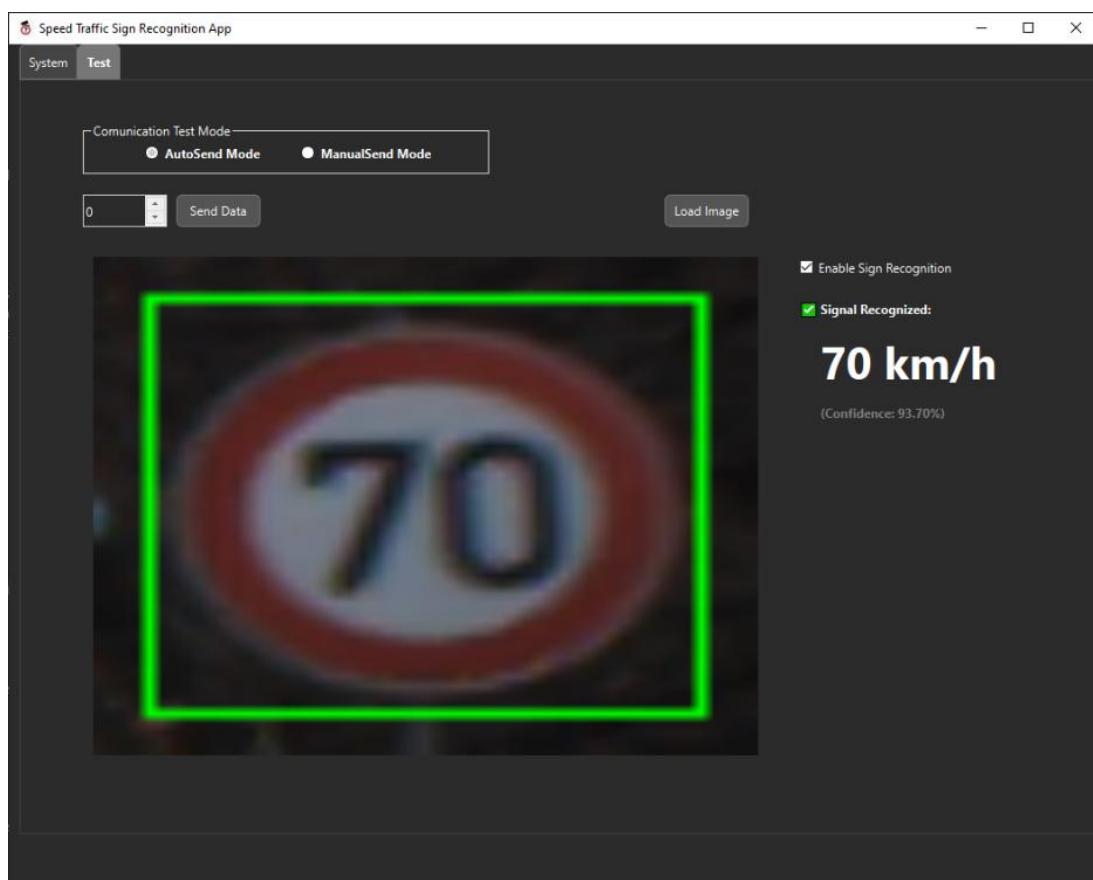


Figura 4.18 Vista interfaz de usuario DEBUG tras reconocimiento de señal

#### 4.2.3 Recepción de imágenes

##### SINTÉTICAS

Tras esto, se prueba a enviar imágenes desde la FPGA, de forma sintética mediante el emulador del sensor, activando el correspondiente switch de la placa de desarrollo (**SW0**). En primer lugar, se envían en formato a color (**TOP\_12MHz\_COLOR.bin**), comprobándose que la imagen con el patrón definido por hardware llega y se visualiza correctamente en la interfaz.

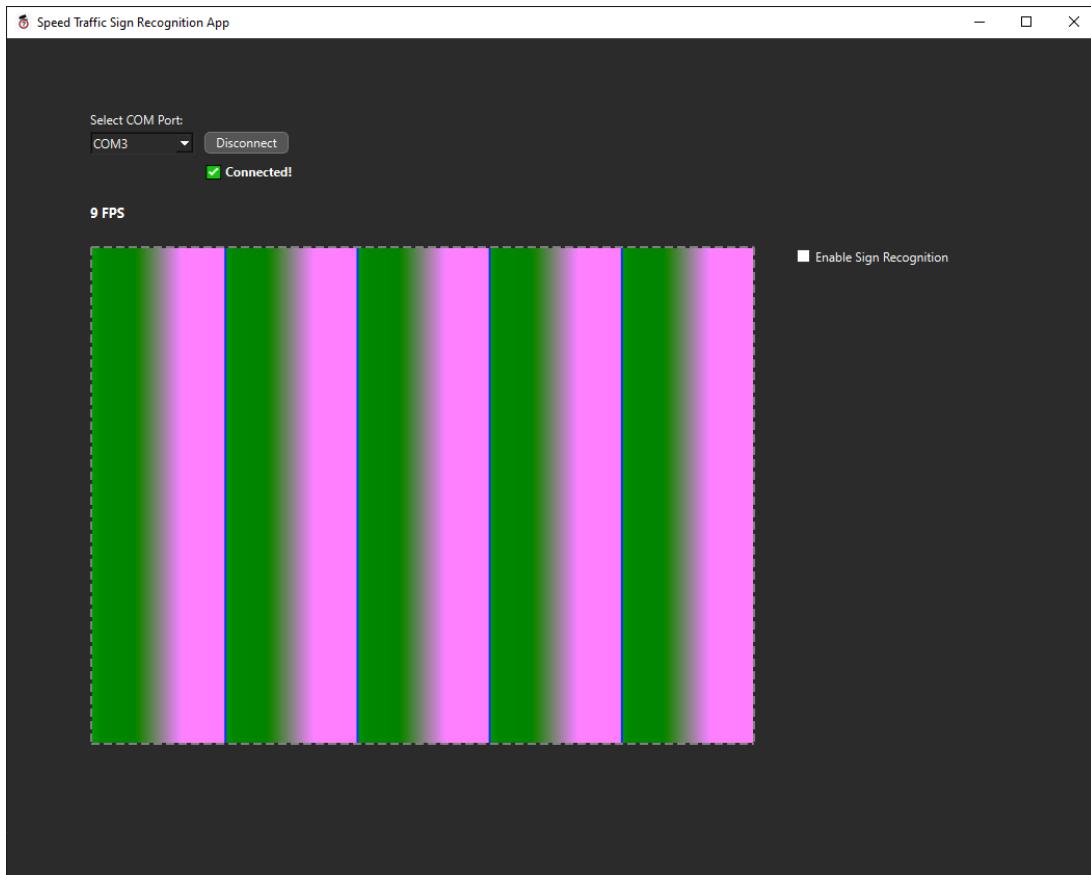


Figura 4.19 Envío imágenes sintéticas a color al PC

Se prueba lo mismo en escala de grises y a mayor frecuencia (**TOP\_25MHz\_GRAY.bin**), obteniendo un valor mayor de FPS:

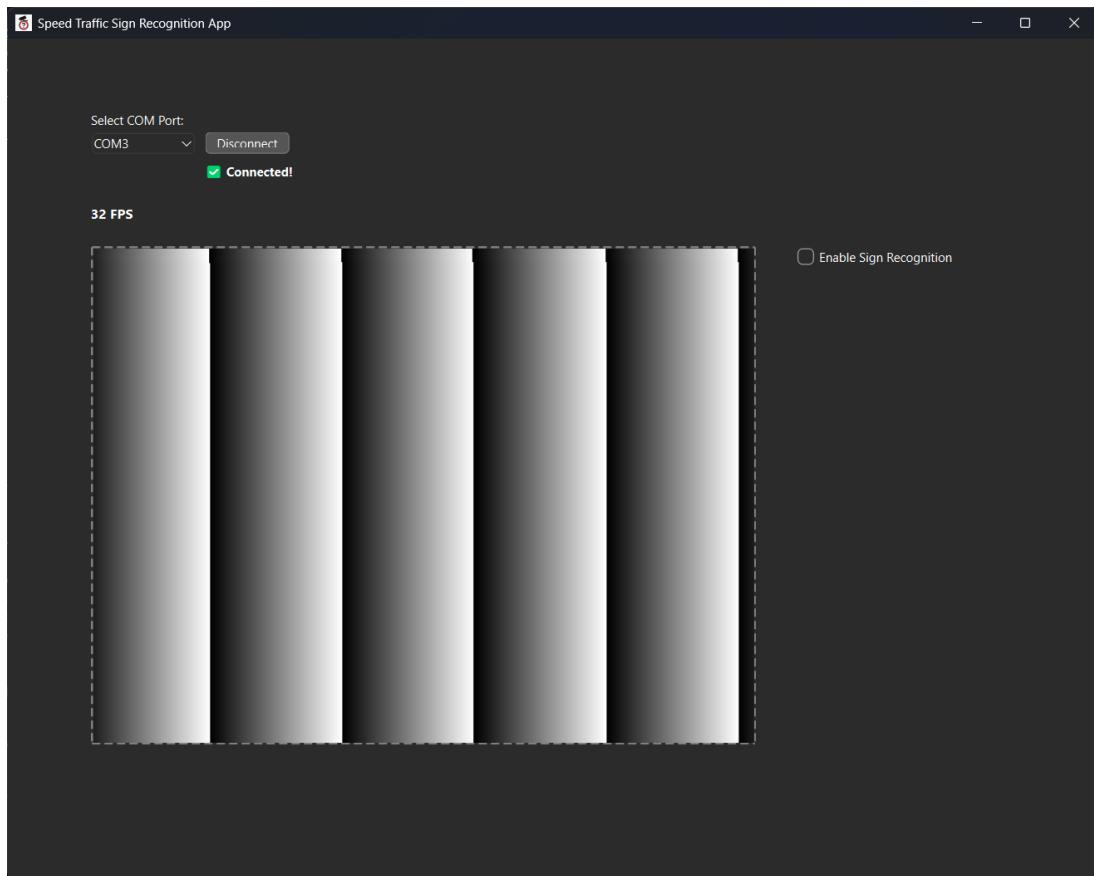


Figura 4.20 Envío imágenes sintéticas en escala de grises al PC

## REALES

Finalmente, se prueba a conectar el sensor real y deshabilitar el emulado, para ver el sistema real en funcionamiento. Se deshabilita el procesamiento, para comprobar solo la comunicación. Se puede ver como las imágenes llegan correctamente, con buena calidad y con una tasa aceptable (12FPS).

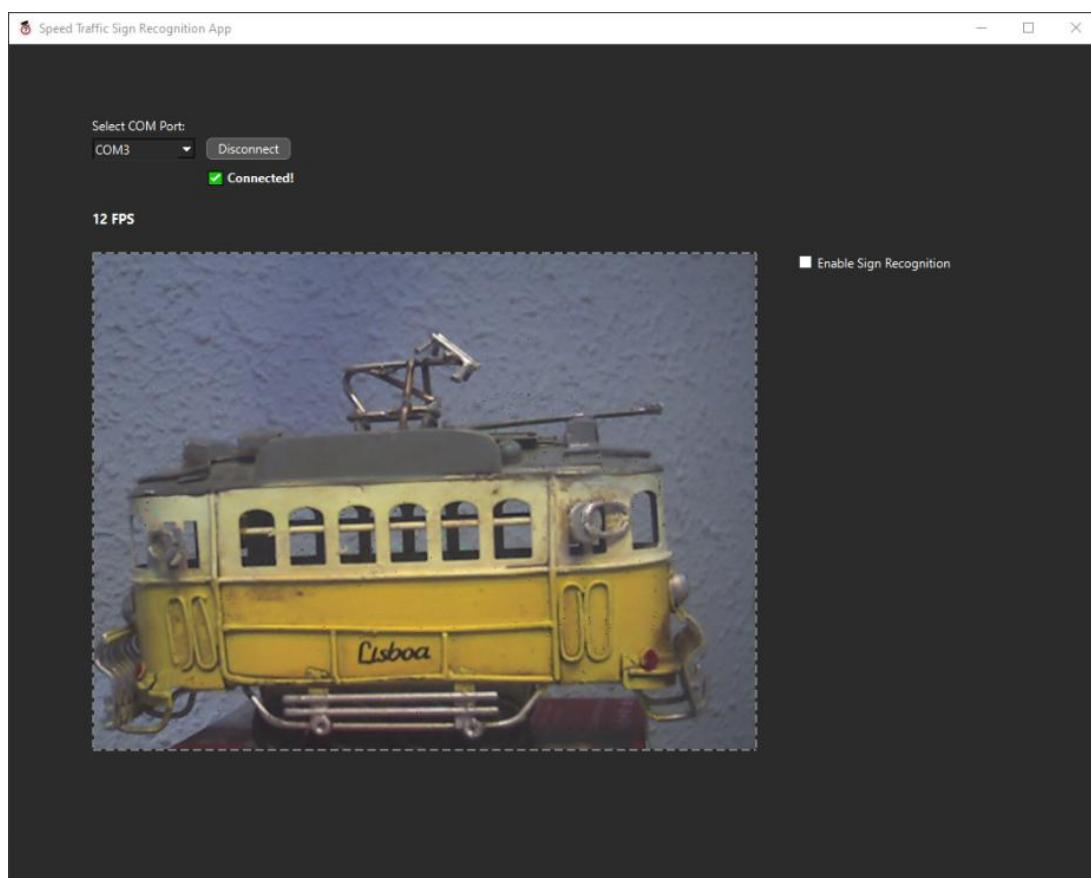


Figura 4.21 Envío de imágenes reales a color al PC

Por hacer la misma prueba que se simuló en la FPGA, se prueba a configurarla a mayor frecuencia y escala de grises, corroborando así, que el número de imágenes recibidas es mayor tal y como se comentó en la parte hardware que ocurría.

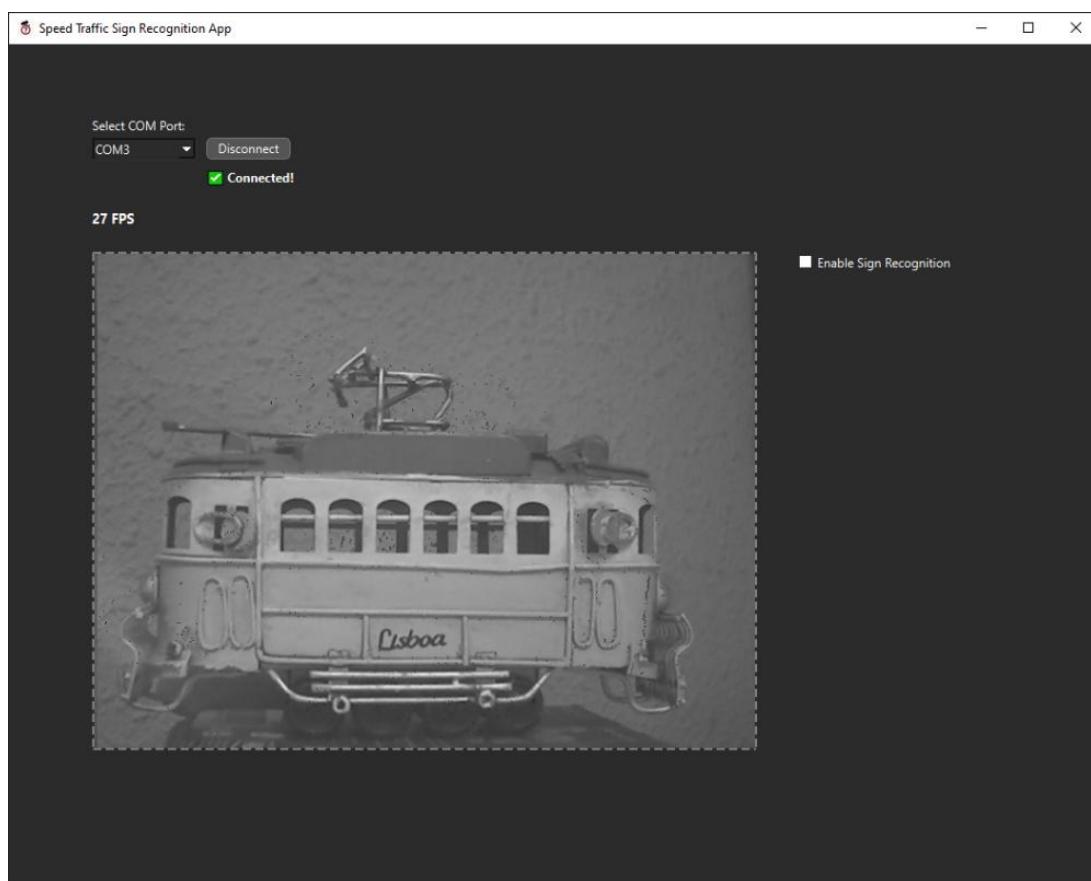


Figura 4.22 Envío de imágenes reales en escala de grises al PC

Ambos *bitstreams* usados, para formato a color y escala de grises, están disponibles en el repositorio del proyecto en GitHub, en la carpeta [/bitstreams](#).

#### 4.2.4 Modelo neuronal con imágenes reales

Para finalizar, se habilita la detección y reconocimiento de señales de tráfico de límites de velocidad, para ver el rendimiento en tiempo real del proceso. Se ve como los FPS disminuyen, pero aun así siguen siendo aceptables y dan sensación de “tiempo real”. Se expone al sensor a distintas señales, viendo como el mismo es capaz de acertar la velocidad indicada (30km/h) y, además, con alto valor de confianza (91.01%).

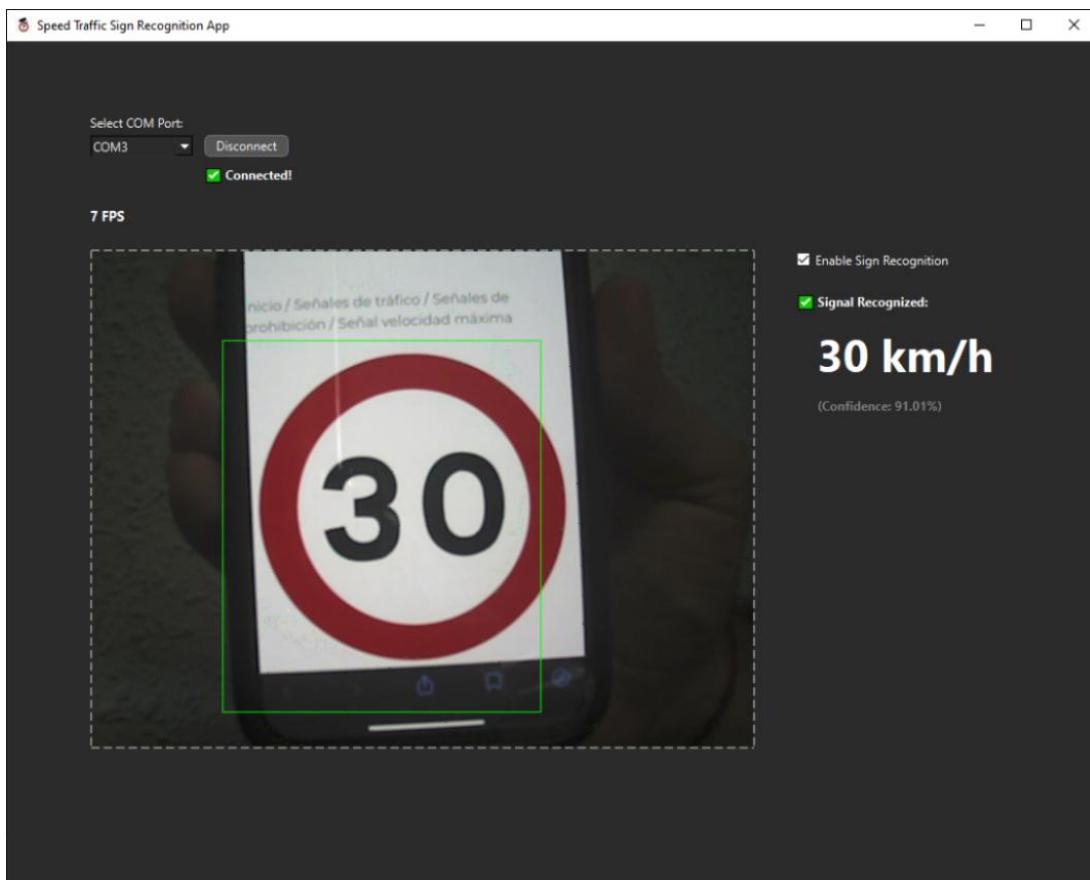


Figura 4.23 Detección y reconocimiento de señal de tráfico en tiempo real



Sistema de reconocimiento de señales de tráfico

# Capítulo 5 Conclusiones y líneas futuras

El presente proyecto ha abordado de forma completa el diseño, implementación y validación de un sistema de reconocimiento de señales de tráfico relacionadas con los límites de velocidad. Todo ello basado en hardware reconfigurable y procesamiento software mediante redes neuronales. Desde la adquisición de imágenes hasta la detección de señales, se ha logrado una solución funcional, eficiente y escalable que integra múltiples tecnologías y cumple con todos los requisitos que se presentaron en el primer capítulo del presente documento.

A nivel hardware, se ha desarrollado una arquitectura modular basada en bloques diseñados en VHDL, que incluyen interfaces de comunicación síncrona y asíncrona, sistemas de control, FIFO, visualización mediante *display* y un completo sistema de adquisición de imagen. Esta arquitectura ha sido verificada mediante simulaciones funcionales y post-síntesis, y desplegada con éxito sobre la FPGA en la placa BASYS 3.

A nivel software, se ha desarrollado una aplicación en Python usando PySide6 (Qt) que actúa como interfaz de usuario, controlador del sistema y motor de inferencia, ejecutando un modelo neuronal entrenado (YOLOv8n). La FPGA se comunica con el PC a través de una interfaz FIFO expuesta por el módulo FTDI (UM232H-B), que abstractea la complejidad del protocolo USB.

Sin embargo, aunque se han alcanzado con éxito los objetivos planteados, el proyecto ha evidenciado ciertos desafíos en cuanto a latencias, dependencia del software para el procesamiento neuronal, y el esfuerzo de integración entre entornos muy distintos (sistemas digitales y *deep learning*). Esta reflexión da lugar a propuestas de mejora.

Una de las mejoras más relevantes que se plantea es la posibilidad de **integrar el modelo neuronal directamente en hardware**, eliminando la dependencia del software para el reconocimiento de señales. Para ello, hubiera sido ideal disponer de una plataforma FPGA con SoC, como la familia **Zynq** de Xilinx, que combina lógica reconfigurable con procesadores ARM y permite la inclusión de **NPUs (Neural Processing Units)** embebidas.

Este enfoque permitiría comparar el rendimiento del procesamiento software con el hardware, evaluar métricas como latencia y consumo energético, y, además, conseguir una solución **completamente embebida**, donde el software actuaría únicamente como interfaz de visualización y control, delegando todo el procesamiento intensivo a la FPGA.

Además, futuras líneas de trabajo podrían contemplar:

- **Ampliación del dataset** para incluir otros tipos de señales.
- Mejora del pipeline de adquisición y preprocessado para soportar formatos en color y resoluciones mayores, e incluso más interesante aún, adaptar la imagen capturada a las necesidades del procesado neuronal. Actualmente, el sistema se

ha desarrollado y probado con una resolución estándar de 640x480 píxeles proporcionada por el sensor **MT9V111**, cuyo interfaz se ha emulado en este proyecto. Futuras mejoras podrían contemplar también el uso de sensores más avanzados y rápidos, así como aprovechar completamente la información en color capturada.

- Adaptación del sistema a plataformas móviles o *edge devices*.
- Exportación del modelo neuronal a formatos optimizados como **ONNX** o **TensorRT** para integrarlo en sistemas más ligeros.

# Capítulo 6 Bibliografía

- [1] S. D. R. G., A. F. Joseph Redmon, "University of Washington, Allen Institute for AI," [Online]. Available: <https://homes.cs.washington.edu/~ali/papers/YOLO.pdf>.
- [2] S. M. S. A. E.-M. AHMAD SHAWAHNA, "FPGA-based Accelerators of Deep Learning Networks for Learning and Classification: A Review (IEEE)," [Online]. Available: <https://arxiv.org/pdf/1901.00121.pdf>.
- [3] FTDI Chip, "FTDI Chip - Datasheet FT232H," [Online]. Available: [https://ftdichip.com/wp-content/uploads/2020/07/DS\\_FT232H.pdf](https://ftdichip.com/wp-content/uploads/2020/07/DS_FT232H.pdf).
- [4] Digilent, "Digilent - BASYS 3 Reference Manual," [Online]. Available: [https://digilent.com/reference/programmable-logic/basys-3/reference-manual?srsltid=AfmBOoqQUf8n2IA6S\\_ckt8ibSktEfR57\\_iDHyd1HMaCCRwaq6KqBo5\\_C](https://digilent.com/reference/programmable-logic/basys-3/reference-manual?srsltid=AfmBOoqQUf8n2IA6S_ckt8ibSktEfR57_iDHyd1HMaCCRwaq6KqBo5_C).
- [5] FTDI Chip, "FTDI Chip - Datasheet UM232H-B Module," [Online]. Available: [https://ftdichip.com/wp-content/uploads/2020/07/DS\\_UM232H-B.pdf](https://ftdichip.com/wp-content/uploads/2020/07/DS_UM232H-B.pdf).
- [6] ON Semiconductor, "Datasheet - Soc VGA CMOS Image Sensor MT9V111," [Online]. Available: <https://www.alldatasheet.com/datasheet-pdf/pdf/759783/ONSEMI/MT9V111.html>.
- [7] Digilent, "BASYS 3 Programming Guide," [Online]. Available: [https://digilent.com/reference/learn/programmable-logic/tutorials/basys-3-programming-guide/start?srsltid=AfmBOort6FIKb4Zw07d4OlHaT7oH\\_me-lmq13OJGliqmkd9W9VlnDCiy](https://digilent.com/reference/learn/programmable-logic/tutorials/basys-3-programming-guide/start?srsltid=AfmBOort6FIKb4Zw07d4OlHaT7oH_me-lmq13OJGliqmkd9W9VlnDCiy).
- [8] Conda, "Documentation - Conda Environment," [Online]. Available: <https://docs.conda.io/en/latest/>.
- [9] Qt, "Documentation - Qt for Python," [Online]. Available: <https://doc.qt.io/qtforpython-6/>.
- [10] FTDI Chip, "D2XX Drivers - DLL FTD2XX," [Online]. Available: <https://ftdichip.com/drivers/d2xx-drivers/>.
- [11] Ultralytics, "Documentation - Ultralytics YOLO," [Online]. Available: <https://docs.ultralytics.com/es/#where-to-start>.

- [12] Institut Für Neuroinformatik, "GTSRB Traffic Signals Dataset," [Online]. Available: [https://benchmark.ini.rub.de/gtsrb\\_dataset.html](https://benchmark.ini.rub.de/gtsrb_dataset.html).
- [13] TensorFlow, "Documentation - TensorBoard," [Online]. Available: <https://www.tensorflow.org/tensorboard?hl=es-419>.
- [14] Netron, "NetronApp - Neural Network Visualizer," [Online]. Available: <https://netron.app/>.
- [15] Roboflow, "custom\_dataset," [Online]. Available: <https://universe.roboflow.com/selfdriving-car-qtywx/self-driving-cars-lfjou/dataset/6>.
- [16] Digilent, "Vivado Instalation Guide," [Online]. Available: <https://digilent.com/reference/programmable-logic/guides/installing-vivado-and-sdk>.
- [17] FTDI Chip, "FT\_PROG - EEPROM Programming Utility," [Online]. Available: <https://ftdichip.com/utilities/>.

# Anexo A. Repositorio en GitHub

Todo el código fuente del sistema, incluyendo los módulos hardware en VHDL, los *scripts* de simulación, los entrenamientos de la red neuronal y la aplicación de escritorio, se encuentra disponible en el repositorio oficial del proyecto: <https://github.com/ManuelSN/Speed Traffic Sign Recognition System>

Además, en dicho repositorio se incluye el instalador ***STSRApplInstaller.exe***, que permite ejecutar directamente la aplicación sin necesidad de instalar Python ni las dependencias manualmente, pensado para **usuarios finales**. El proceso de instalación se describe en el Anexo B. Instalación del software.

## Anexo B. Instalación del software

A continuación, se muestra el proceso de instalación del software del proyecto. Para ello, será necesario descargar y ejecutar el archivo *STSRApplInstaller.exe* para Windows.

Este se ha generado mediante la utilidad “Inno Setup Compiler”, de forma que es un paquete autocontenido, con todas las dependencias ya instaladas. Una vez hecho esto, nos preguntará si queremos permitir a esta aplicación hacer cambios en el dispositivo, tras confirmar, aparecerán las siguientes ventanas del instalador:

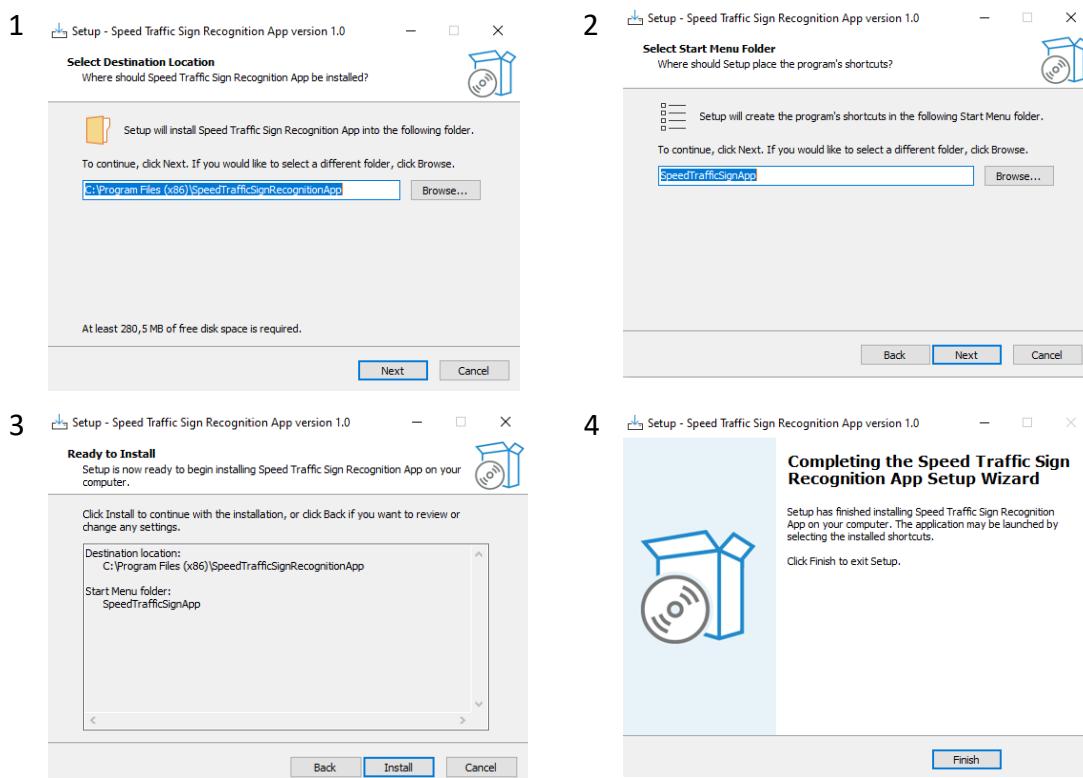
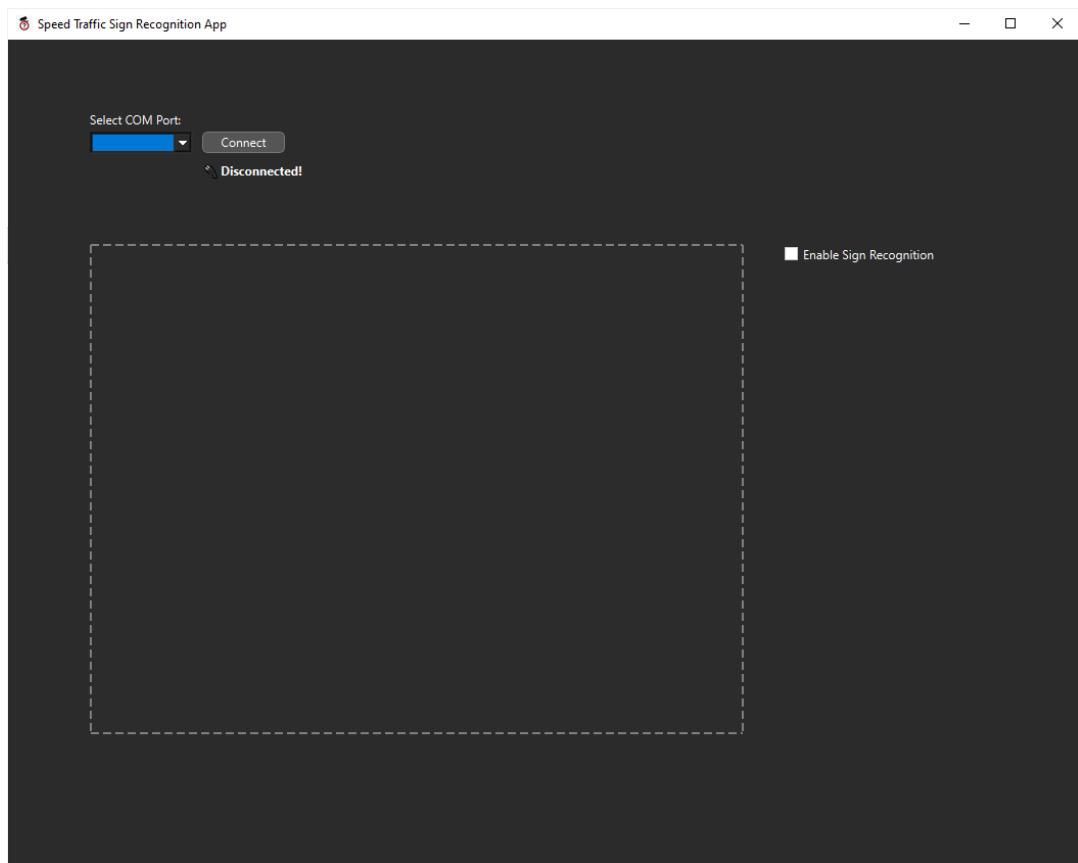


Figura 0.1 Proceso de instalación del software

En la primera ventana, se selecciona dónde se quiere instalar la aplicación. Tras pulsar en “Next”, se pasa a la segunda ventana donde permite indicar donde crear el acceso directo a la misma. Tras esto, en la tercera ventana se hace *click* en instalar, comenzando así el proceso de instalación que durará unos segundos.

Tras finalizar, se cierra la cuarta y última ventana, y ya se puede buscar en la barra de Windows y abrir “**Speed Traffic Sign Recognition App**”:



*Figura 0.2 Vista inicial de la aplicación*