UNIVERSIDAD DE MÁLAGA

E.T.S. DE INGENIERÍA DE TELECOMUNICACIÓN
UNIVERSIDAD DE MÁLAGA

MASTER'S THESIS

Traffic Sign Recognition System

MASTER IN ELECTRONIC SYSTEMS FOR SMART ENVIRONMENTS

Department of Electronic Technology

School of Telecommunication Engineering

University of Malaga

Manuel Sánchez Natera,
Malaga, 2025

E.T.S. DE INGENIERÍA DE TELECOMUNICACIÓN, UNIVERSIDAD DE MÁLAGA

Traffic Sign Recognition System

Author: Manuel Sánchez Natera

Supervisor: Martín González García

Department: Electronic Technology

Degree: Master Degree 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).

E.T.S. DE INGENIERÍA DE TELECOMUNICACIÓN, UNIVERSIDAD DE MÁLAGA

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).

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.

Table of contents

List of figures

List of tables

List of acronyms

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*

Chapter 1 Requirements and Use Cases

1.1 Introduction

The objective of this chapter is to extract, analyze, and define the needs of the traffic sign recognition system. The final user requirements will be detailed, as well as how the project has been designed to meet them.

1.2 State of the Art

In the last decade, Advanced Driver-Assistance Systems (ADAS) have progressively incorporated traffic sign detection and recognition as a key functionality to increase road safety. Manufacturers such as Tesla, BMW, Mercedes, or Toyota, among others, integrate front-facing cameras that, in combination with computer vision algorithms, allow the driver to be warned of speed limits or even automatically adjust the adaptive cruise control.

Technological evolution has led to the fact that, nowadays, deep learning models for real-time detection such as YOLO (You Only Look Once), in its different versions (YOLOv3, YOLOv5, YOLOv8), have become an important standard for object detection tasks in driving environments. These models offer a very suitable balance between accuracy and inference speed, enabling their integration into low-power embedded systems [1].

However, practical implementation in commercial vehicles is usually carried out on specialized SoCs (e.g., NVIDIA Jetson) or general-purpose processors, which limits energy efficiency in scenarios requiring continuous real-time processing. In this regard, Field-Programmable Gate Arrays (FPGAs) represent a competitive alternative by allowing parallelization of operations, reducing latency, and adapting hardware architecture to the specific needs of the system [2]. The combination of FPGA for image acquisition and preprocessing together with a Personal Computer (PC) for neural model inference constitutes a hybrid approach that leverages the advantages of both worlds.

The motivation of this project is precisely centered on this hybrid approach. On the one hand, the FPGA enables capturing and transmitting images in real time from a digital sensor, ensuring a continuous and efficient data flow. On the other, a pre-trained and fine-tuned YOLOv8 model processes the images on a PC quickly and reliably. This makes the approach especially suitable for applications in driving simulators, academic environments, and as an intermediate step for future implementations in real ADAS systems, where robustness and latency requirements are critical.

1.3 Objectives

The main goal of this project is the design of a system that, based on a set of hardware components and a desktop software application, is capable of displaying real-time images while detecting and recognising traffic signs corresponding to speed limits.

1.4 Project Requirements

1.4.1 Requirements Diagram

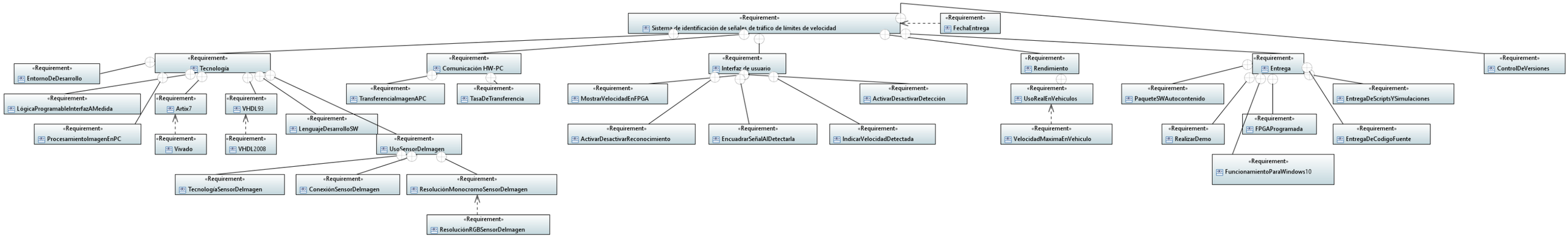


Figure 1.1 Requirement diagram

The system must comply with the following main requirements:

- Capture images at least in VGA format (Video Graphics Array), i.e., 680×480.
- Establish communication between the hardware and the PC.
- Achieve a frame rate higher than 10FPS (Frames Per Second).
- Allow pausing and resuming image reception on demand from the PC interface.
- Detect and recognize traffic signs related to speed limits appearing in the received image.
- Be self-contained: the software must be portable and compatible with Windows 10 or later. Likewise, the hardware must be delivered fully functional to the user, without requiring manual intervention.

Below are other derived requirements, which are divided into non-functional and functional requirements.

1.4.2 Non-Functional Requirements

The following non-functional requirements are defined:

ID	Name	Priority	Precedence
R1.1	CustomInterfaceWithProgrammableLogic	Essential	-
R1.2	Artix-7	Essential	R1.1
R1.3	Vivado	Essential	R1.1, R1.2
R1.4	VHDL93	Essential	R1.1
R1.5	VHDL2008	Optional	R1.1, R1.4
R2.1	ImageProcessingOnPC	Essential	-
R2.2	SWDevelopmentLanguage	Desirable	R2.1
R2.3	Windows10Compatibility	Essential	R2.1
R2.4	DevelopmentEnvironment	Optional	R2.1
R8	Self-ContainedSWPackage	Essential	-
R9	Pre-programmedFPGA	Essential	-
R10	DeliveryOfSimulationScripts	Essential	-
R11	DeliveryOfSourceCode	Essential	-
R12	SystemDemo	Essential	-
R13	VersionControl	Essential	-
R14	DeliveryDate	Essential	-

Table 1.1 Non-functional requirements of the project

R1.1 CustomInterfaceWithProgrammableLogic

Use programmable logic to implement a tailored interface.

R1.2 Artix-7

Low-cost technology using the BASYS 3 development board with Xilinx Artix-7 FPGA.

R1.3 Vivado

FPGA programming and simulation environment.

R1.4 VHDL93

Minimum version of the hardware description language used.

R1.5 VHDL2008

Optional, may be used for extended features.

R2.1 ImageProcessingOnPC

All image processing will be implemented on the PC.

R2.2 SWDevelopmentLanguage

Preferably C/C++ for the user interface.

R2.3 Windows10Compatibility

Guarantee software functionality on Windows 10.

R2.4 DevelopmentEnvironment

Use Qt framework with OpenCV and FTDI libraries FTDI (Future Technology Devices International).

R8 Self-ContainedSWPackage

The software must be delivered as a standalone installer.

R9 Pre-programmedFPGA

The FPGA must be shipped already configured with bitstream.

R10 DeliveryOfSimulationScripts

Provide Vivado-compatible simulation scripts.

R11 DeliveryOfSourceCode

Provide both VHDL and PC application source code.

R12 SystemDemo

Demonstration of the system's functionality.

R13 VersionControl

Create a repository for hardware and software.

R14 DeliveryDate

The project delivery date will be in September 2025.

1.4.3 Functional requirements

The functional requirements of the project are listed below.

ID	Name	Priority	Precedence
R3.1	UseOfImageSensor	Essential	-
R3.2	ImageSensorTechnology	Essential	R3.1
R3.3	ImageSensorConnection	Essential	R3.1
R3.4	MonochromeResolution	Essential	R3.1
R3.5	RGBResolution	Optional	R3.1
R4.1	ImageTransferToPC	Essential	-
R4.2	TransferRate	Essential	R4.1
R5	DisplaySpeedOnFPGA	Essential	-
R6.1	EnableDisableRecognition	Essential	-
R6.2	EnableDisableDetection	Essential	-
R6.3	FrameTheSignWhenDetected	Essential	-
R6.4	IndicateDetectedSpeed	Essential	-
R7.1	RealVehicleUse	Optional	-
R7.2	MaximumVehicleSpeed	Optional	R7.1

Table 1.2 Functional requirements of the project

R3.1 UseOfImageSensor

Necessary for capturing traffic sign images.

R3.2 ImageSensorTechnology

Use of video image sensor with CMOS (Complementary Metal-Oxide-Semiconductor) technology.

R3.3 ImageSensorConnection

The system must allow connection to an image sensor via the FPGA.

R3.4 MonochromeResolution

The sensor must have greyscale resolution (up to 30FPS transfer rate with this format).

R3.5 RGBResolution

RGB format (up to 30FPS). May be necessary to detect areas of colour.

R4.1 ImageTransferToPC

USB transmission of images captured by the sensor from the FPGA to the PC.

R4.2 TransferRate

A minimum transfer rate of 10 images per second (FPS) must be guaranteed. This is to give the user the impression of real-time video.

R5 DisplaySpeedOnFPGA

Display the recognised speed at all times on a display connected to the FPGA.

R6.1 EnableDisableRecognition

From the user interface on the PC, you can enable and disable the speed recognition displayed on the detected signal.

R6.2 EnableDisableDetection

From the user interface on the PC, you can enable and disable speed signal detection.

R6.3 FrameTheSignWhenDetected

A rectangle should be drawn around the detected signal in the PC user interface.

R6.4 IndicateDetectedSpeed

The recognised speed in kilometers per hour (km/h) of the detected traffic sign must be indicated.

R7.1 RealVehicleUse

Use of this system in a real moving vehicle.

R7.2 MaximumVehicleSpeed

Indicate the maximum speed at which the vehicle can travel so that the system can function correctly, without missing any signals that need to be detected and recognised.

1.5 Use Cases

The external actors involved are shown in the following table:

Name	Description
User	Person who interacts with the system.
System	Technology that enables the detection and recognition of traffic signs related to speed limits.

Table 1.3 External actors

The functional actions that the user initiates or expects from the system are shown in the following use case diagram.

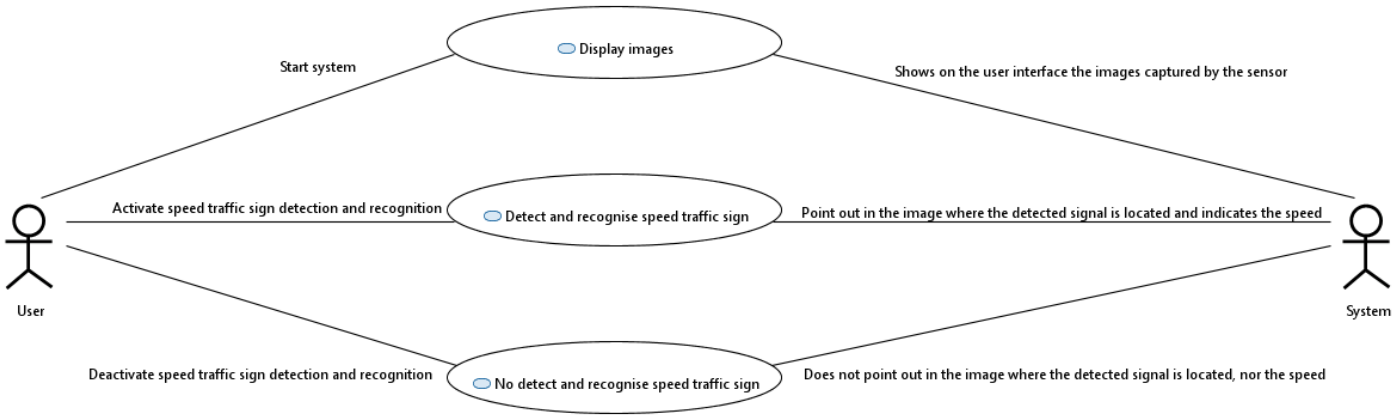


Figure 1.2 Use case diagram.

Below is a textual description of the use cases (Figure 1.2):

C1. *Display images*

By default, images captured by the camera will be displayed at a transmission speed sufficient to give the user a real-time feel.

- **Main actor:** System.
- **Participants and objectives:** Both actors participate in this use case. The user starts the system by launching the application and correctly connecting the device to the PC. The system captures the images through the sensor, sends them to the PC, and displays them on the user interface.
- **Preconditions:** *"SpeedTrafficSignRecognitionApp"* installed on the PC, FPGA development board powered up, connected to the PC via UM232H-B device, configured with the bitstream, and image sensor correctly connected.
- **Minimum guarantees:** The images are displayed at a minimum of 10FPS, to give the user the impression that it is a real-time video.

- **Main success scenario:** The PC requests images from the FPGA, which initiates the process of capturing and sending data. As the desktop application receives them, they are processed only if the detection and recognition option is enabled.
- **Secondary scenarios:** No images are sent to the PC, or they arrive corrupted and do not display correctly.

C2. *Detect and recognise speed traffic sign*

The user enables speed limit sign detection and recognition from the interface. Once this is done, the application displays the image with the detected sign framed and, next to it, a number indicating the speed limit in km/h.

- **Main actor:** User.
- **Participants and objectives:** Both actors participate in this use case. The user activates the detection of speed limit signs. The system displays the captured images on the interface and indicates to the user where the detected sign is located by drawing a box around it and displaying the numerical value of the speed limit.
- **Preconditions:** Image transmission to the PC works correctly.
- **Minimum guarantees:** With a rate of at least 10FPS, the system is capable of detecting all traffic signs related to speed limits.
- **Main success scenario:** The system correctly frames the detected signals and accurately displays the numerical value (km/h) corresponding to the speed limit indicated on the traffic sign.
- **Secondary scenarios:** Although the user has enabled this option, the system is unable to detect or recognise the signals.

C3. *No detect and recognise speed traffic sign*

The user disables the detection and recognition of speed traffic signs from the interface. Once this is done, the application displays the original image, without any box or drawing around it, nor any information about the detected traffic sign.

- **Main actor:** User.
- **Participants and objectives:** Both actors participate in this use case. The user disables speed traffic sign detection. The system displays the captured images on the interface without indicating to the user where the detected sign is located, nor does it indicate the speed.
- **Preconditions:** Image transmission to the PC and signal detection and recognition are functioning correctly.
- **Minimum guarantees:** The system responds immediately to the user's request and stops detecting and recognising traffic signs in the captured images.

- **Main success scenario:** The system displays the original images captured by the sensor, without any drawings or figures locating the traffic signals detected on them, or any other information.
- **Secondary scenarios:** Even though the user has disabled this option, the system is unable to stop detecting and recognising signals.

Chapter 2 System Description

The main objective of the developed system is the automatic detection and recognition of traffic signs related to speed limits. This requires real-time image processing, suitable for integration into vehicular environments or driving simulators, where continuous and efficient visual monitoring of the environment is necessary.

The system consists of two main modules:

- **Hardware Module**: responsible for acquiring images using a digital sensor and subsequently transmitting them to the software module.
- **Software Module**: responsible for receiving images, processing them using a neural model, and presenting the results through a Graphical User Interface (GUI).

The user can control the operation of the system from within the application itself: establish the connection with the hardware, view the captured images, and activate or deactivate signal detection and recognition.

Thanks to its modular architecture and portability, this system is suitable for academic or testing purposes, as well as for future integration into Advanced Driver Assistance Systems (ADAS).

2.1 General Description

The proposed system is based on a modular architecture consisting of hardware and software components, which are integrated via the PC's USB (Universal Serial Bus) port, as shown in the Figure 2.1.

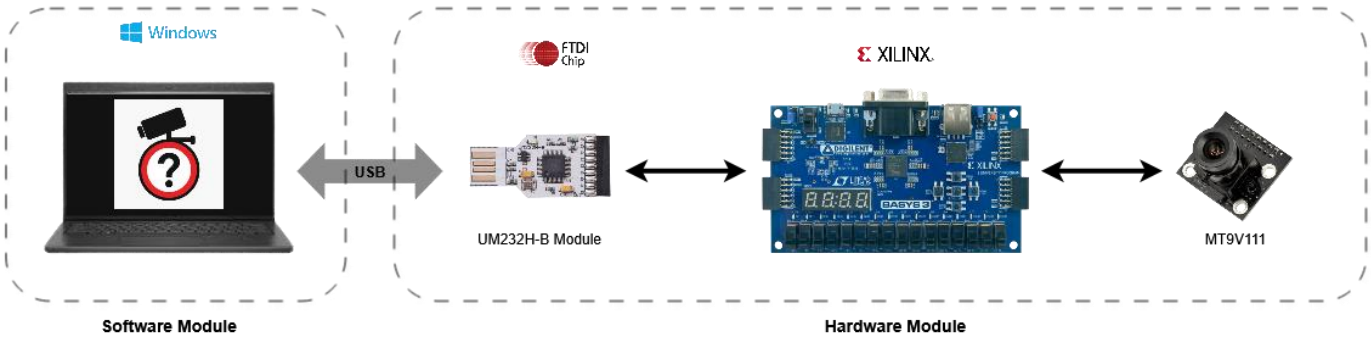


Figure 2.1 General system architecture

The system architecture consists of the following elements:

- Computer with Windows operating system.
- Desktop application called *"SpeedTrafficSignRecognitionApp"*, compatible with Windows operating system.
- UM232H-B module from manufacturer FTDI for USB communication.
- BASYS 3 development board with Xilinx Artix-7 FPGA for image acquisition.
- MT9V111 SoC (System on Chip) with digital output in YUV 4:2:2 format.

Note: Specific details of each of these components are described in Chapter 3 Design and Implementation.

2.2 Software Description

The software module is developed in **Python 3.11.11**, using libraries such as **PySide6** for the graphical interface and **Ultralytics YOLO** for traffic sign recognition using a pre-trained **neural model**.

The application uses ***multithread*** (**PySide6 QThread**) to keep the graphical interface responsive while performing image acquisition and processing tasks.

The graphical interface allows:

- Establish connection with the FPGA.
- Display captured images in real time.
- Enable or disable signal detection and recognition.
- Display processing results, including the detected class (speed limit) and the associated confidence level.

2.3 Hardware Description

The hardware module is implemented on the **Artix-7 FPGA** on the BASYS 3 board. Its main function is to acquire images from the MT9V111 sensor and send them to the PC via a USB interface controlled by the **FT232** chip.

2.3.1 UM232H-B Module

The FT232H chip is responsible for adapting the FPGA's parallel communication to the USB protocol. Although it supports multiple modes, it has been configured in **FT245 asynchronous FIFO** mode using the "***FTProg***" utility to achieve a simple and efficient interface.

This mode uses signals such as **`RD#`**, **`WR#`**, **`RXF#`** and **`TXE#`** to synchronise data transmission between the FPGA and the PC. Although the chip supports speeds of **up to 480 Mbps** (USB 2.0), in **FT245 asynchronous FIFO** mode the maximum transfer rate is **8 MBps**, which is more than sufficient for the system's transmission needs [3].

2.3.2 MT9V111 Sensor

The **MT9V111** image sensor is a CMOS SoC that provides images in VGA resolution (640×480) with a digital data stream. This sensor is an essential part of the system and connects directly to the FPGA, providing synchronisation signals (**`PIXCLK`**, **`LINE_VALID`**, **`FRAME_VALID`**) and data in **YUV 4:2:2 format**.

Chapter 3 Design and Implementation

This chapter details both the hardware and software design and implementation process.

3.1 Hardware Module

3.1.1 Introduction

The hardware design, as seen in the system architecture, is based on the BASYS 3 development board, manufactured by Digilent. This integrates multiple components, the main one being the Xilinx Artix-7 FPGA, complemented by external QSPI (Quad Serial Peripheral Interface) flash memory, standard interfaces such as USB and VGA, mechanical buttons, switches, LEDs and PMOD expansion pins Figure 3.1.

The BASYS 3 is powered via the USB connector J4, which is also used to configure the FPGA.

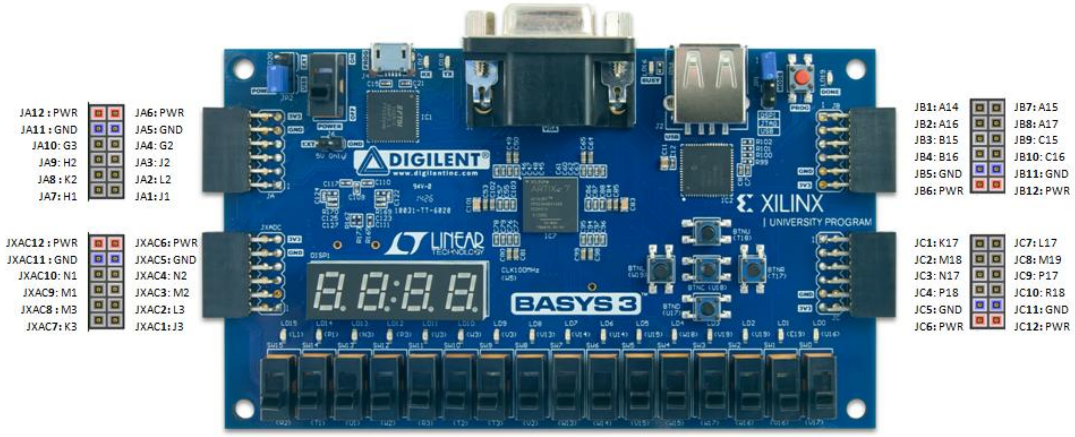


Figure 3.1 PMOD pins BASYS 3 development board

3.1.2 BASYS 3 Development Board

Oscillator

Every digital design needs a clock signal. For this design, the BASYS 3 has a **100MHz** oscillator that is the **system clock input**. It is used to synchronise the different logic blocks of the hardware described in **VHDL**.

This input is connected internally to **pin W5** of the Artix-7 FPGA, as indicated in the constraint file (****.xdc***), where it is named so that it can be referenced in the design (Master Clock, **`MCLK`**). From this, derived clocks can be generated using dividers or PLL blocks if the design requires it.

7-segment display

The BASYS 3 board incorporates a 4-digit common anode display, where each digit contains 7 LEDs that can be activated individually, giving rise to many possible combinations or patterns (Figure 3.2). In this case, each 7 segments will be used to represent numbers from 0 to 9. In addition to these, the display has an eighth segment called **DP** that represents the decimal point, although it is not used in this project [4].

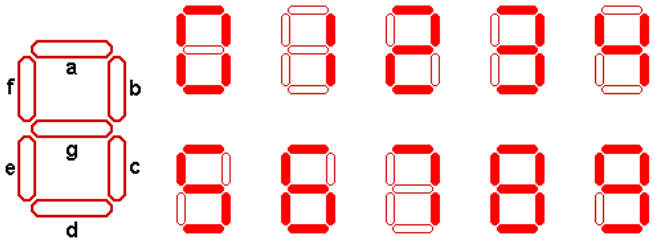


Figure 3.2 Segments of 7-segment display

The anodes of each digit are controlled by the **ANx** signals, and the common cathodes of the segments are connected from **CA** to **CG** (Figure 3.3). This allows for a multiplexed configuration, where the segments are shared by all digits, but only one is active at any given time.

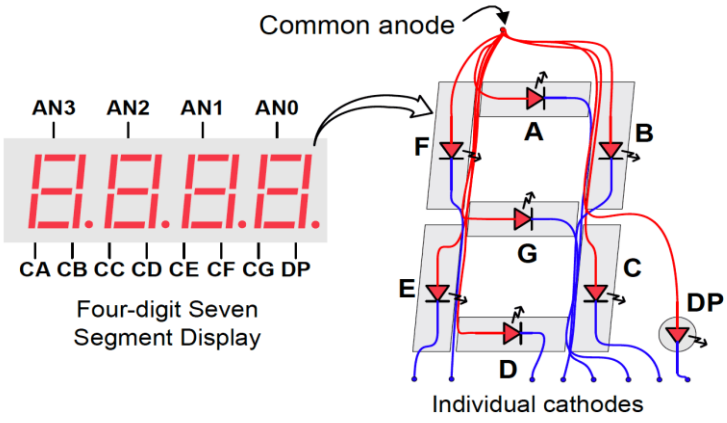


Figure 3.3 Connections to display

To make all digits appear to be lit simultaneously, it is recommended to activate each one and turn off the rest with a high refresh rate (> 60Hz). If the refresh rate drops below 45Hz, the segments may flicker when switching digits.

The activation signals are inverted, so that both the anodes (**ANx**) and cathodes (**CA to CG**) are activated at a low level (**'0'**).

PMOD Connectors

These connectors allow you to connect peripherals such as those used in this project: **UM232H-B y MT9V111.** Each connector provides 12 pins, of which 8 are multipurpose, one pair is for power (PWR) and the other pair is for ground (GND).

BASYS 3 Connection

This development board has a microUSB connector connected to a JTAG controller that goes directly to the FPGA, allowing in-circuit programming. Thanks to this connection, the system can be powered, programmed and debugged at the same time. For this project, which requires the system to boot without programming, the JP1 jumper must be set to the QSPI position so that when the FPGA is powered, it will automatically load the bitstream previously stored in the SPI flash memory during the production process.

3.1.3 USB Interface Connection with PC

The USB protocol via the UM232H-B module will be used as the connection interface between the PC and BASYS 3.

Figure 3.4 UM232H-B Module

This is a development module from manufacturer FTDI that integrates the FT232H IC (Integrated Circuit), which is a single-channel interface that can be configured to convert data between USB and protocols such as UART, SPI, I2C, 245 FIFO, among others.

The system does not impose specific requirements on the version of the USB protocol used, as the FT232H chip is compatible with UHCI, OHCI, and EHCI controllers, ensuring its operation on USB 1.1, 2.0, and 3.0 ports (although it may not be able to take full advantage of their performance).

Unlike other FTDI chips, this one is capable of operating in synchronous and asynchronous modes, making it ideal for systems where fast and flexible communication is required [5].

For this project, it will be configured using its "***FTProg***" utility, programming its EEPROM to work in FT245 asynchronous FIFO mode (Figure 3.5).

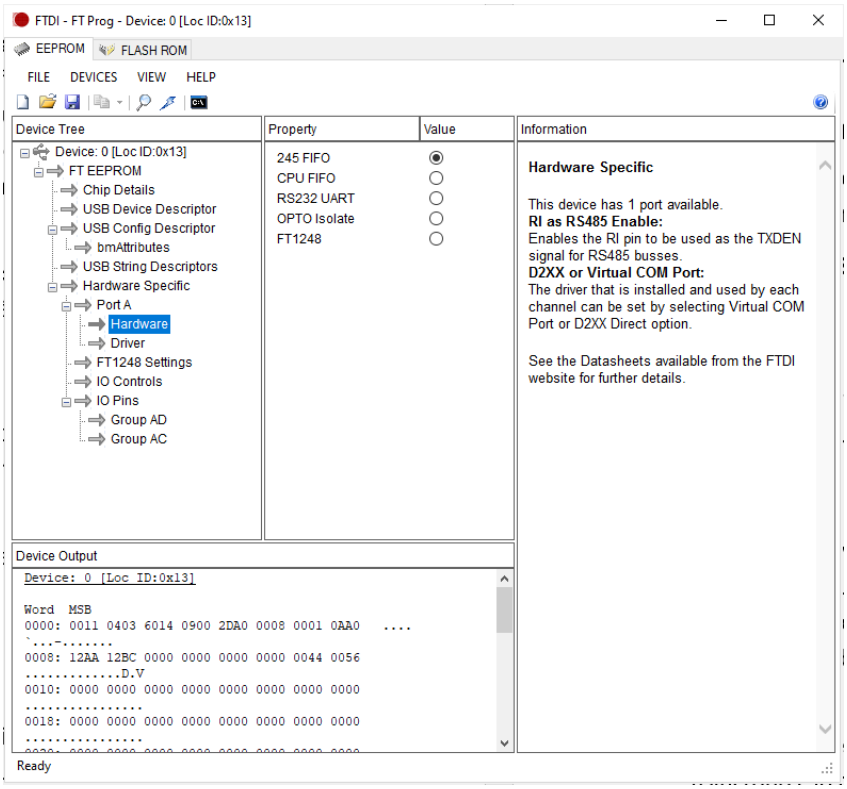


Figure 3.5 FTDI configuration from FTProg [17]

This is a simple 8-bit parallel communication protocol, in which data is transmitted using control signals: **`RD#`**, **`WR#`**, **`RXF#`** and **`TXE#`**.

This module has been chosen for its ease of integration through standard libraries with support in various programming languages. Furthermore, from a hardware perspective, it is compatible with the PMOD on the development board and has a 5V power output, which could be used in the future to power the FPGA directly without the need for a microUSB cable, making the system even more portable.

3.1.4 Image Sensor

The most important component, which will largely determine the system's performance, is the CMOS image sensor, integrated on a SOC known as MT9V11, manufactured by *Micron*. Among the highlights are [6]:

Image Resolution

- The sensor offers a VGA resolution of 640x480 pixels.
- It uses an Active Pixel Sensor (APS) matrix, which provides good light sensitivity and low power consumption.
- It also supports *windowing* modes to capture partial regions of the image at higher speeds.

Image Frequency

- The sensor is capable of capturing images at a rate of up to 30 frames per second at full resolution.
- This frame rate will vary depending on the exposure settings, active window size, and input clock frequency (**XCLK**).

Pixel size and optical format

- Each pixel measures 6.0µm x 6.0µm, providing good sensitivity for standard lighting conditions.
- The optical format of the sensor is approximately 1/4", which corresponds to an active diagonal of about 4.8 mm, an industry standard that indicates compatibility with compact optics.

Data Interface

- The sensor delivers the digital image via an 8-bit parallel data bus (**DATA [7:0]**).
- The synchronisation of transmitted data is managed through the following signals:
    - **PCLK**: pixel clock.
    - **LVAL** (or HREF): indicates active lines.
    - **FVAL** (or VSYNC): indicates active frames.

Configuration

- This SoC allows various parameters to be configured via a bus I^2C.
- The sensor requires an external clock signal to operate (**XCLK**).
- Typically, a frequency of 27MHz is used, which is the maximum input frequency allowed to achieve maximum performance. In our case, as we cannot reach this frequency exactly, with the 100MHz system clock signal (**MCLK**), we will work at 25MHz.
- This clock will be used internally to generate all the synchronisation signals mentioned above.

3.1.5 Functional Blocks

Below is a detailed block diagram representing the hardware implementation developed in VHDL. It shows the connections between the different logic blocks, as well as the relevant control and data signals that enable the system to operate in real time.

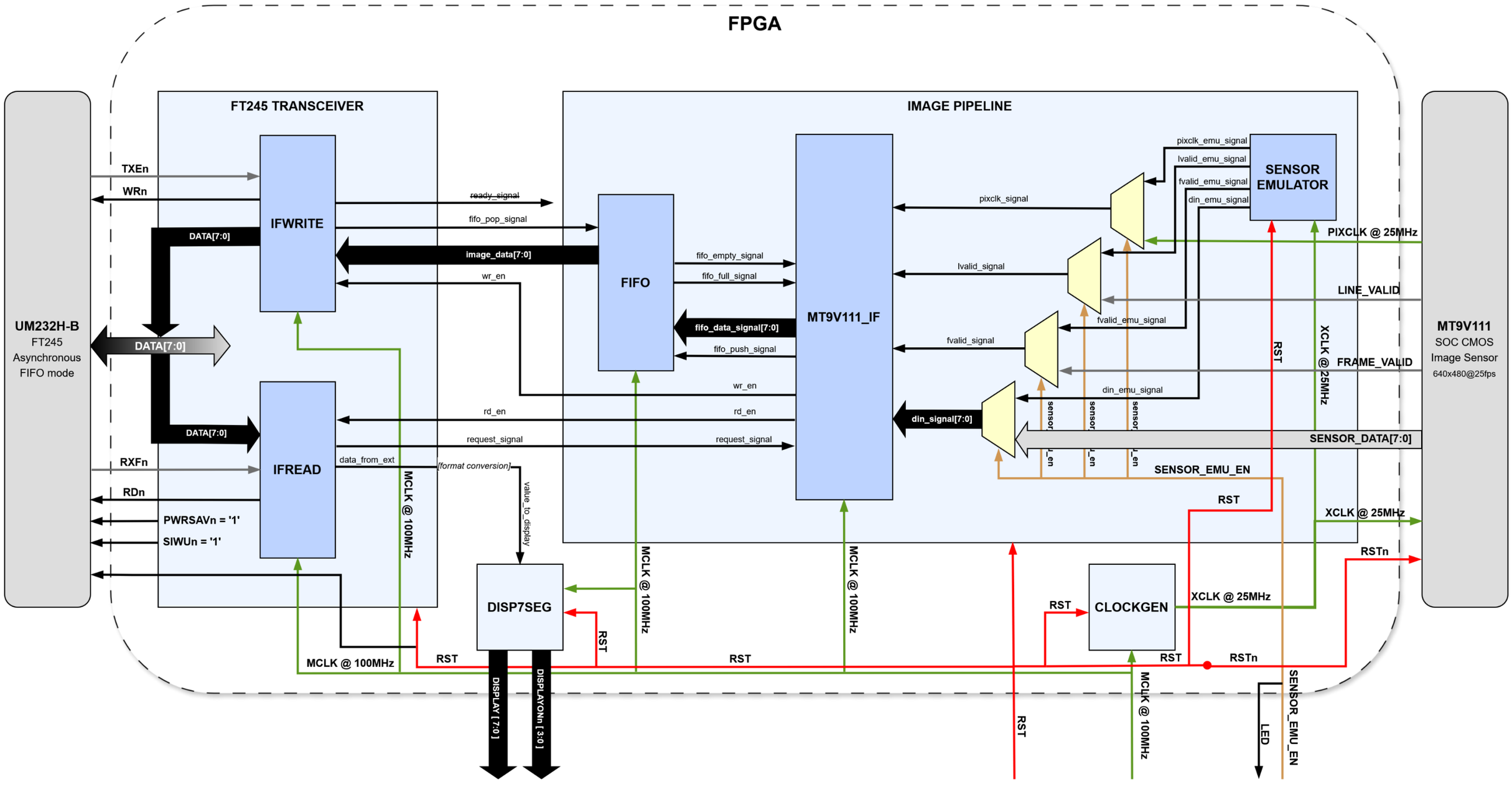


Figure 3.6 System hardware architecture

All signals ending in the letter “n” are active low (`TXEn`, `RXFn`, `DISPLAYONn`, etc.). The system has a single 100MHz clock domain or **Master Clock (MCLK)** generated by the FPGA. The entire system must be synchronous to this clock input, but as it has asynchronous control signals such as **`RXFn`** and **`TXEn`** of the UM232H-B module, these must be synchronised with the system clock. In order to minimise the risk of **metastability**, which can cause capture errors and unpredictable behaviour in synchronous logic, a **two-stage synchroniser** such as that in the Figure 3.7, consisting of a shift register with two ***Flip-Flops*** to synchronise each asynchronous signal and thus always use its synchronous and stable signals (`sync_signal`).

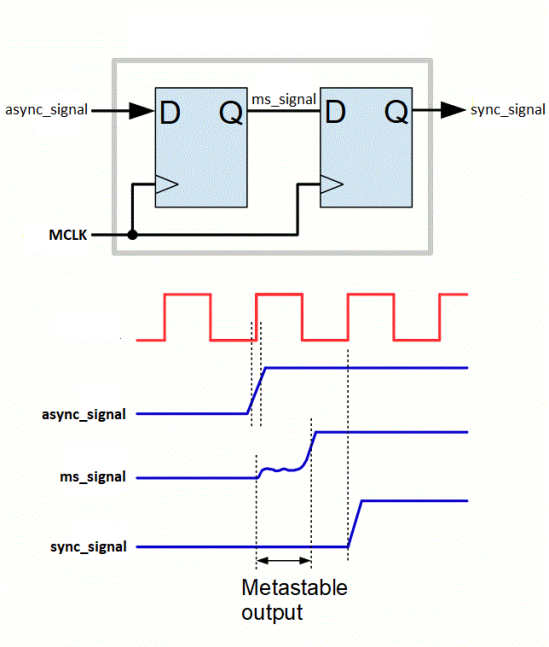


Figure 3.7 Synchroniser circuit 2-FF

The system has a global reset signal called **`RST`**, which is active at a high level and is used to initialise the entire system at the user's request via the **`BTNC`** button on the BASYS 3. An inverted version of this signal, **`RSTn`**, which is active at a low level and is mainly intended for the MT9V111 image sensor.

FT245 TRANSCEIVER

This functional block comprises the read (**IFREAD**) and write (**IFWRITE**) interfaces of the UM232H-B.

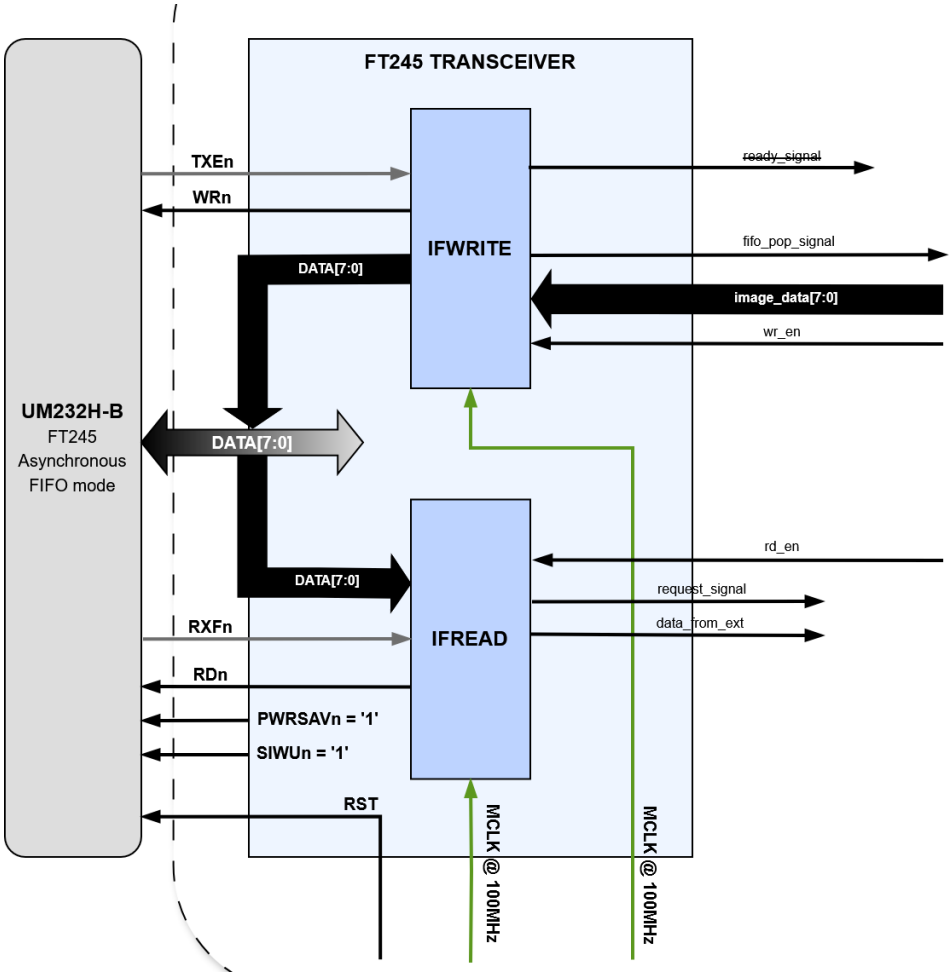


Figure 3.8 FT245 TRANSCEIVER block architecture

Both share an 8-bit bidirectional or inout (input/output) DATA bus, through which data is exchanged between the PC and FPGA. The rest of the control signals are independent and are described below:

- **IFREAD**

  This component is responsible for reading data from the UM-232H-B module when the PC sends information to the FPGA.

  - **Inputs:**

    RXFn: Low-level active flag. Indicates that data is available for reading from the UM232H-B module (status of its RX FIFO).

    - **0**: there is data in the buffer.

- **1**: buffer empty.

- **Outputs:**

  RDn: Control bit active at low level. Activated to read the data available in the USB RX FIFO.
  - **0**: the available data is read.
  - **1**: the data is not read.

  PWRSAVn: active at low level. Signal that tells the USB module not to enter power saving mode.
  - **0:** so that it does not enter power saving mode.
  - **1:** to enter power saving mode.

  In this design, this signal is set to **'0'** so that the device never enters power saving mode.

  SIWUn: active low. Keeps the chip awake, preventing the USB interface from suspending.
  - **0**: forces USB into '*sleep*' mode.
  - **1**: normal operation.

  In this design, this signal is set to **'1'** so that the USB operates normally.

- **IFWRITE**

  This component manages the sending of data from the FPGA to the PC via the UM232H-B module. It controls when writing is possible and generates the write signal.

  - **Inputs:**

    TXEn: Low-active flag. Indicates that the UM-232H-B's internal buffer has space to accept new data (status of its TX FIFO).
    - **0**: writing is possible.
    - **1**: buffer full or device not ready.
  - **Outputs:**

    WRn: Low-active control bit. Activates the writing of the current data from the TX FIFO to the USB bus.
    - **0**: write data.
    - **1**: do not write data.

These two read and write control interfaces described follow the corresponding timing specified by the manufacturer of the UM232H-B device configured in asynchronous FIFO mode [3]:

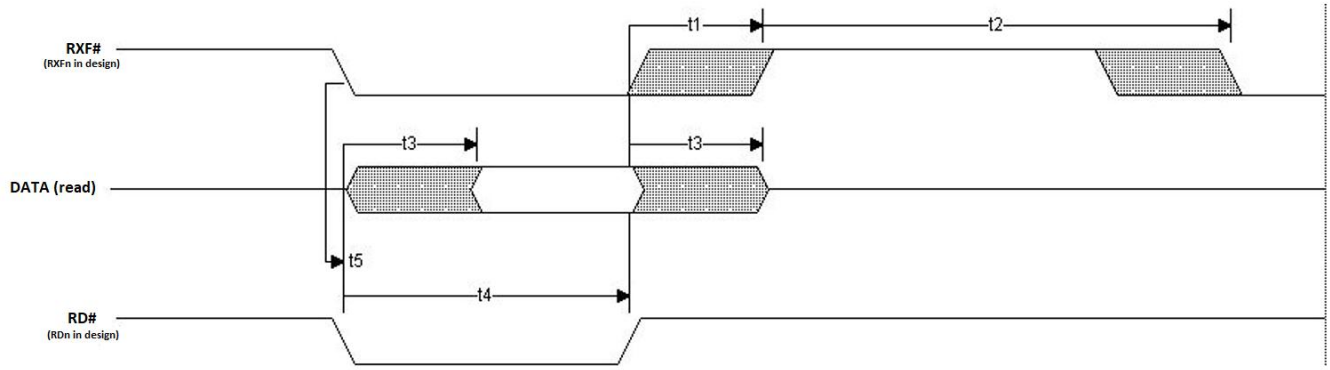


Figure 3.10 FT245 asynchronous FIFO interface – read cycle signals

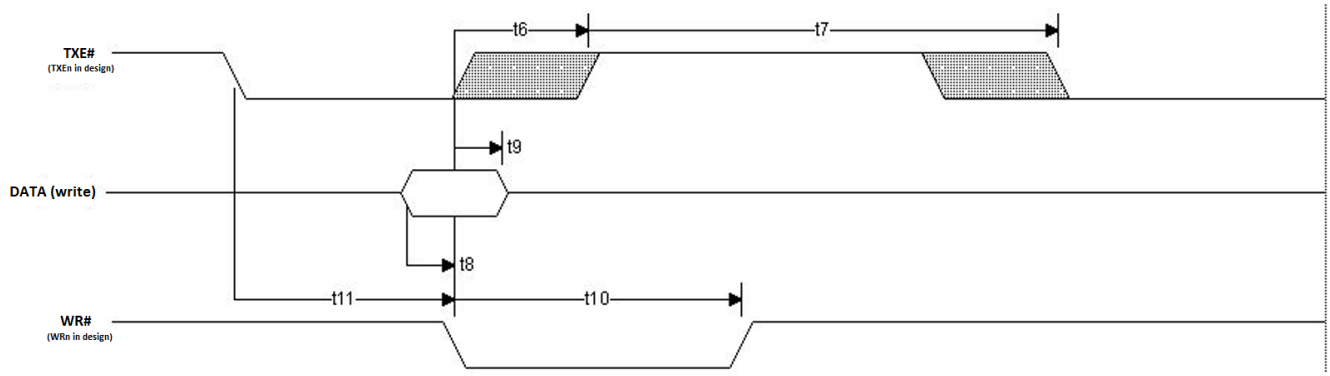


Figure 3.9 FT245 asynchronous FIFO interface – write cycle signals

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

Table 3.1 Asynchronous FIFO timing requirements

Before describing the implementation of this component, we will briefly review the role of the main signals involved. **TXEn** is the control signal generated by the FTDI to indicate to the FPGA that the write buffer is available to receive a new byte. The **WRn** signal is controlled by the FPGA and indicates to the FTDI that the data on the **DATA** bus is valid and can be captured.

Following these timing requirements (Table 3.1), The process of writing data in **IFWRITE** is described below (Figure 3.9):

1. In the initial state of the interface (**IDLE)**, the signals **TXEn** and **WRn** are at logic level '1' and, therefore, inactive. At this point, the system is indicated that the interface is ready to begin transmitting data via the **READY** output signal to **'1'.** Only when writing is enabled by the system, i.e. the **WR_EN** input is set to **'1'**, does the write cycle begin.
2. When this occurs, the system enters the **WAIT_FOR_TXE** state and indicates that the write interface is busy by setting the **READY** output signal to **'0'**. As the name of this state indicates, it waits for the synchronous version of the **TXEn** input signal **(TXEn_sync)** to be **'0'.** When it is, the system is requested to transmit new data by activating the **WRREQ** output signal to **'1'.** Furthermore, once the **DIN** data is ready, the **WRn** signal will be lowered to **'0'**, recording the data on the next MCLK edge (10ns later), to comply with the requirement of having the data ready at least 5ns before the **WRn** falling edge (**T8**). It advances to the **WRITE_DATA** state**.**
3. Once in this state, **WRREQ** is deactivated to **'0'** and, with **WRn** already at a low-level data writing begins, which lasts **NWCYCLES** of clock cycles. his constant has been set to 4 (i.e. 40 ns) to ensure that the **WRn** signal remains active for at least 30 ns to complete the write process (**T10**). This also meets the requirement that the data must remain stable on the bus for at least 5 ns after the start of writing (**T9**).
4. Once the write cycle is complete, **WRn** is set back to **'1'**. The **TXEn** signal must remain inactive for at least 49ns (**T7**) before being reactivated to transmit data. After this time, there would be two possible transitions: if writing is still enabled via the **WR_EN = '1'** input, t goes directly to the **WAIT_FOR_TXE** state; otherwise, it would return to the initial or **IDLE** state.

This writing process gives rise to the definition of **IFWRITE's FSM**, which describes the communication process from the FPGA to the PC. Taking into account that the cycle time or system clock period, **MCLK**, is 10ns (100MHz), the state diagram of this FSM is designed as follows:

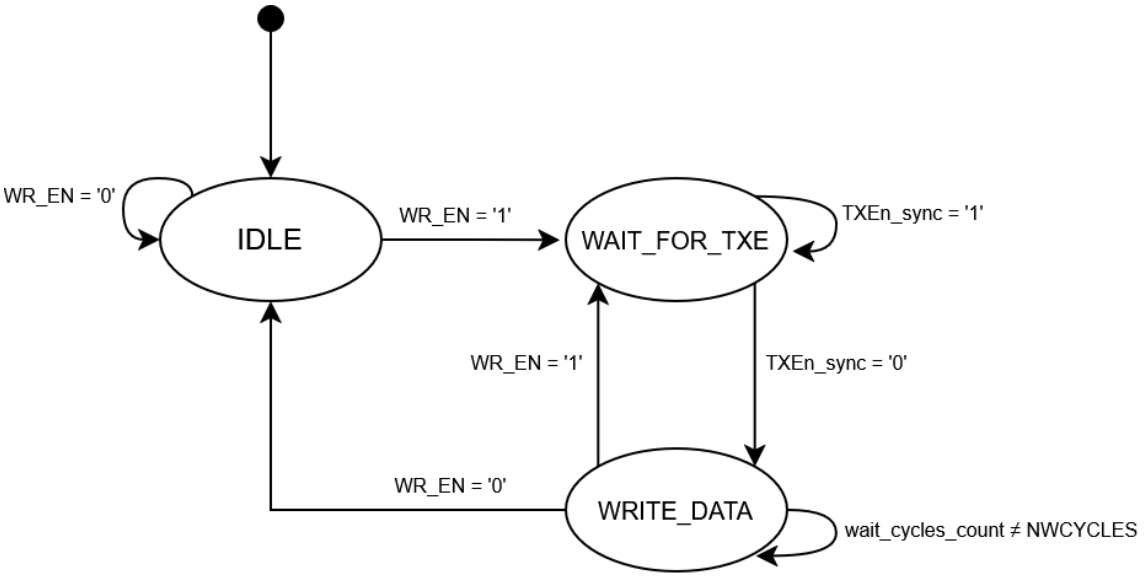


Figure 3.11 IFWRITE state diagram

State	WRn	WRREQ	READY	DOUT	Comment
IDLE	'1'	'0'	'1'	—	Waiting for write command
WAIT_FOR_TXE	'1'	'0' → '1'	'0'	—	Waiting for FTDI availability (TXEn_sync = '0')
WRITE_DATA	'0'	'0'	'0'	DIN	WRn active during NWCYCLES, DOUT stable
(End of cycle)	'1'	'0'	'1' (WR_EN='0')	—	Return to IDLE state or wait for next write cycle (WAIT_FOR_TXE)

Table 3.2 IFWRITE states and signals

Before describing the implementation of the **IFREAD** component, we briefly recall the role of the main signals involved. **RXFn** is the control signal generated by the FTDI to indicate to the FPGA that a new byte is available for reading. The **RDn** signal is controlled by the FPGA and indicates to the FTDI that the read process is active.

Following the timing requirements for these signals (Table 3.1), the process of reading data in **IFREAD** is described below (Figure 3.10):

1. In the initial state (**IDLE**) of the interface, the **RXFn** and **RDn** signals are at logic level '1' and are therefore inactive. t this point, the system is indicated that the interface has not yet received any data via the **RDREQ** output signal at **'0'**. Only when reading is enabled from the system, i.e. the **RD_EN** input is at **'1'**, does the read cycle begin.
2. When this occurs, the state changes to **WAIT_FOR_RXFn**. As its name suggests, it waits for the synchronous version of the **RXFn** input signal (**RXFn_sync**) to be **'0'.** When it is, the **RDn** signal will be lowered to **'0'** on the next MCLK edge (10ns later), although there is no minimum time required between the falling edge of **RXFn_sync** and that of **RDn** according to Table 3.1 (**T5** is 0ns). The system advances to the **READ_DATA** state.
3. Once in this state, with **RDn** already at a low level, data reading begins, which lasts **NRCYCLES** of clock cycles. This constant has been set to 4 (i.e. 40 ns) to ensure that the **RDn** signal remains active for at least 30 ns to complete the reading process (**T4**). In the last wait cycle of **NRCYCLES**, **RDREQ** is set to **'1'** and the **DIN** input data is captured.
4. Once the read cycle is complete, **RDn** is set back to **'1'**, indicating to the FTDI that the read operation is finished. From this point on, two timing requirements must be met according to the manufacturer's table. The first is **T1** (between 1 and 14ns), which corresponds to the time from when **RDn** is deactivated until **RXFn** can be reactivated. The second affects the **RXFn** signal, which must remain inactive for at least 49ns (**T2**) before being reactivated to indicate that new data is available to be read. After this time, there would be two possible transitions: if reading is still enabled via the **RD_EN = '1'** input, it goes directly to the **WAIT_FOR_RXFn** state; otherwise, it would return to the initial or **IDLE** state.

This reading process gives rise to the definition of the **IFREAD FSM**, which describes the communication process from the PC to the FPGA. The state diagram for this **FSM** is as follows:

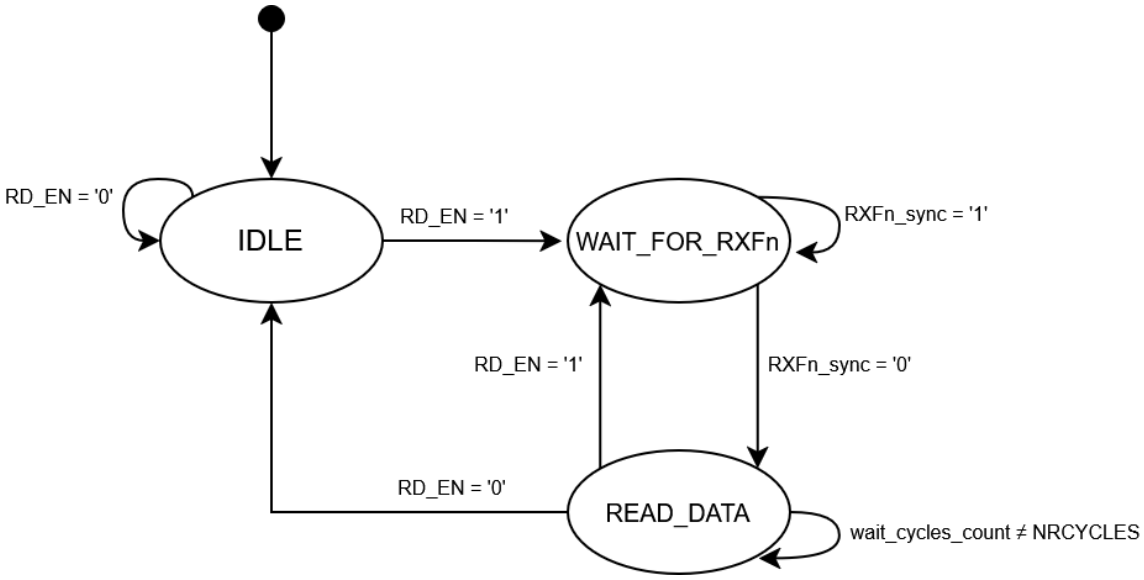


Figure 3.12 IFREAD state diagram

State	RDn	RDREQ	DIN	Comment
IDLE	'1'	'0'	—	Initial state. RXFn not yet active. Wait for RD_EN = '1'.
WAIT_FOR_RXFn	'1'	'0'	—	Wait until RXFn_sync = '0'. RDn is activated in the next cycle.
READ_DATA	'0'	'0' → '1'	DIN captured in wait_cycles_count = 1	RDn remains active during NRCYCLES, DIN is captured, RDREQ is generated.
(End of cycle)	'1'	'0'	—	RDn is deactivated. Transition to WAIT_FOR_RXFn or IDLE depending on RD_EN.

Table 3.3 IFREAD states and signals

CLOCK GENERATOR

This component is responsible for generating the **XCLK** clock output, which will be the input for the MT9V111 SoC, as mentioned above. This generator consists of a frequency divider, implemented by means of a binary counter that counts cycles (constant **NCYCLES + 1)** of the input clock. In this case, the input clock signal is 100MHz (**MCLK**) and the target is, for example, 25MHz. We will have to divide our input by 4 (**DIVIDER**), so, following the formula:

```
2*(NCYCLES + 1) = DIVISOR
```

The constant will need to be defined **NCYCLES = 1**.

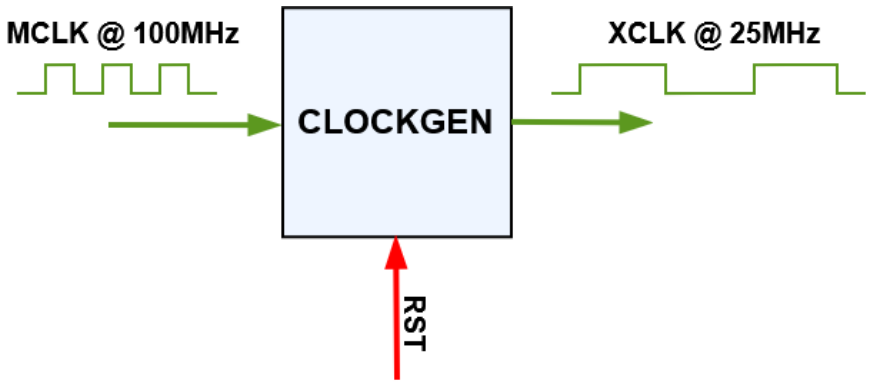


Figure 3.13 CLOCK GENERATOR block architecture

Alternatively, the FPGA has dedicated **MMCM/PLL** blocks, which allow clocks to be generated with greater precision in frequency, duty cycle and phase, as well as distributing them throughout the global clock network. In this case, given that the application only requires division by a simple integer value, a logic divider was chosen for simplicity of implementation, although the use of an **MMCM** would be the most recommended option in a production design.

7 SEGMENTS DISPLAY

This block implements control of the 4-digit 7-segment display incorporated in the BASYS 3, described in the section 3.1.2 BASYS 3 Development Board.

Although four digits are available, only three-digit natural numbers will be represented, i.e. from 0 to 999.

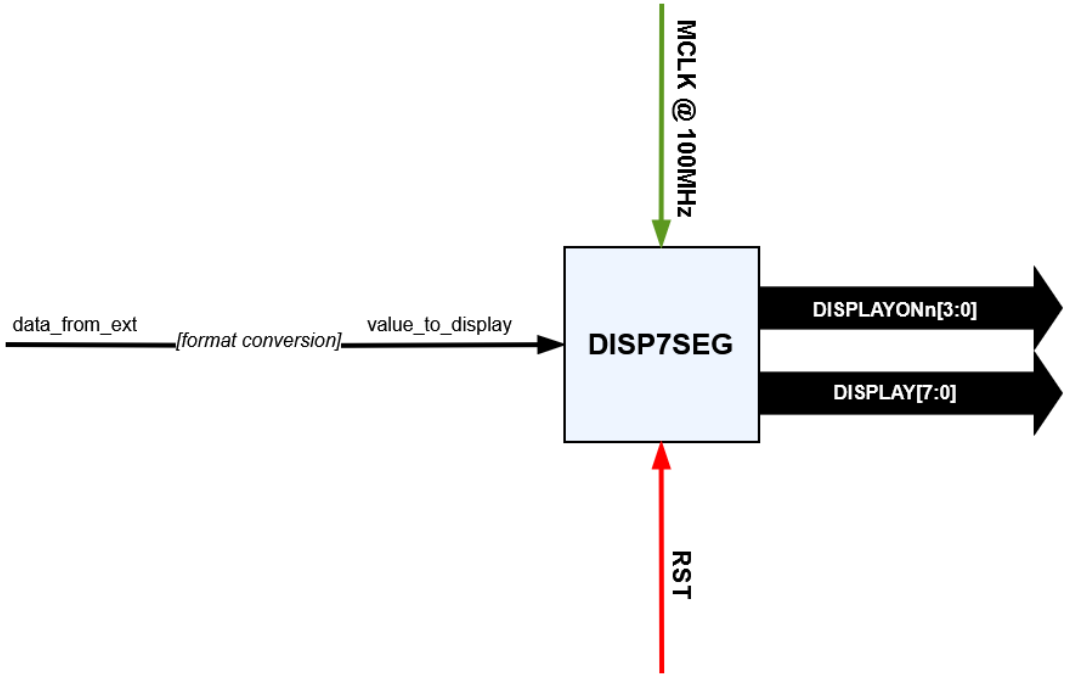


Figure 3.14 7 SEGMENTS DISPLAY block architecture

Essentially, this component breaks down the **`value_to_display`** number into hundreds, tens and units. It then cyclically activates one of the three digits, i.e. each anode corresponding to the digit (**`DISPLAYONn`**) at a speed high enough for the human eye to perceive all three lighting up at the same time. It also generates the signals needed to activate the cathodes (**`DISPLAY`**) corresponding to the segments that make up each number. When the external **`RST`** signal is activated, the display shows **`"000"`**.

IMAGE PIPELINE

This functional block is responsible for acquiring real images captured by the **MT9V111** image sensor or emulated from the **SENSOR EMULATOR** component. When received through the **MT9V111_IF** interface, they are stored in the form of bytes containing pixel information before being sent to the PC through the **FT245 TRANSCEIVER** block described above. This storage is done so that no images or frames are lost, nor are they corrupted by losing information within the same frame.

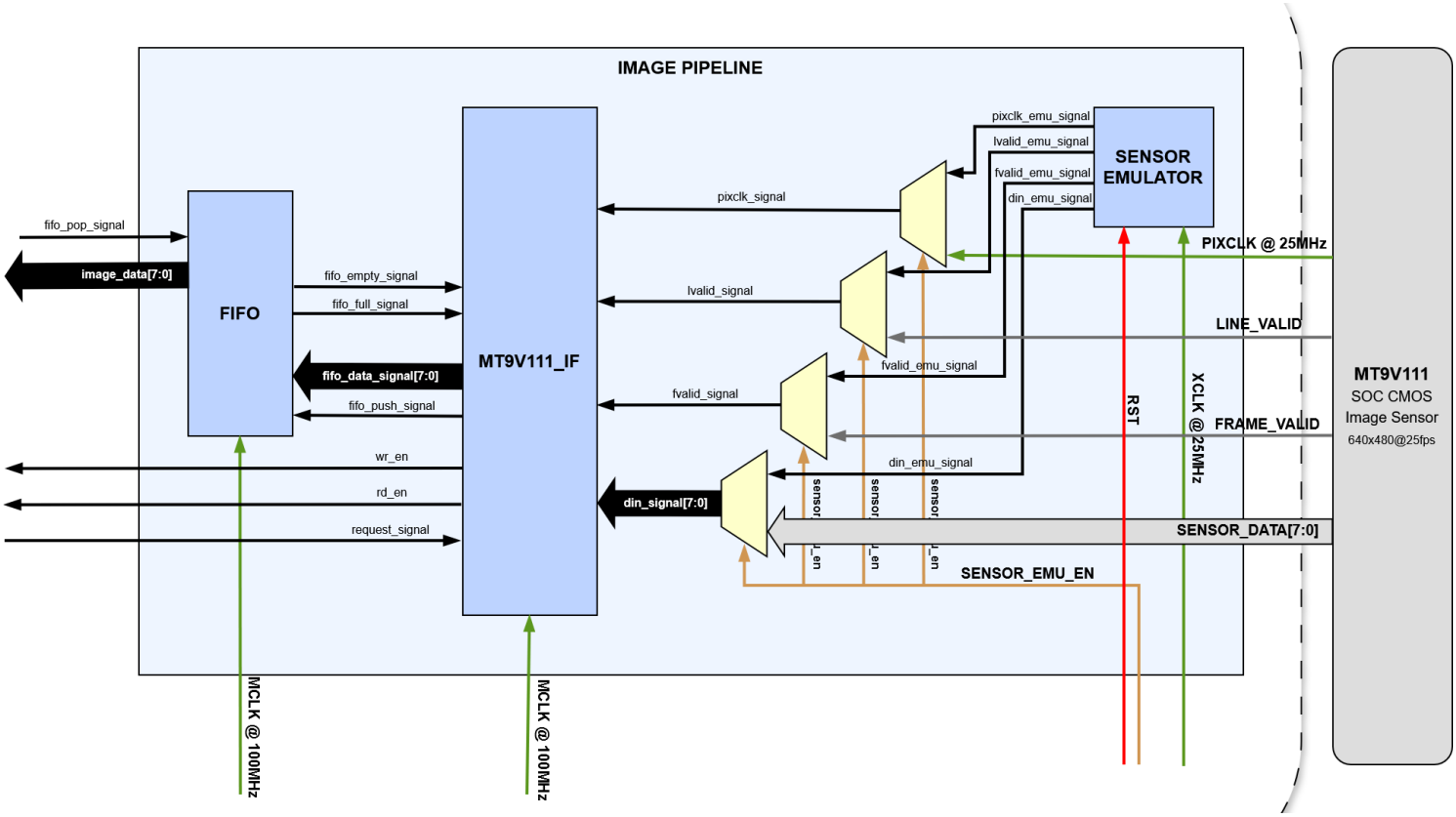


Figure 3.15 IMAGE PIPELINE block architecture

FIFO

Reading the image from the sensor is faster than transmitting it to the PC, so it is necessary to implement FIFO RAM in the system, which allows the data received from the image sensor to be stored and ensures that it is not lost and that there are no data integrity issues while it is being sent to the PC.

Modern FPGAs, such as the Artix-7 (BASYS 3), have dedicated memory blocks called **Block RAM (BRAM)**, designed precisely to store large amounts of data synchronously and efficiently. Automatically, when processing the RTL design of the FIFO, the synthesiser detects the pattern and uses BRAM or LUTRAM (unless otherwise specified with attributes or pragmas, which is not the case).

FIFO has been implemented using internal memory of this type, specifically with **16 RAMB36E1 blocks** (36Kbit memories). As indicated in the FPGA usage report in 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

Figure 3.16 Post-implementation FPGA utilisation report in Vivado

This has been done automatically by the Vivado synthesiser upon detecting the **FIFO** configuration with **8 bits of data bus width** or word size (**`DATA_WIDTH`**) and **16 bits of address bus width (`ADDR_WIDTH`)**. This translates into a storage capacity of:

2^{16} = 65536 words * 8 bits/word * 1 byte/8 bits * 1kbyte/1024 bytes = **64KB**

The size of this FIFO was established based on a worst-case write cycle simulation. Since it must be ensured that while writing, the FIFO must be able to store data that cannot be sent via **IFWRITE** in the worst case, i.e., its longest write cycle. To do this, the worst-case times from the manufacturer's timing requirements table seen above are collected, in which the FTDI only allows you to write once every:

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

Note: Although the minimumT_{11} according to the datasheet is 0ns, 20ns are added due to the latency of two **`MCLK`** cycles introduced by the two-stage synchroniser through which the **`TXEn`** input signal passes.

Considering that the system operates with **`MCLK`** periods of **10ns**, this sum of times is adjusted:

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

Adding a 10% safety margin for greater robustness:

$\mathbf{T_{worst}} = T_{total} * 1.1$ = 99ns ≈ **100ns**

A simulation of the entire system is performed with **`TXEn`** cycles of **100ns**. The **SENSOR EMULATOR** component is used for this purpose, observing that the FIFO never fills up and does not exceed its capacity when set to **64KB**. This simulation will be discussed further in the testing and verification chapter.

For this dual-port memory block (Dual-Port RAM), a control logic based on two signals is implemented: **`POP`** and **`PUSH`**. These indicate when data should be

released (**DOUT**) or stored (**DIN**), respectively. The FIFO status logic depends on the value of the stored word counter, with **EMPTY** active when there is no data stored (**WORD_COUNTER_REG = 0**) and **FULL** active when the memory is full (**WORD_COUNTER_REG = 2^{16}**). Its operating principle is that of a circular memory as shown in Figure 3.17. In the implementation of this project, in addition to the control signals referring to the pointers (**RPOINTER_REG** and **WPOINTER_REG**), an extra control signal called **WORD_COUNTER_REG** has been used to determine the exact degree of memory occupancy and validate its size, as seen above.

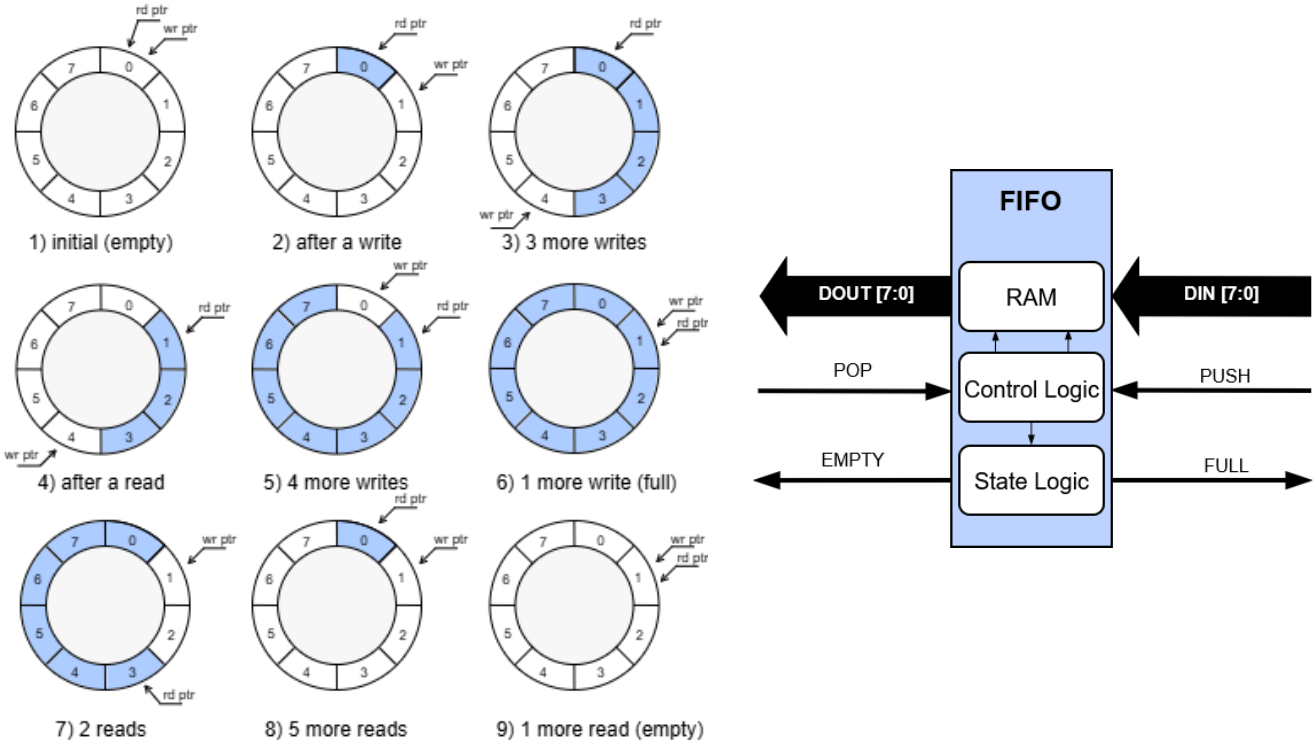


Figure 3.17 FIFO component architecture and implementation

MT9V111_IF

This block acts as an interface with the MT9V111 sensor. As its name suggests, its purpose is to receive image data from the sensor and transfer it to the system's FIFO, respecting control signals and avoiding information loss. This sensor operates synchronously with the falling edges of its **PIXCLK** clock. his clock is simply a delayed copy of the **CLKIN** input clock (renamed **XCLK** in this design), generated by the **CLOCKGEN** component from the FPGA, with a slight delay (as shown in the timing diagrams in Figure 3.18).

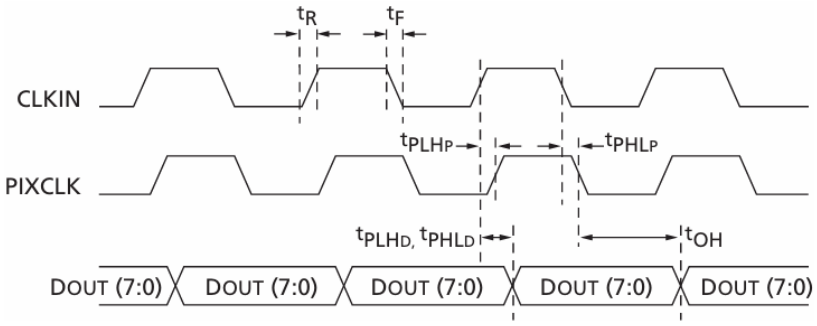


Figure 3.18 PIXCLK and DOUT propagation delays

The operation of this component is based on a finite state machine (FSM) that manages the data flow from the real or emulated sensor (**SENSOR EMULATOR** component) and controls access to the intermediate storage FIFO before it is sent to the PC.

To do this, it first synchronises the **PIXCLK** sensor clock signal with the **MCLK** system clock using a two-stage synchroniser. The image data is sent by the sensor in YUV4:2:2 format, even though each byte is treated individually.

This format is a colour representation used in digital video, based on separating the luminance information (Y) representing brightness from the chrominance information (U or Cb and V or Cr) representing colour (blue and red tones respectively). This type of format allows for better colour compression without seriously affecting perceived quality, because the human eye is more sensitive to brightness than to colour. The notation used, 4:2:2, refers to the subsampling: 4 Y samples, 2 U (Cb) samples and 2 V (Cr) samples. This means that, in sensors such as this, the data comes out as a sequence of bytes:

Cb0, Y0, Cr0, Y1, Cb2, Y2, Cr2, Y3…

where each group of 4 bytes represents 2 pixels:

Byte	Content	Description
1	U0	Blue chrominance
2	Y0	Pixel luminance 0
3	V0	Red chrominance
4	Y1	Pixel luminance 1

As shown in the data output timing diagram in the sensor datasheet [6]:

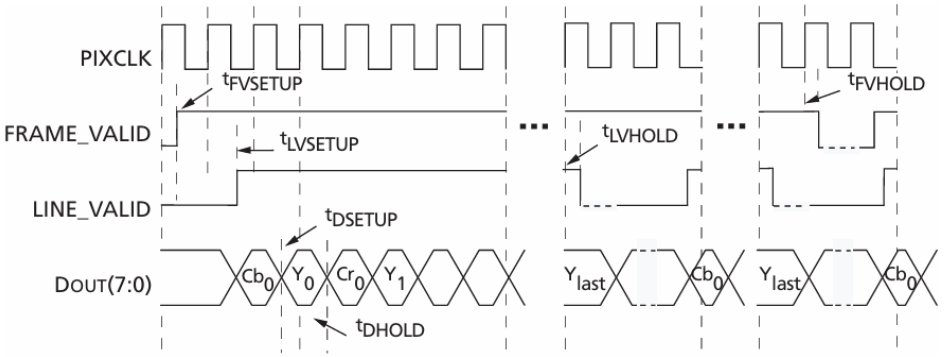


Figure 3.19 MT9V111 output data timing diagram

This means that, if you want to send the image in greyscale, only the even bytes corresponding to the luminance should be captured. This reduces the amount of information to be transmitted and therefore increases the number of frames per second (FPS) transmitted to the PC. In this case, both colour images and greyscale images have been tested. However, it was ultimately decided to send them in YUV format, as required by the image processing model, as will be explained later.

The control logic of this FSM controls key signals already mentioned, such as **`WR_EN`**, **`RD_EN`** and **`FIFO_PUSH`**.

The process of capturing an image from this **MT9V111_IF** interface is described below:

1. The interface starts from its initial or **OFF** state. This state indicates that the capture process is paused, waiting for the system (specifically, the software on the PC) to request a new image capture, i.e., to activate the input signal **`FRAME_REQ = '1'.`**
2. Once it receives the request for a new frame, the FSM moves to the **IDLE** state, where it waits for the rising edge of the **`FRAME_VALID`** input

signal coming from the real or emulated sensor. This signal indicates that the sending of a new frame has begun, which is internally translated into the signal **sof_signal='1'**.

3. After detecting the start of transmission of a new image, the system switches to **BLANKING** mode. In this mode, data received that does not contain image information is processed and used as synchronisation periods in the transmission process by the sensor. In this case, the system waits for the **LINE_VALID** input signal from the sensor to activate to '1'. This indicates that the first line of pixels is about to be transmitted.
4. Once this signal is active, it switches to the capture state (**CAPTURE**) if valid data corresponding to the transmitted image is received. At this point, the **WR_EN** signal (**FT245_TRANSCEIVER** block) is activated while the FIFO is not empty (**FIFO_EMPTY = '0'**), in addition to activating **FIFO_PUSH** to store the received byte (**DIN**) in the FIFO. This is done on each **MCLK** clock edge until **LINE_VALID = '0'** is detected, indicating that the first row of the image has finished transmitting.
5. After this, it returns to the **BLANKING** state, where it again waits for **LINE_VALID** to be activated and again moves to the **CAPTURE** state until it is deactivated again.
6. Finally, this process is repeated until, in the **BLANKING** state, the falling edge of **FRAME_VALID** is detected, indicating the end of the frame, internally referred to as **eof_signal = '1'.** It then returns to the **OFF** state if the FIFO is empty (**FIFO_EMPTY = '1'**) or to the **FLUSH** state if the FIFO still has data. In this case, **WR_EN** is activated and the data is sent without activating the **FIFO_PUSH** signal, until the FIFO is completely empty. Then, it returns to the **OFF** state and the process starts again.

The state diagram for this FSM is shown below.

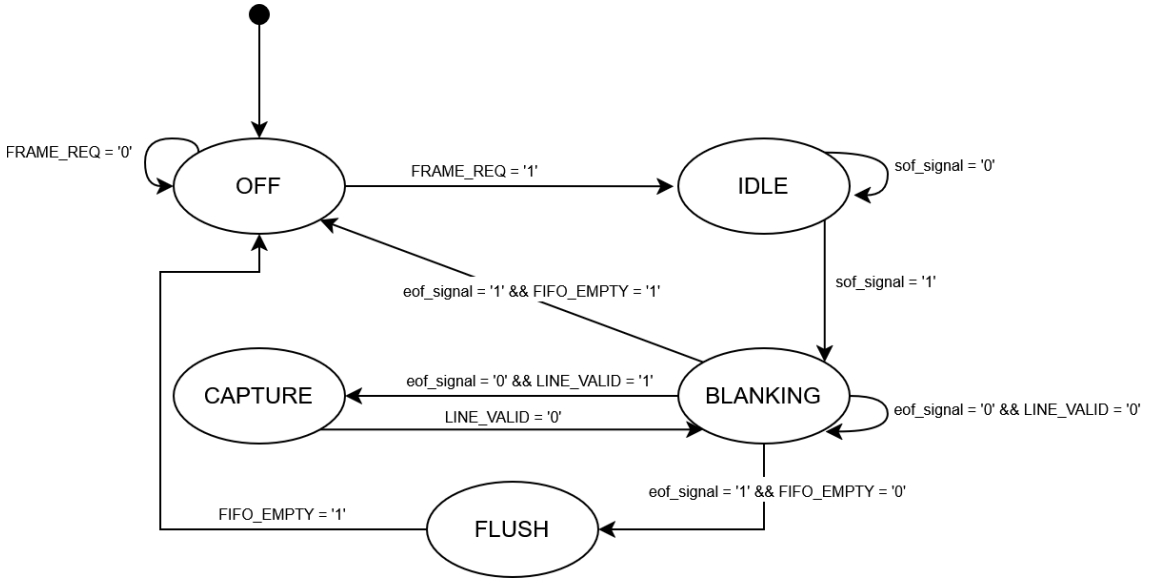


Figure 3.20 MT9V111_IF state diagram

State	FIFO_PUSH	WR_EN	RD_EN	IMAGE_DATA	Comment
OFF	'0'	'0'	'1'	(others => '0')	Wait for FRAME_REQ = '1' to proceed to IDLE.
IDLE	'0'	'0'	'1'	Last value	Wait for sof_signal = '1' to start capture.
BLANKING	'0'	not FIFO_EMPTY	'0'	(others => '0')	Wait between lines. If LINE_VALID = '1', proceed to CAPTURE.
CAPTURE	data_valid_signal	not FIFO_EMPTY	'0'	DIN	Data capture while LINE_VALID = '1'. On each rising edge of pixclk_sync, data_valid_signal is '1'.
FLUSH	'0'	not FIFO_EMPTY	'0'	DIN	Empties the FIFO after the end of the frame (eof_signal) if there is pending data.

Table 3.4 MT9V111_IF states and signals

SENSOR EMULATOR

This component has been developed to emulate the operation of the MT9V111 image sensor, allowing verification of the correct operation of the interface (**MT9V111_IF**), as well as the entire process of sending data to the PC without the need for actual hardware, all through simulations. This emulation is based on the fully controlled generation of the **PIXCLK**, **FRAME_VALID**, **LINE_VALID** and **IMAGE_DATA** signals. To enable this component, the **SW0** switch on the BASYS 3 must be activated or moved upwards. If it is not, the system will operate with the actual sensor.

Its operation is based on the timing parameters indicated in the datasheet corresponding to the active image and blanking (idle) periods. This allows for a deterministic and reproducible environment.

The behaviour of the emulator consists of a finite state machine (FSM) with 5 states:

- **SOFBLANKING**
  Initial frame blanking. Activates the signal **FRAME_VALID**.
- **ACTIVE**
  Sending valid data. Activate **LINE_VALID** and transmit pixel information.
- **HBLANKING**
  Blanking or pause between active lines. **LINE_VALID = '0'**.
- **EOFBLANKING**
  End-of-frame blanking. Last cycles with **FRAME_VALID = '1'**.
- **VBLANKING**
  Wait between frames. **FRAME_VALID = '0'**.

The change between these states is controlled by counters that accurately simulate the actual sensor times [6]:

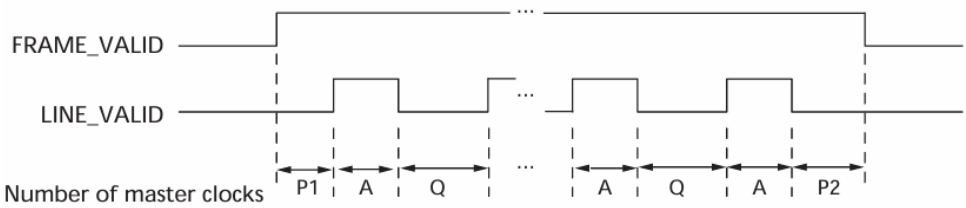


Figure 3.21 Frame control signal timing

Parameter	Name	Equation (Master Clocks)	Default Timing At 12 MHz
A	Active Data Time	(Reg0x04 - 7) x 2	= 1,280 pixel clocks = 1,280 master clocks = 106.7us
P1	Frame Start Blanking	(Reg0x05 + 112) x 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	(Reg0x05 + 121) x 2 (MIN Reg0x05 value = 9)	= 318 pixel clocks = 318 master clocks = 26.5us
A + Q	Row Time	(Reg0x04 + Reg0x05 +114) x 2	= 1,598 pixel clocks = 1,598 master clocks = 133.2us
V	Vertical Blanking	(Reg0x06 + 9) x (A + Q) + (Q - P1 - P2)	= 20, 778 pixel clocks = 20,778 master clocks = 1.73ms
Nrows x (A + Q)	Frame Valid Time	(Reg0x03 - 7) x (A + Q) - (Q - P1 - P2)	= 767,036 pixel clocks = 767,036 master clocks = 63.92ms
F	Total Frame Time	(Reg0x03 + Reg0x06 + 2) x (A + Q)	= 787,814 pixel clocks = 787,814 master clocks = 65.65ms

Table 3.5 Frame times

The sensor allows a maximum **PIXCLK** frequency of **27MHz**, but since the system clock, **MCLK**, is **100MHz**, this frequency cannot be reached with dividers. **The maximum frequency that can be configured by the system is 25MHz**.

This emulator allows you to simulate different frequency values for **PIXCLK** by simply adjusting the counters to align with the times shown in the Table 3.5:

```
architecture Behavioral of SENSOR_EMULATOR is

------------------------------------------------------------------------------------
--  Pixel clocks counter signals to emulate real sensor parameters                --
------------------------------------------------------------------------------------
                                                        -- PIXCLK clocks  | 12MHz (DEFAULT) |     25MHz      |
 -- Need 2 bytes to send data of 1 pixel (NUM_PIXELS * 2)  -----------------|-----------------|----------------|
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) := x"14";
constant IMAGE_PATTERN_FRAME_END_BYTE       : STD_LOGIC_VECTOR(7 downto 0) := x"28";
constant IMAGE_PATTERN_LINE_START_BYTE      : STD_LOGIC_VECTOR(7 downto 0) := x"97";
constant IMAGE_PATTERN_LINE_END_BYTE        : STD_LOGIC_VECTOR(7 downto 0) := x"17";
constant IMAGE_PATTERN_HBLANKING_BYTE       : STD_LOGIC_VECTOR(7 downto 0) := x"A5";
constant IMAGE_PATTERN_VBLANKING_BYTE       : STD_LOGIC_VECTOR(7 downto 0) := x"45";
```

Figure 3.22 Definitions in the EMU SENSOR component in VHDL

By maintaining the number of **PIXCLK** clocks counting in each case and increasing the frequency, the time it takes to reach these clocks is reduced. Therefore, the total frame time is reduced from 65.65ms at 12MHz (default

frequency) to 31.512ms when the frequency is increased to the maximum mentioned above, i.e. 25MHz.

The synthetic image to be transmitted by this emulator, with a resolution of 640x480, is based on a rising pattern or gradient with a step of 256 pixels (**`IMAGE_GRADIENT_PATTERN_STEP`**). To facilitate the identification of the different **BLANKING** states in simulation, different constant values have been defined to be transmitted in each of them, in order to differentiate them. The same applies to the start and end of **FRAME** and the start and end of **LINE**.

The state diagram for this component is shown below:

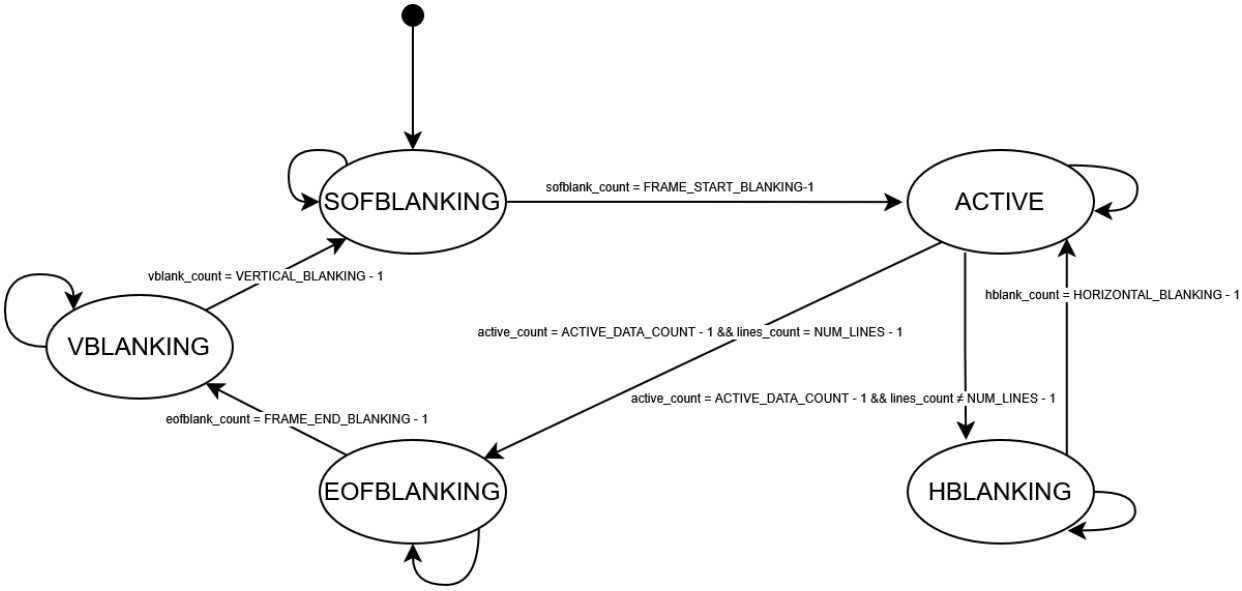


Figure 3.23 State diagram of SENSOR EMULATOR

State	FRAME_VALID	LINE_VALID	IMAGE_DATA	Comments
SOFBLANKING	'1'	'0'	`IMAGE_PATTERN_FRAME_START_BYTE`	Frame start preamble (initial blanking).
ACTIVE	'1'	'1'	GET_HEX_PIXEL_VALUE (`active_count`)	Transmission of synthetic image data.
HBLANKING	'1'	'0'	`IMAGE_PATTERN_HBLANKING_BYTE`	Time between lines (horizontal blanking).
EOFBLANKING	'1'	'0'	`IMAGE_PATTERN_FRAME_END_BYTE`	Indicates the end of frame transmission.
VBLANKING	'0'	'0'	`IMAGE_PATTERN_VBLANKING_BYTE`	Time between consecutive frames.

Table 3.6 SENSOR EMULATOR states and signals

3.1.6 I/O Pin Assignment

An important part of the hardware module design and implementation process is defining the FPGA's input and output connections. To do this, Xilinx provides an ***.xdc*** *(Xilinx Design Constraints)* file, which is a design constraints file that defines how the internal logic signals of the VHDL project, specifically the **TOP**, are connected to the physical pins of the FPGA.

This is a text file with TCL-based syntax, where the following are indicated:

- Pin assignments (*PACKAGE_PIN*).
- The direction of the signal (input or output).
- The electrical standard for the pin (*IOSTANDARD*).
- In some cases, temporary restrictions (*create_clock, set_input_delay*, etc.).

Specifically, these input/output signals are available on the BASYS 3 development board, accessible via the **PMOD** connectors described above. Here, the logic signals will therefore be defined with the pins where the following are connected: real image sensor (**MT9V111**), FTDI module (**UM232H-B**), **7-segment display**, **reset button** and a **switch** to enable/disable the sensor emulator.

Each **PMOD** pin on each port is internally associated with an FPGA pin, and each of these has been assigned an internal logic signal from the system:

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

Table 3.7 PMOD pin assignment and internal signals of the design

In this file, the clock signal is also created on the corresponding pin, with the following lines:

```
## 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}]
```

In addition, connecting the various components integrated into the BASYS 3 itself to the FPGA pins:

```
##  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]}]
```

The LED is used to indicate whether or not the sensor emulator is enabled.

After completing the design and this constraint file, the following steps must be completed to obtain a binary file to load into the FPGA from the hardware description made in VHDL. This file is called a **bitstream**, and these are the steps to follow:

- **Synthesis**
  This process compiles our RTL design described in VHDL, generating a network of logic gates, *Flip-flops*, other primitives such as BRAMs, and abstract interconnections (***netlist***). It verifies that the hardware description is syntactically and semantically correct. It optimises the design according to the required logic and the selected options.
- **Implementation**
  This process follows synthesis and assigns or maps the generated netlist to the physical resources of the FPGA: **LUTs** (Look-Up Tables), **Flip-Flops** (FFs), **BRAMs**, and **IOBs** (I/O blocks). In addition, it also places the components on the physical chip matrix (**placement**), physically connects these blocks or components (**routing**) and checks that the design complies with timing constraints (**timing analysis**). This is the report obtained after this stage, showing the FPGA resource usage:

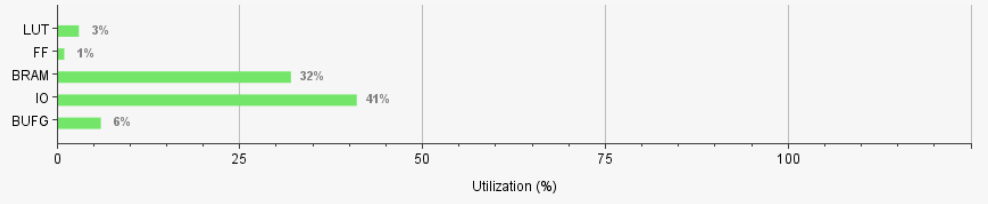


Figure 3.24 Post-implementation FPGA utilisation report

- **Bitstream generation**
  Based on the design already implemented, the ***.bit*** file is generated. This file contains all the binary information that configures the FPGA internally. This **bitstream** is stored in external non-volatile memory, which in the case of the BASYS 3 is **QSPI Flash** memory. This allows the Artix-7 FPGA to communicate with this memory during boot-up, automatically loading the design without the need to manually reprogram from Vivado each time [7].

3.2 Software Module

3.2.1 Introduction

The design and implementation of the software developed in this project allows for easy interaction with the hardware module. To this end, an application has been created entirely in **Python**, using **PySide6**—the official Qt6 binding for Python—for the implementation of the **graphical user interface (GUI)**. This application was developed in **Visual Studio Code** (VS Code) due to its flexibility as an IDE and its integration with Python and ***Conda***.
It uses both the Qt API through the PySide6 library and a ***designer***, which is a graphical tool that allows interfaces to be designed visually, facilitating the creation and organisation of elements such as buttons, menus and windows.

As mentioned above, **multithreaded** execution (**PySide6 QThread**) is used to keep the graphical interface responsive while image acquisition and processing tasks are being performed.

The software architecture is based on several main classes, including the **Controller**, the **View** (GUI), the **Image Processor**, and a **Mediator** that manages communication between them. In addition to proprietary code, third-party libraries such as **ftd2xx** have been used to control the **UM232H-B** device, and image processing tools have been used to run the trained neural model integrated into the system.

Development has been carried out in the Visual Studio Code environment, using a virtual environment managed with ***Conda*** to facilitate the installation and isolation of project dependencies (PySide6, ftd2xx, etc.). This ensures portability and control over their versions [8].

For more details on how to create this environment or all the necessary dependencies, see the **README** file for the project's software module on GitHub.

The installation guide for the ***SpeedTrafficSignRecognitionApp*** application can be found in **Appendix B. Software Installation** at the end of this document.

3.2.2 General Diagram

The following figure shows the execution flow diagram of the developed application.

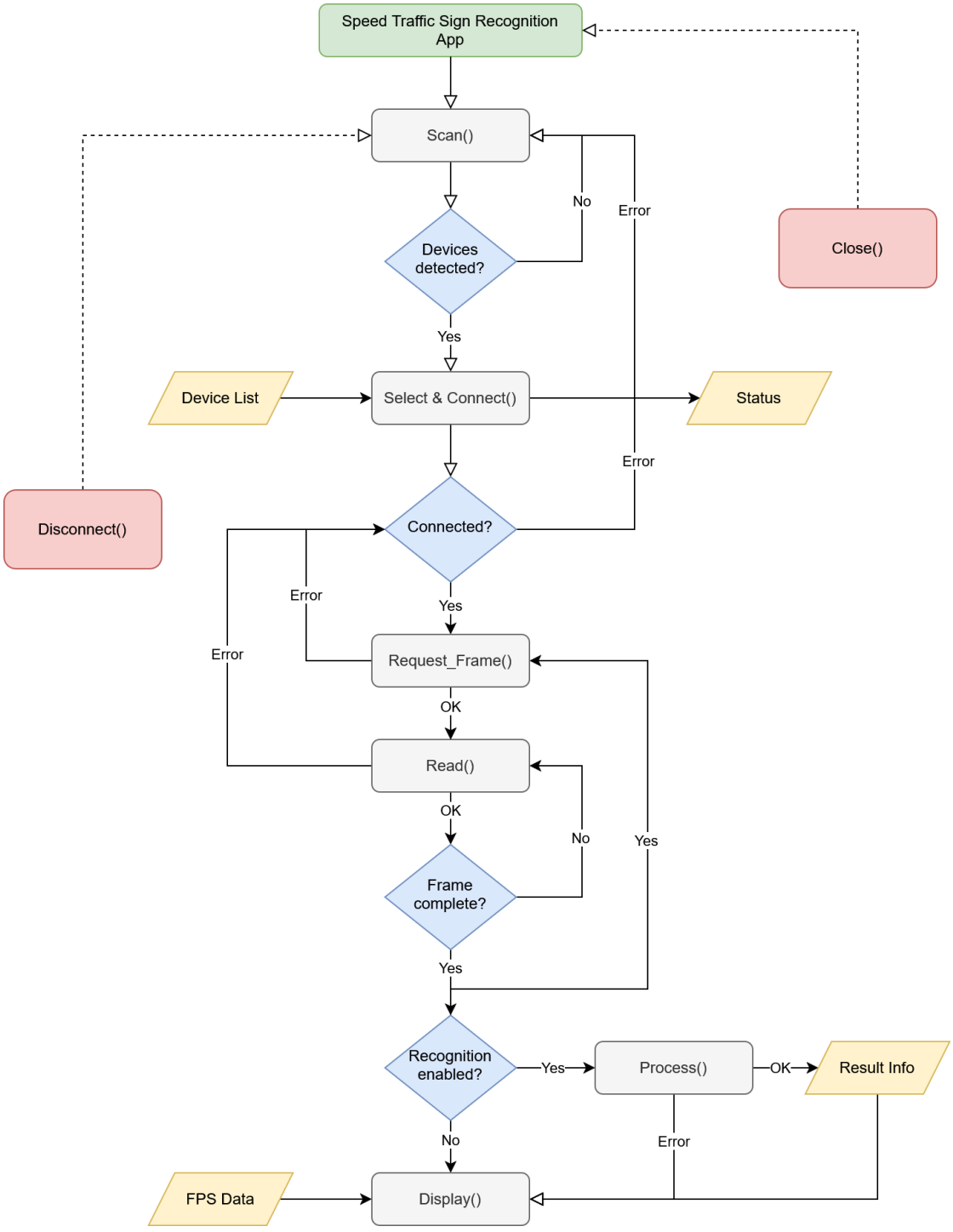


Figure 3.25 Complete flow diagram of the application

The flow begins with the initialisation of the app and the scanning or detection of devices connected to the PC. If the hardware module is correctly detected via the FTDI chip (UM232H-B), it appears in the list of devices, allowing it to be selected and connected to.

Once the connection is established, the system reads the bytes received from the FPGA from one of its threads. When a frame is complete, it checks whether the user has enabled the recognition of traffic signals related to speed limits. If so, it proceeds to process it from another thread (**multithread**), passing the image through the trained neural model. If this fails because no sign is recognised, there is no sign, or recognition has not been enabled, the original image received is displayed and no processing results are shown. If, on the other hand, a sign is detected, the information corresponding to the recognised sign is displayed together with the edited image locating the detected sign. In any case, as long as the frame reading process goes well, an indicator of the frame rate per second received will be displayed.

At any point during the execution flow, the user can press the disconnect button (`disconnect`), which forces a return to the scanning point for devices connected to the PC via serial ports. You could also simply close the application (`close`).

The diagram also shows alternative exit points for connection, reading, or processing errors, allowing for robust execution in the event of failures.

3.2.3 Class-Based Architecture

Each of the actions described is carried out by a different class or module: **Mediator**, **Controller**, **Image Processor**, and **View** (GUI).

Mediator

This class acts as the central coordinator of the application. Its purpose is to manage requests from the graphical interface (***View***), the controller (***Controller***) and the image processor (***Image Processor***). Thanks to this approach, each component can operate independently, facilitating software maintenance and scalability.

When instantiating the ***Mediator*** class, the objects corresponding to the controller, processor and view are passed to it. This mediator class receives connection, disconnection, write and read events from the controller. It notifies both the view, to update the interface at all times, and the processor, to process the data read. On the view side, it receives user events and acts on them with the other modules. It only coordinates calls between the different classes themselves.

The use of the **Model-View-Controller (MVC)** design pattern combined with the **Mediator** pattern allows for the centralisation of communication logic between components and reduces coupling, improving software modularity. This approach

facilitates future extensions, such as replacing the controller with another data source or even easily changing the view or GUI.

View

The **View** class is responsible for the application's graphical user interface (GUI). It displays visual information, receives user events, and updates graphical elements or widgets (labels, buttons, images, etc.). It inherits from *QMainWindow* and the form generated by Qt Designer (*Ui_MainWindow*) and uses PySide6 as its graphical framework. It communicates with the Mediator only through callbacks that are registered with it (**`setConnectionCallback`**, **`setCloseCallback`**, etc.). It uses signals and slots to update the GUI from other components [9]. Its flowchart is shown below:

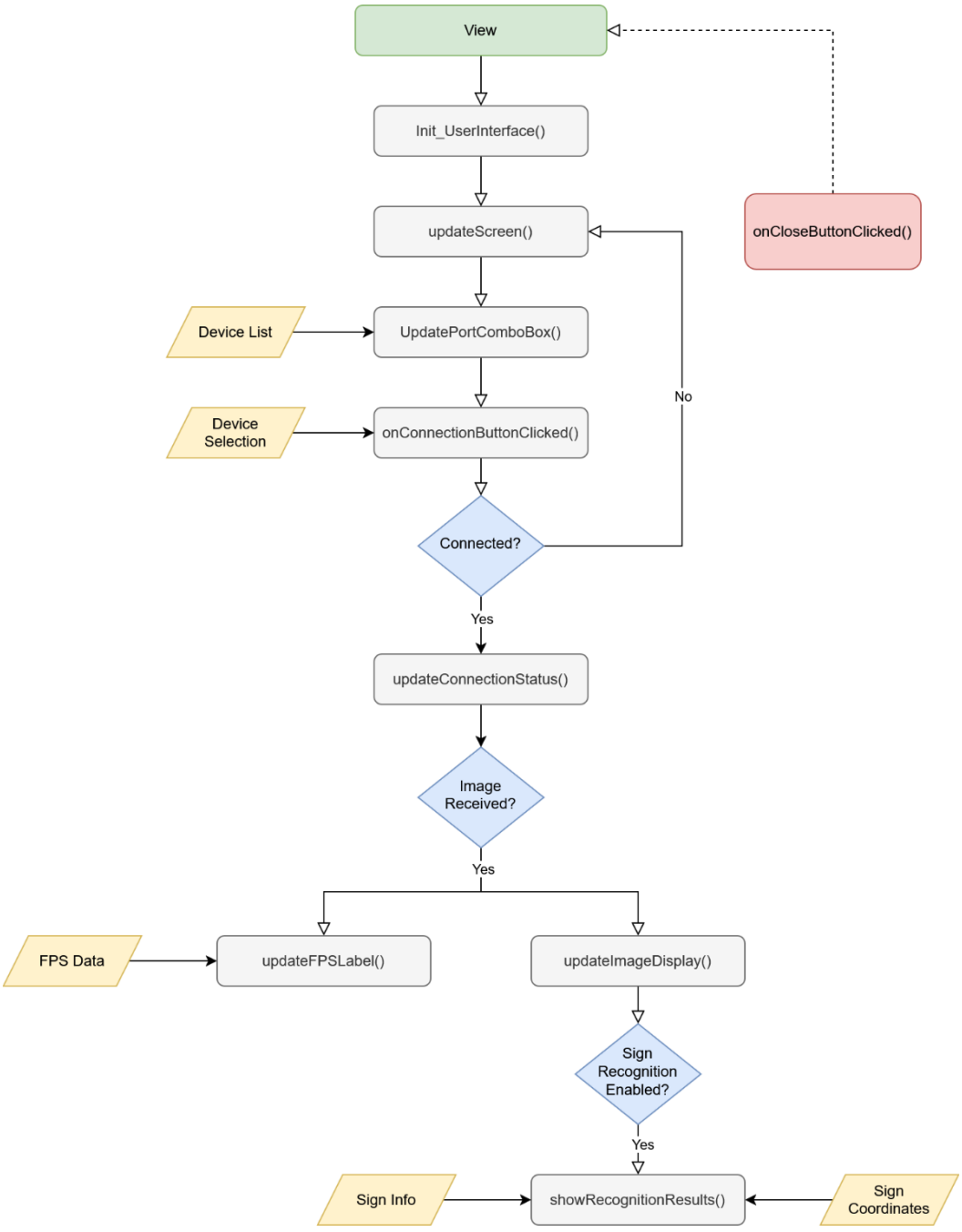


Figure 3.26 Flow diagram of the View class

First, in the class constructor, the GUI is initialized (**`Init_UserInterface`**). Here, the interface is loaded, which is the ***.ui*** file generated from the following command in the terminal:

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

Where "***userInterface_release.py"*** is the name of the file generated through the Qt designer.

This is done by calling **`setupUi`**, which is a method automatically generated by **Qt Designer** when you export the interface to a ***.py*** file. Basically, this function creates and places all the widgets defined in the ***.ui*** file within the current object, which in this case is a subclass of **`QMainWindow`**.

The ***"_release.py"*** interface will be used for the user view. However, another interface, ***"_debug.py"***, has been designed as a developer view to enable various tests to be performed on the system. To load one interface or the other, simply act on the **_DEBUG_COMPILATION** tag**:**

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

This demonstrates the power of modular software, allowing any GUI to be easily loaded.

Next, the interface style is applied, i.e., the font for the text, the shape of the box where the image will go, the app window is centered, among other things. Buttons and widgets are also registered and connected to internal functions or external callbacks.

Once the interface has been initialized, the *comboBox* widget is updated with the list of detected devices indicated by the Mediator, which in turn has been indicated by the controller. If the user presses the **`connect`** button, activating the **`onConnectionButtonClicked`** event, the connection is established through a call to the Mediator from a callback. If the connection is successful, the interface is updated, showing the connection status through **`updateConnectionStatus`**. Here, the interface remains on standby and once an image is received by the Mediator, it will pass it to this class through the **`updateImageDisplay`** along with the FPS value to display the information in a label widget. If the user has enabled the checkbox for sign recognition, a rectangle will also be drawn on the image at the coordinates where the sign has been detected and if it has been detected (Sign Coordinates), along with information about it just to the right. A label will indicate the speed in km/h, the confidence of the recognition as a percentage, and whether or not a sign has been recognized (Sign Info). By default, if enabled, the information for the last recognized sign will always be displayed.

Controller

The "**FTDIController"** class is the component responsible for managing all communication between the application and the FTDI chip (UM232H-B). Its function is to abstract the low-level logic necessary for scanning, connecting, reading, and writing data through the PC's USB port, providing a clean and transparent interface for the mediator.

This class depends on the ***ftd2xx*** library, which allows direct access to FTDI devices through its ***D2XX*** driver, which must be installed on the PC [10]. It also relies on ***serial.tools.list_ports*** to detect available ports and on ***PySide6.QtCore*** for signal (**Signals**) and timer (**QTimer**) management.

These dependencies are necessary to perform the actions discussed in the following flowchart for this class:

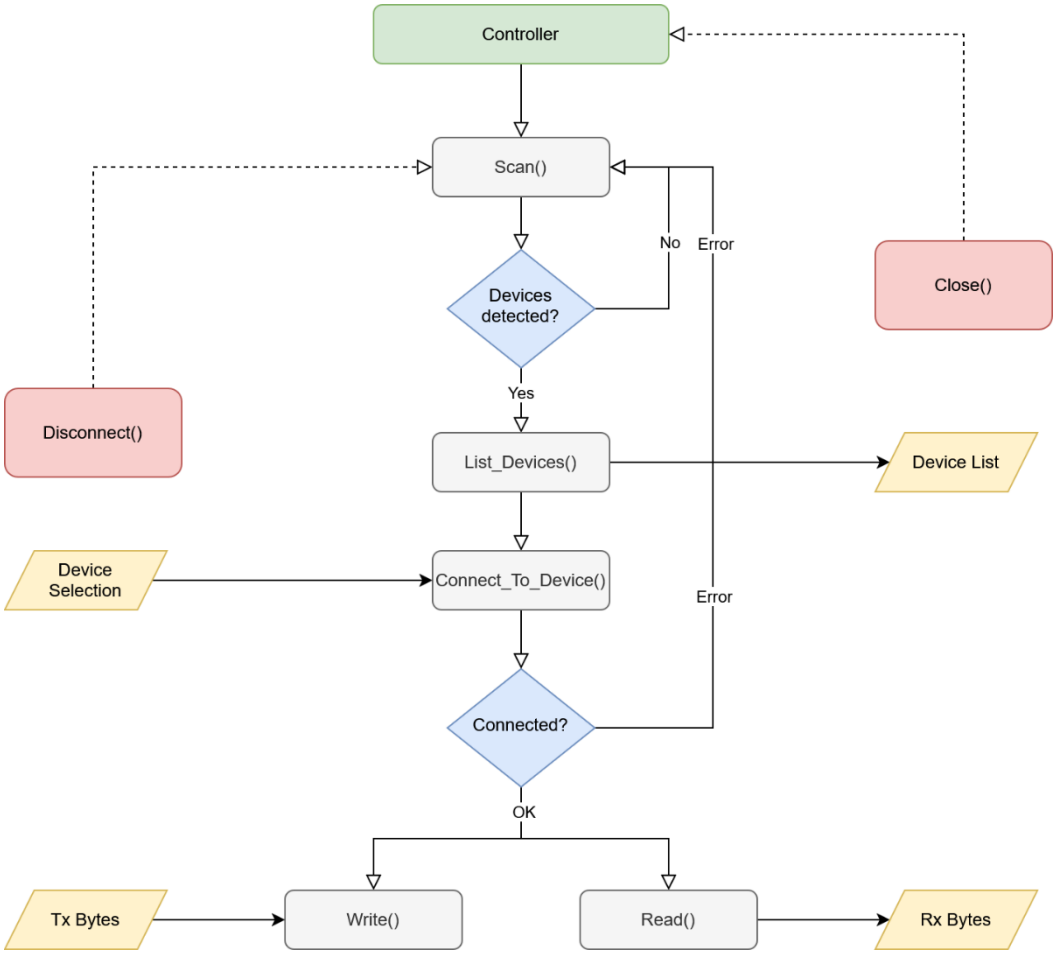


Figure 3.27 Controller class flowchart

Once the corresponding “controller” object of the **“FTDIController”** class has been created, the PC's serial ports are scanned for devices. This scan is performed at 1-second intervals (a constant parameter of the class that cannot be configured by the user), while the devices found are listed using the ***serial.tools.list_ports*** **and** ***ftd2xx*** libraries to obtain details such as VID, PID, serial number, and description. This continues until the external connection event is received along with the selected port.

Internally, this **`connectToDevice`** function calls the FTDI library functions, opening a connection with the device and configuring various communication parameters, including timeouts and read and write buffer sizes, among others. In this case, it would not be necessary to put it in asynchronous FIFO 245 mode, as this has already been done through its own app, saving it directly to its EEPROM. Once the connection has been established correctly, the controller is available to respond to external write and/or read requests that will arrive from the **Mediator**. The **`Write`** action requires the bytes to be transmitted as input, while the **`Read`** action will return the bytes received. At any time, an external disconnection (**`Disconnect`)** event may occur, causing the flow to resume again, activating periodic scanning.

Image Processor

The **"ImageProcessor"** class is responsible for processing images received from the FPGA, including converting raw data or bytes into images that can be displayed in the GUI, as well as executing a neural model for automatic recognition of traffic signals related to speed limits.

It is implemented as a component capable of working in the background using its own processing thread. This module receives complete frames from the Mediator class and, once they have passed through the model, emits an **`imgReadyToDisplay`** signal (Qt Signal) to the Mediator, which in turn sends it to the View. The latter updates the image to be displayed and the model results. It does not directly access the GUI or the controller and also works on a previously loaded model. The flowchart for this class is shown below:

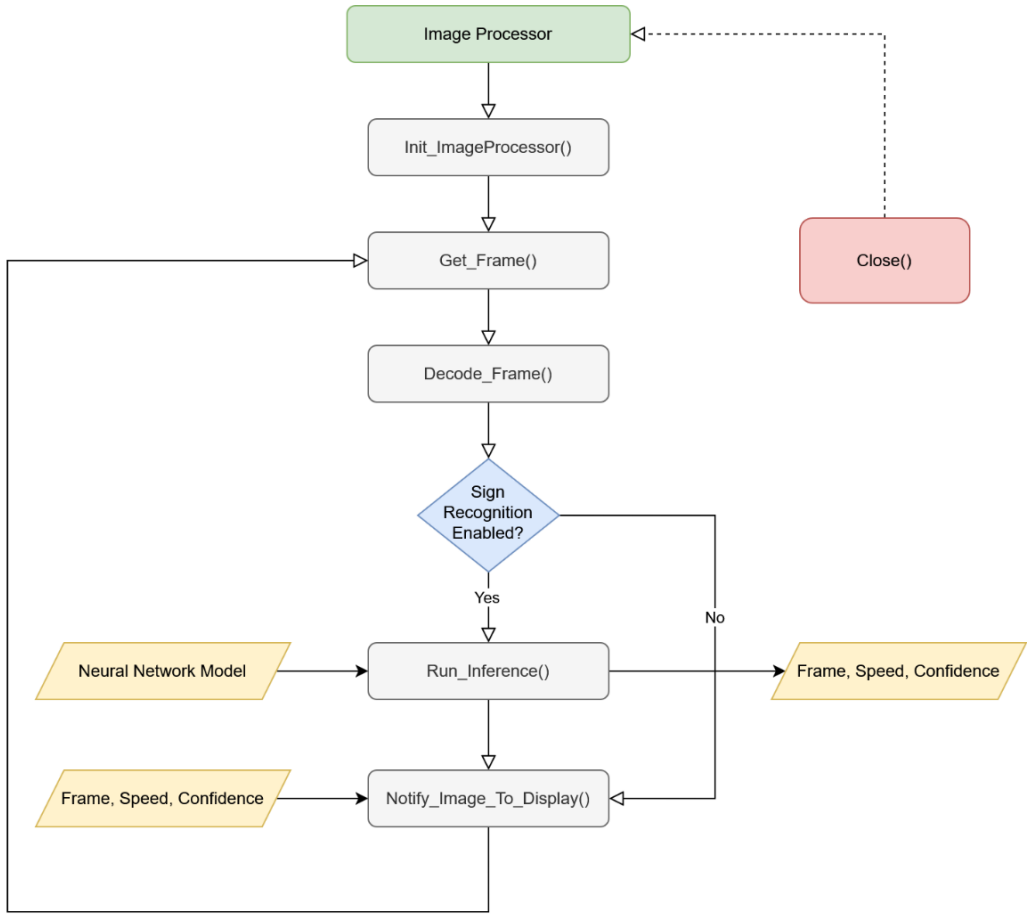


Figure 3.28 Image Processor Class Flowchart

First, when initializing the class, a previously trained YOLO neural model (best.pt) is loaded. This training process will be detailed later.

Once loaded, the **"ProcessorWorker"** thread is started, which, once it receives a complete frame, executes the model inference (**`_runInference`**) on it to detect and

recognize traffic signals. To decouple the image acquisition flow from the inference process, a FIFO queue (queue.Queue) with limited length is used (`FRAME_QUEUE_LEN = 16`). This queue acts as an intermediate buffer, allowing images to be added without blocking acquisition, discarding old frames if processing is slower than expected, avoiding saturations or overflows, and dynamically adapting to inference time thanks to the parameter `MAX_PROCESS_FRAMES_STEP`.

The image data arriving at the queue is encoded in **YUV4:2:2** format, corresponding to the output format of the image sensor (real or emulated) connected to the FPGA. Before passing the image to the inference model, the frame must be correctly decoded and converted to a format compatible with **PyTorch** and **OpenCV** (RGB888 or Grayscale 8 bit).

This decoding process (`Decode_Frame`), depending on the configured frame format, color (`FRAME_COLOR_SIZE`) or grayscale (`FRAME_GRAY_SIZE`), expects a certain number of bytes or another. In this case, **the model is only compatible with images in color format**, as it was trained in this way. The image or frame received is converted to RGB format compatible with **OpenCV**. Subsequently, it is converted to a **QPixmap** object so that it can be displayed in the PySide interface. In addition to this, inference with **YOLOv8** requires, as input parameters, the image to be resized to **512x512**, **`numpy uint8`** matrix format in **RGB** format, a minimum confidence value (in this case 0.85), a threshold below which detections will be discarded. All this is because the images during training had this format [11].

Once the inference is carried out, if the user has enabled recognition, the results are filtered, selecting the detection with the highest confidence value above the established threshold (0.85), which is saved and sent to the GUI.

3.2.4 Neural Model

The functional core of the software module is a neural model trained to detect and recognize traffic signals related to speed limits.

The steps for obtaining it are detailed below:

1. Selection and filtering of the dataset

The initial dataset used is the **GTSRB** (German Traffic Sign Recognition Benchmark), widely known in the field of traffic sign recognition [12]. However, this set includes more than 40 types of signs, which complicates training when the project's objective is to detect only speed limit signs.

Therefore, a process of filtering and remapping classes and converting the dataset is carried out. This reduces the complexity of the model and training time and allows computational resources to be focused on the actual objective of the system.

2. Conversion to YOLO format

Once filtered, the images are reorganized and annotations are generated in *class_mapping.txt*, in the format required by **YOLO**, along with the standard folder structure:

This is done automatically with the script:

python *convertDatasetIntoYOLOformat.py*

The result is a dataset ready to be processed by the **Ultralytics** framework**.**

3. Dataset configuration

Once this is done, the *"yolo_dataset.yaml"* file is generated with the relative paths of the images and the final classes:

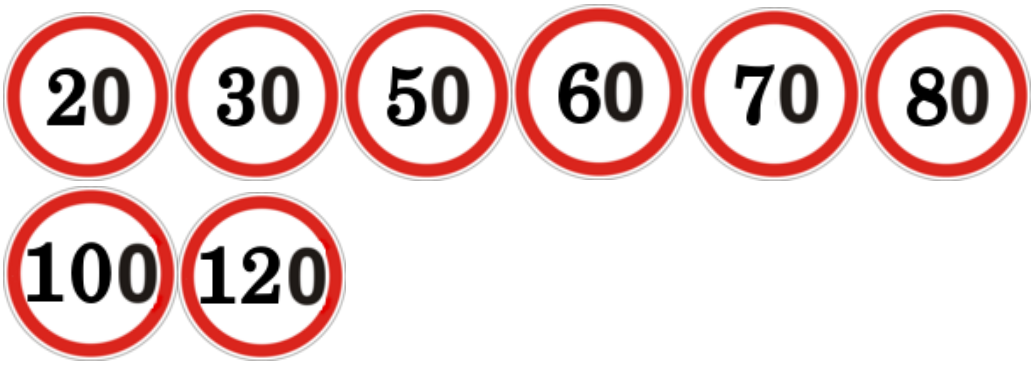
```
train: yolo_dataset_filtered/images/train
val: yolo_dataset_filtered/images/test

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

Figure 3.29 yolo_dataset.yaml file

The invalid class has been included because it corresponds to this signal:

which could mislead the system, so it is decided to add it to the model so that it is able to detect it and differentiate it from the others as well.

Therefore, currently, the classes that the system is able to recognize are the following:



4. Model training

The **YOLOv8n** (nano) version was selected for its balance between accuracy and efficiency for execution on embedded hardware (possible future case).
Training was performed using the "*train_yolo.py" script*.

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

Figure 3.30 Part of the code from the "train_yolo.py" file

The pre-trained model *"yolov8n.pt"* is used. It is trained for 20 **`epochs`**, with an input image size of 512x512 (**`imgsz`**), and **`batch`** 4, i.e., the number of images that are processed simultaneously before updating the model weights during training. This value seeks to strike a balance between accuracy and performance for deployment on systems with limited resources (lightweight models such as YOLOv8n).
This process automatically generates the final "*best.pt*" file of the trained model.

5. Evaluation and results

Once training is complete, the model is automatically evaluated on the validation set (`model.val`). Figure 3.31 shows the training and validation curves for the model.

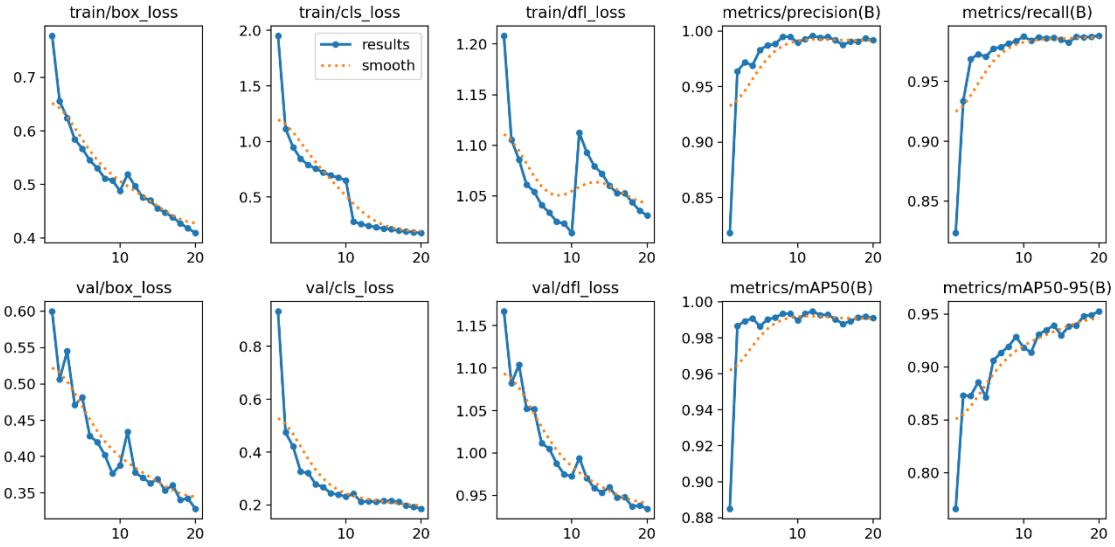


Figure 3.31 Model training results

The ***box_loss***, ***cls_loss***, and ***dfl_loss*** graphs show the evolution of the loss functions associated with the location of bounding boxes, signal classification, and coordinate regression, respectively. In all cases, a progressive decrease in loss is observed as the number of epochs increases, indicating that the model improves its predictive ability with training.

On the other hand, the evaluation metrics (***precision***, ***recall***, and ***mAP***) reflect the quality of the model on the validation set. Precision and recall reach values close to 1, which means that the model generates hardly any false positives or false negatives. Likewise, the ***mAP@50*** and ***mAP@50-95*** metrics, which combine precision and recall at different intersection over union (IoU) thresholds, show an upward trend to very high values (**>0.95**), confirming that the detector generalizes correctly.

Taken together, these curves show that the training was successful: the model converges, losses are reduced, and evaluation metrics reach values that indicate very high performance.

Figure 3.32 shows the normalized confusion matrix of the model evaluated on the validation set.

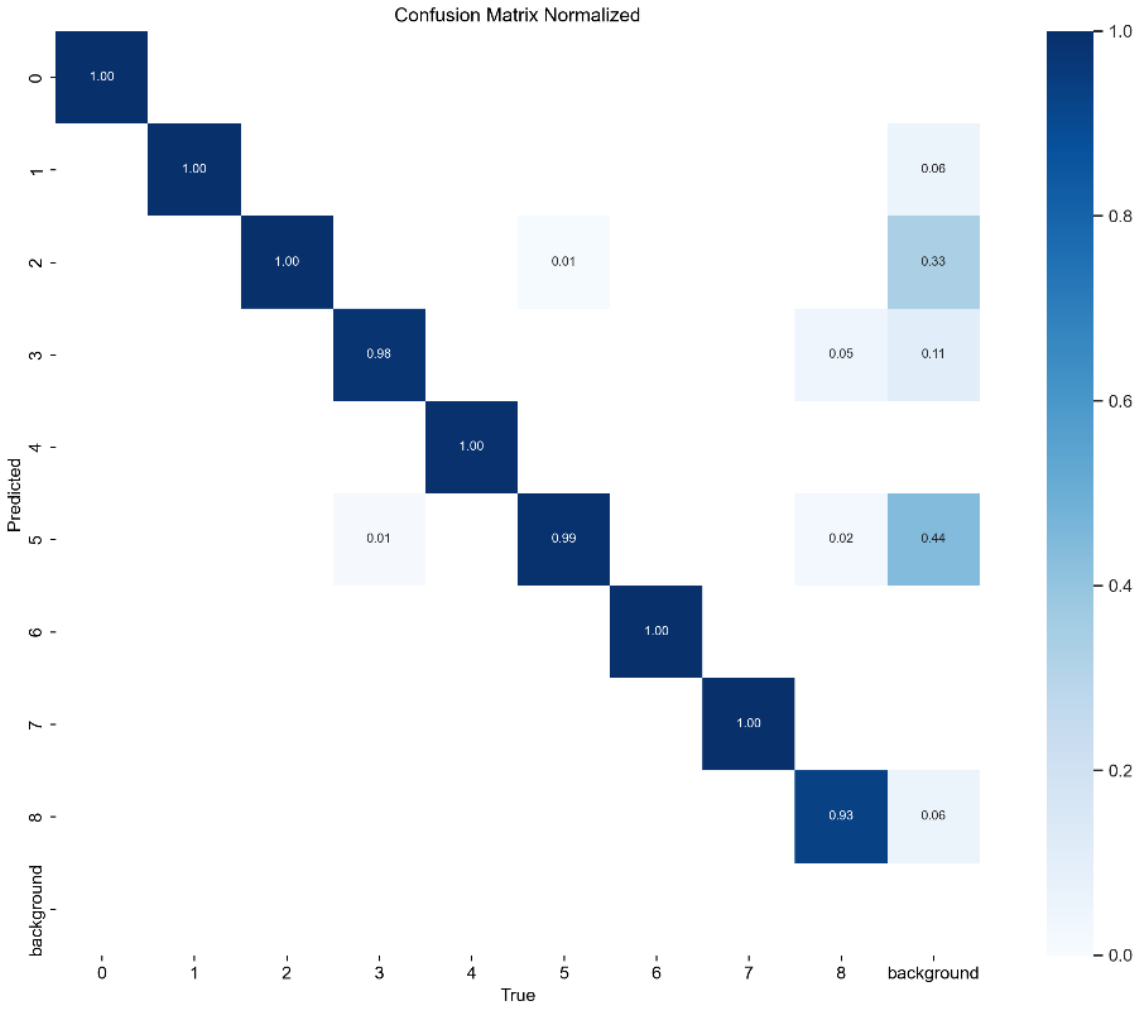


Figure 3.32 Model confusion matrix

The horizontal axis represents the actual classes and the vertical axis represents the classes predicted by the model. The values on the diagonal indicate correct predictions, while the values outside the diagonal represent classification errors.

It can be seen that most of the values on the diagonal are very close to 1, which means that each traffic sign class is identified with great accuracy. The values outside the diagonal are low, although in some cases there is some confusion between signals with similar limits, which is to be expected given their visual similarity.

These results confirm that the model not only has high overall metrics, but also achieves consistent performance class by class, keeping the error rate very low even in signals with similar characteristics.

You can also view these graphs interactively on the **TensorBoard** website itself, launched with the script [13]:

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

6. Model format

In this project, we work directly with the ***"best.pt"*** file, i.e., in **PyTorch** format.

For other applications, it may be necessary to export the model to another existing format:

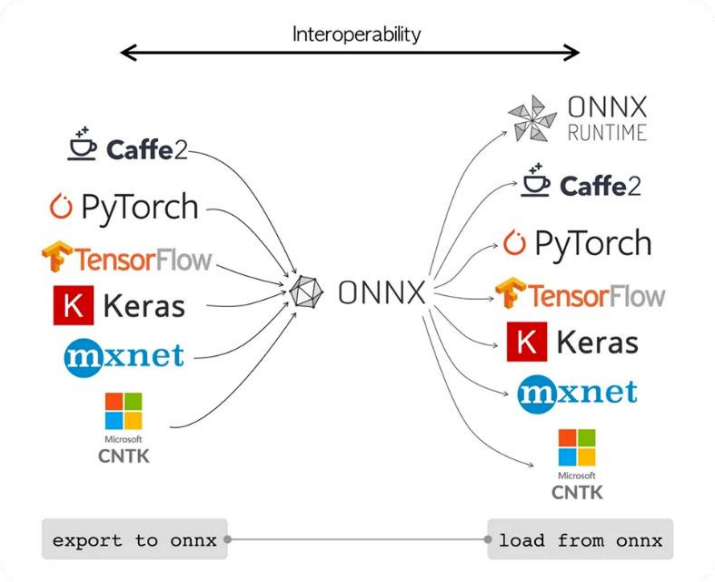


Figure 3.33 Types of models

For example, the format that offers the greatest interoperability is *ONNX* (Open Neural Network Exchange), which is obtained with:

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

7. Model architecture

The architecture used in this project corresponds to the YOLOv8n (nano) model, a lightweight version optimized for real-time detection tasks with limited computational resources. Its design follows a hierarchical and modular approach, dividing the data flow into different functional blocks. The following figure represents the computational graph of the model loaded from *"best.pt"* [14]:

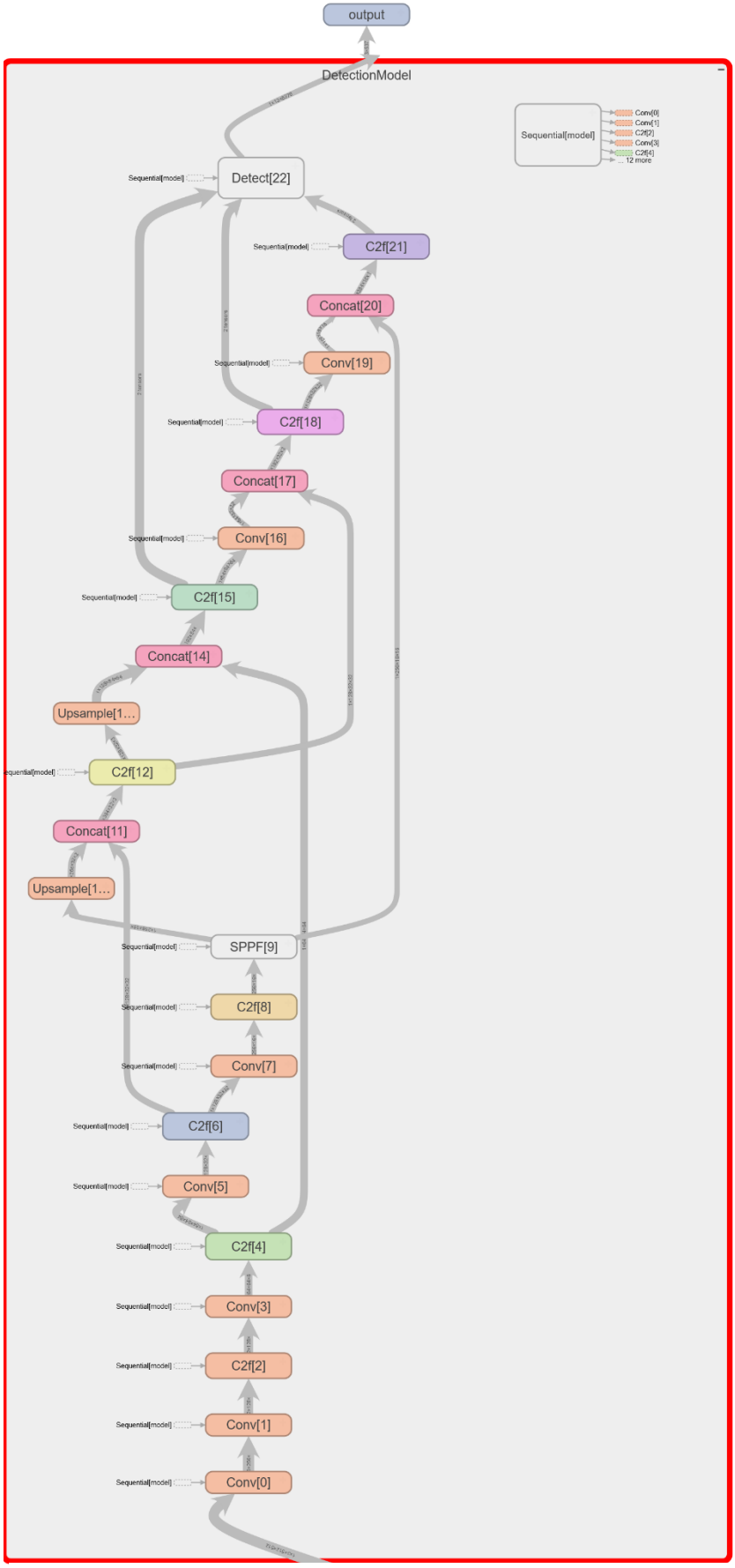


Figure 3.34 Neural network structure

1. **General structure**
   - **Input**
     The input image (normalized and resized to 512x512) is injected into the bottom of the model.
   - **Output**
     The upper block (Detect[22]) generates the following output:
     - Coordinates of the bounding boxes.
     - Detected classes.
     - Confidence values.

2. **Main functional blocks**

- **Conv (Convolutions)**
  They extract basic spatial features such as edges, corners, or textures.
- **C2f (*Cross Stage Partial*)**
  They improve training efficiency by splitting and merging information flows.
- **Concat (Concatenation)**
  They merge information from different stages of the model to improve detection capabilities, especially for small objects.
- ***Upsample* (Resolution increase)**
  They allow high-resolution information lost during downsampling to be recovered, which is vital for detecting small objects.
- **SPPF (*Spatial Pyramid Pooling-Fast*)**
  Combines information from multiple spatial scales to provide global context. Improves the robustness of the model in the face of variations in size and location.
- ***Detect***
  This is the final detection block (Detect[22]). It combines all the extracted features and generates the final predictions (boxes, class, confidence).

3. **Advantages of hierarchical design**
   This type of architecture with multiple parallel resolutions allows small objects (with high-resolution layers) and large objects (with deeper, summarized layers) to be detected, as well as minimizing information loss thanks to ***Concat-type*** connections**.**

The **Ultralytics** package was used for loading, training, and evaluating the model:

```
pip install ultralytics
```

After an initial testing phase of the model, it was found to work correctly with noisy images, as it had been trained with a dataset (GTSRB) designed for real driving situations. For this reason, it was decided to perform a ***fine-tuning*** process using an additional dataset (custom_dataset) composed of sharper images, in order to improve the model's ability to recognize signs in high-quality visual conditions without degradation [15]. This achieved a balance between robustness against noise and precision in more controlled contexts.

The complete documentation of the model training and ***fine-tuning*** process (*best_finetune.pt*), along with the scripts used, are available in the Software/Neural_Network of the GitHub repository.

Chapter 4 Testing and Validation

4.1 Hardware Module

This chapter covers the functional verification process carried out on the different hardware blocks developed in VHDL, which make up the complete embedded system implemented in the FPGA. The aim is to ensure the correct functioning of each component separately and of the complete system in real and simulated conditions.

To this end, a series of testbenches and TCL simulation scripts have been designed to automate simulation tasks and observe the behaviour of internal signals in different scenarios. These simulations are performed in the ***Xilinx Vivado 2019.1*** environment, specifically with its Vivado Simulator tool (see figure) integrated into the ***Vivado HLx WebPack*** suite [16].

To start a simulation, select the ***"Run Behavioural Simulation"*** option. This performs an RTL-level simulation, verifying the logic and FSM before synthesis:

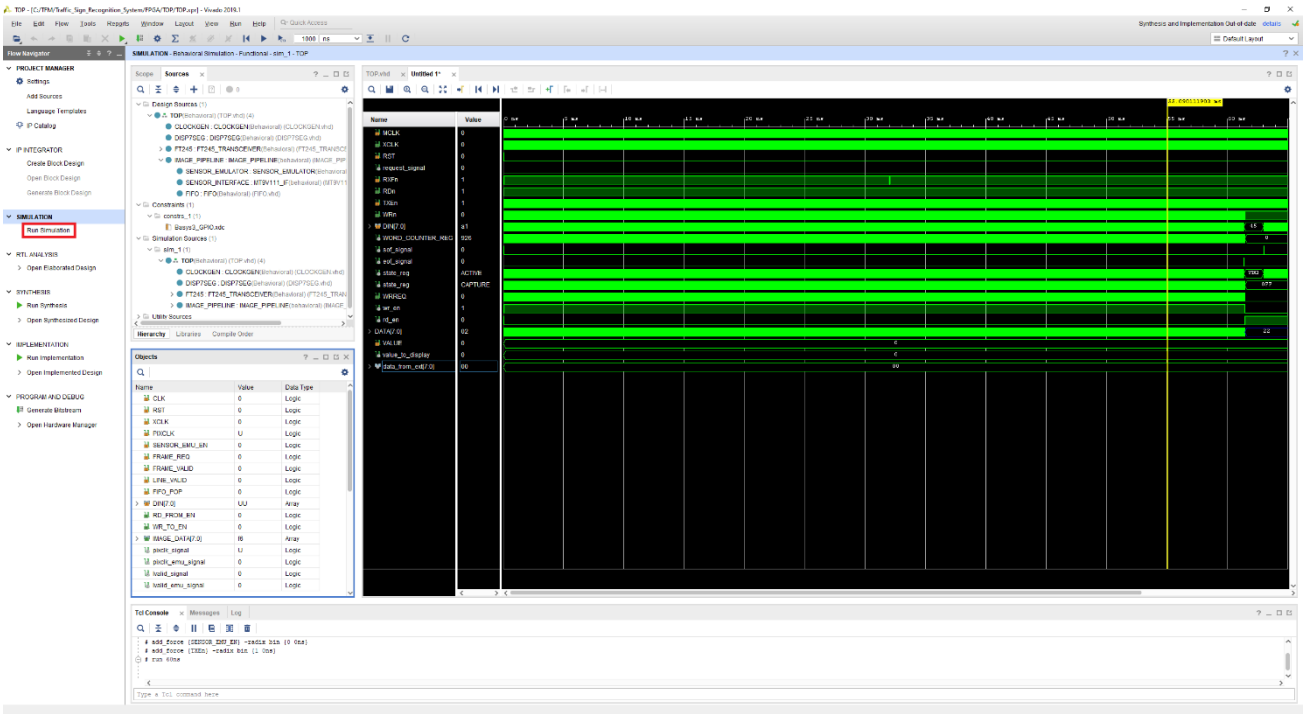


Figure 4.1 View of the simulation in Vivado

The following describes the different windows available in the view that appears when running this type of simulation:

- ***Waveform***
  This is the main signal display window. It shows how signals evolve over time. It allows you to zoom in, measure times between transitions, and detect problems such as glitches or unwanted delays.
- ***Objects***
  Lists all internal and external signals for the TOP module or selected hierarchy. From here, they can be added to the waveform for visualisation.
- ***Scopes***
  Shows the design hierarchy, useful if the design has several levels, as is the case here. It allows you to navigate through the submodules and view their internal signals.
- ***Console* and TCL *console***

  Displays messages, errors, warnings, or logs during simulation. The TCL console allows you to run commands directly or load scripts.

  First, unit tests are performed in blocks, simulating each part of each block separately. Subsequently, each complete block is simulated individually.

  The other simulation options, post-synthesis and post-implementation, are intended to simulate the generated netlist, not the described RTL model.

4.1.1 FT245_IF

This test focuses on validating the correct operation of the **FT245_IF** block. As mentioned in previous sections, this block contains an FSM that controls the signals involved in communication between the FPGA and the PC via the UM232H-B module.

As discussed in the chapter on hardware design and implementation, this block is composed of two components, **IF_WRITE** and **IF_READ**, corresponding to the write and read interfaces, respectively.

First, the simulation showing the behaviour of **IF_WRITE** in an asynchronous write process, which would be a transmission from the FPGA to the PC, is presented. To do this, the *"FT245_WRITE_async_IF_sim.tcl"* script is executed, which automates the compilation of the design, the launch of the simulation and the visualisation of the key signals**.**

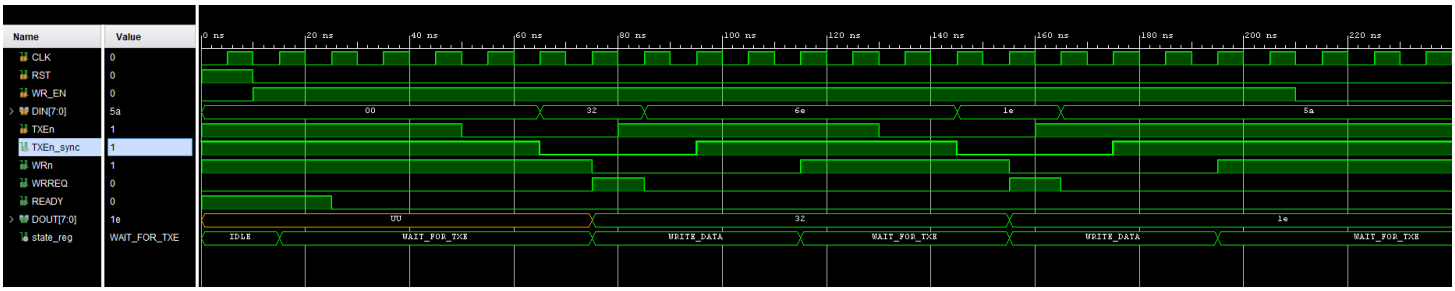


Figure 4.2 IF_WRITE write cycle simulation

By analysing the image, you can see the evolution of the interface control signals (**`WR_EN`**, **`WRn`**, **`WRREQ`**, **`READY`**) in response to different input data (**`DIN`**) and the activation or deactivation of the **`TXEn`** signal (`TXE#` on the FTDI chip), which is key to starting and ending the write cycle. The **FT245** uses this signal to indicate whether or not the internal buffer is ready to receive data. Therefore, we can see that the FPGA remains in the **WAIT_FOR_TXE** state, after moving to this state from **IDLE** when **`WR_EN='1'`**, until the synchronised version of **`TXEn`** (**`TXEn_sync`**) drops to **`'0'`**, starting the write cycle, which is reflected in the transition to the **WRITE_DATA** state and the activation of the **`WRn`** signal at low level. At this point, **`WRREQ`** is activated, a signal that represents a data request to an external component responsible for supplying the information to be transmitted. In addition, the input data, DIN, is seen to leave the write interface (**`DOUT`**) towards the FPGA. It can be seen how the input data 0x6e is lost because it does not meet the timing requirements of the FT245, as the availability of the data does not coincide with the moment when the FT245 indicates that it is ready to transmit by activating **`TXEn_sync`**. However, the next one (`0x1e`) is captured correctly, as it meets these conditions. The active **`READY`** signal simply indicates that the interface is at rest, i.e. in **IDLE** state.

In each cycle, you can see how the interface meets the requirements imposed by FT245:

- ✓ **T6:** `WRn` active after `TXEn` inactive ≥ 1ns (`WRn` goes low one cycle after `TXEn_sync = '0'`).
- ✓ **T10:** Active pulse of `WRn` (duration) ≥ 30ns (`NWCYCLES` = 4 -> 40ns)
- ✓ **T8:** *Setup* of data before `WRn` ↓≥ 5ns (`DIN` already stable when `WRn` is activated).
- ✓ **T9:** Data *hold* time after `WRn` ↓ ≥ 5ns (`DIN` remains at the interface input until after `WRn` ↑).

It has been found that each byte transmitted requires approximately **90ns**. This equates to a theoretical transfer rate of around **11 MB/s** (90ns × 1 byte → 1/90ns ≈ 11,1 × 10^6 bytes/s).

However, the system cannot achieve this speed in practice, as the **FT232H**, configured in **FT245 asynchronous FIFO mode**, has a maximum transfer limitation of **8 MB/s**, as specified in the manufacturer's datasheet. This means that, even though the FPGA logic is capable of generating higher bandwidth, the actual restriction is imposed by the USB device itself, setting the maximum communication performance.

Given this limitation, the effective rate at which images can be transmitted to the PC depends on the size of each frame. In the case of the YUV422 colour format (640 × 480 pixels, 2 bytes per pixel), each image occupies **614,400 bytes**, while in the monochrome format (1 byte per pixel) the image occupies **307,200 bytes**. Dividing the maximum bandwidth (8 MB/s) between these frame sizes gives an approximate upper limit of **13FPS in colour** and **26FPS in greyscale**. As will be seen later, these values coincide with the measurements obtained in actual tests.

Below is a simulation showing the behaviour of **IF_READ** in an asynchronous reading process, which would be a transmission from the PC to the FPGA. To do this, the "*FT245_READ_async_IF_sim.tcl*" script is run, which automates the compilation of the design, the launch of the simulation and the visualisation of the key signals**.**

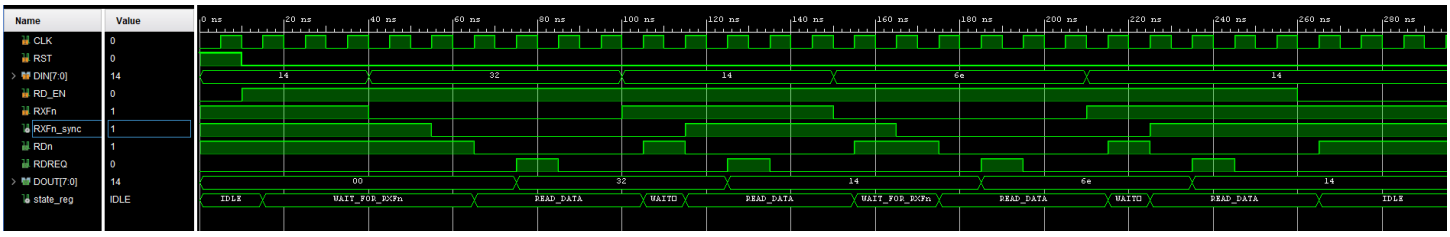


Figure 4.3 IF_READ read cycle simulation

Here, the behavior of the control signals (**`RDn`**, **`RDREQ`** is shown as the interface's response to the **`RXFn`** signal, which indicates data availability in the **FT245** buffer **(`RXFn`**, which is `RXF#` on the FTDI chip). The read operation flow begins with the interface in **IDLE** state until reading is enabled, i.e., **`RD_EN = '1'`**. At that point, it transitions to

the **WAIT_FOR_RXFn** state, where it remains until the synchronized version of **RXFn** (**RXFn_sync**) is activated at a low level. This simulates the FT245's notification that data is available for the interface. At this point, it switches to the **READ_DATA** state, activating **RDn** at a low level, thus initiating the process of reading the data at the interface input (**DIN**). At the same time, **RDREQ** is activated at a high level, which is an output signal from this component indicating that the **DOUT** data is available to be consumed by another external block. The read cycle lasts NRCYCLES = 4 CLK clock cycles, as can be verified by measuring the time the interface FSM remains in the **READ_DATA** state (time between **RDn** activation and deactivation). If reading is still enabled, as is the case (**RD_EN**), it returns to the wait state for the **RXFn_sync** read start signal. From what we can see, four read cycles have been performed in the simulation, where the second and fourth cycles spend less time waiting for the activation of **RXFn_sync**, since when it returns to this state it is already active.

After this last read cycle, it is finally disabled, returning the FSM to the **IDLE** state.

In each cycle, it can be seen how the interface complies with the FT245 requirements, which are only imposed from the start of reading, not during the wait:

- ✓ **T5**: RDn active after RXFn falling edge RXFn ≥ 0ns
- ✓ **T4**: Active RDn pulse ≥ 30ns (NRCYCLES = 4 -> 40ns)
- ✓ **T1**: RDn inactive before new RXFn (rising edge of RDn and wait)

4.1.2 DISP7SEG

This block is responsible for displaying a 3-digit natural number, i.e., from 0 to 999, on the BASYS 3's 4-digit 7-segment display. Internally, it divides the value into hundreds, tens, and units, activating them one by one cyclically while turning off the rest using the **AN[3:0]** signals, giving the impression that they are displayed simultaneously.

This can be seen in the simulation performed by running the *DISP7SEG_sim.tcl* script, which, when the input is **VALUE = 213**, breaks down this value, reflecting the result in the signal **digits[2:0] = "2, 1, 3"**.

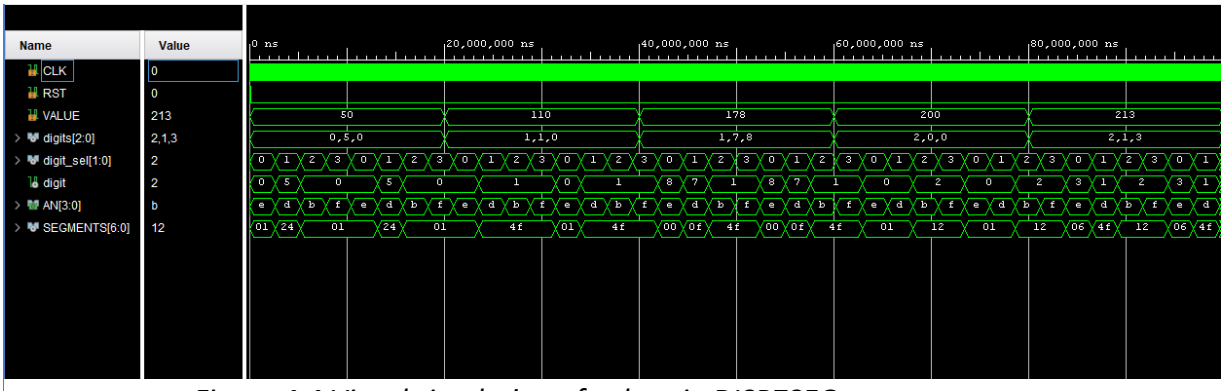


Figure 4.4 Visual simulation of values in DISP7SEG

The **digit_sel** signal rotates between **00**, **01** and **10**, thus selecting the active digit in each refresh cycle. The **digit** signal, meanwhile, contains the number to be displayed at that moment (i.e., 3, 1, 2 in order).

As mentioned above, **AN[3:0]** controls the digit lighting at any given moment. Since it is a common anode display, it is activated at a low level. Therefore, you can see how **AN** rotates between **1011**, **1101** and **1110**, indicating that the block is rotating the lighting of digits 0, 1, and 2. **The fourth has no use in this application** (it remains off at a high level, `'1'`).

Finally, **SEGMENTS[7:0]** represents the 7 segments that form the number (not including the decimal point, `DP`). ach hexadecimal value represents a combination of active segments, so you can see how the value corresponding to the current digit changes (for example, `0x4f` corresponds to the active segments for displaying the "3").

4.1.3 CLOCKGEN

The purpose of this module is to generate a lower frequency **CLKGEN** clock signal from a **CLKREF** reference signal. This is done using a counter controlled by the **NCYCLES** parameter.

Below is a screenshot of the simulation obtained after running the script:

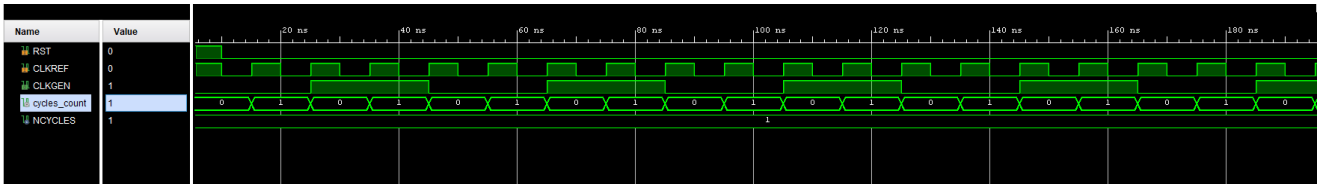


Figure 4.5 Simulation of 25MHz clock generation with CLOCKGEN

This image shows the input to the block, **CLKREF**, which is the clock signal used as a reference. In this case, **NCYCLES = 1** is defined, which is the maximum value that the **cycles_count** counter will reach. This counter counts each rising edge of **CLKREF** until it reaches **NCYCLES**, at which point it resets. This counter causes **CLKGEN** to alternate between `'0'` and `'1'` every two **CLKREF** cycles, thus producing a signal with ¼ of the original frequency. In this case, the input is that of the **100MHz** system, and therefore the generated signal is **25MHz**.

Finally, a **12.5MHz** signal is generated for the project, which means that **NCYCLES = 3**.

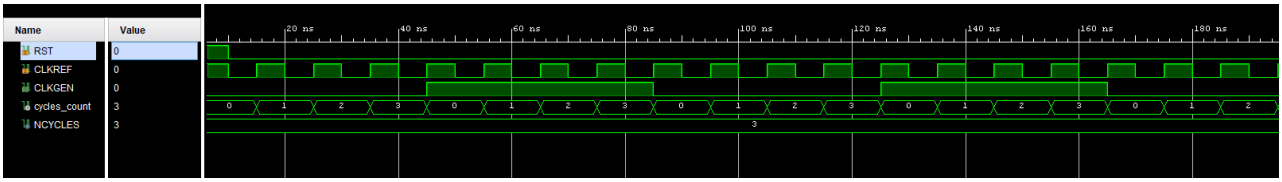


Figure 4.6 *Simulation of 12.5MHz clock generation with CLOCKGEN*

This frequency has been selected because it is necessary to ensure the transmission of complete frames in **colour format**, an essential requirement given that the trained **neural model works exclusively with colour images**. As will be detailed in later sections, the use of greyscale images is not compatible with the detection model developed.

4.1.4 IMAGE_PIPELINE

This block constitutes the intermediate processing core between the image sensor interface and the modules that write or process the data. Its main function is to capture, temporarily store, and transfer digitised image data in a secure and synchronised manner.

FIFO

One of the essential components of this subsystem is the FIFO (First In, First Out) memory, which is responsible for storing the captured image bytes to be consumed by the following modules (such as the PC transmission interface or the signal processor). This queue allows two different time domains (capture and transmission) to be decoupled, and correctly manages input and output flows.

This component is a synchronous storage queue, configurable in data width (**`DATA_WIDTH`**), which is the number of bits stored in each position or cell of the FIFO, and address width, which is the number of bits needed to address all positions in the FIFO (**`ADDR_WIDTH`**). This means that if, for example, `ADDR_WIDTH = 2`, you will have a FIFO that can store up to $2^2 = 4$ data positions (depth). Knowing this, the operation of the FIFO is simulated using a script, obtaining the following result:

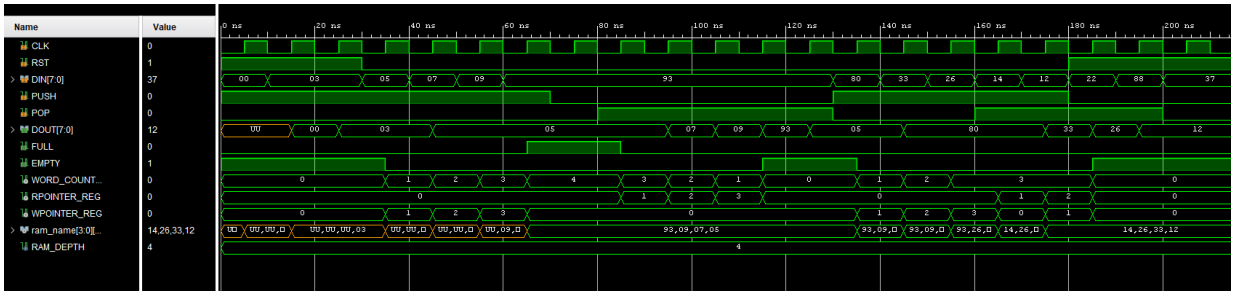


Figure 4.7 FIFO operation simulation

Here, you can see how the **`DIN`** values are written: **`0x03, 0x05, 0x07`** and **`0x09`** between 0 and 90ns when **`PUSH = '1'`** is activated. In turn, the **`WORD_COUNTER_REG`** counter increases to 4, indicating the number of words or values stored in the FIFO at each instant. The **`FULL`** signal is activated when maximum capacity is reached (**`RAM_DEPTH = 4`**). The data write pointer reflects the position of the last saved data, rotating cyclically between 0 and 3 in this simulated example. You can see how the content of **`ram_name`** is correctly updated with the new data that is being written or saved.

It can also be seen how this FIFO is designed so that the DOUT output data always reflects the value stored in the current position of the read pointer, even if the POP = '1' signal is not activated. This behaviour allows the next available data to be observed without consuming it, avoiding dead cycles and improving efficiency and synchronisation between the image capture block and the block that consumes them.

After 100 ns of simulation, the reading process begins by activating the **POP = '1'** signal. In each read cycle, the value of **WORD_COUNTER_REG** is reduced as values are retrieved from the **FIFO**, the **RPOINTER_REG** pointer is rotated, and **DOUT** is updated with the content read in order (0x03 → 0x05 → 0x07 → …). The **EMPTY** signal is activated when the last data from the FIFO is consumed.

At 160 ns, several writes and reads are performed simultaneously in the same clock cycle. The word counter remains constant (WORD_COUNTER_REG) as one piece of data enters at the same time as another leaves. The write and read pointers are both updated in the same cycle. The memory (ram_name) correctly updates the FIFO status.

This behaviour is expected in the synchronised FIFO design. The essential condition of this design is that **PUSH** must not occur when FULL = '1' and POP must not occur when EMPTY = '1', and this is also verified in the simulation result.

SENSOR_EMU

This block implements the real MT9V111 image sensor emulator. This allows the entire capture process to be validated without the need for the physical sensor.

To do this, a simulation script is run, showing how the emulator generates the **PIXCLK** clock signal, which is the same as the one present at its input, **XCLK**. It also generates the **FRAME_VALID** and **LINE_VALID** control signals present in the real sensor. **IMAGE_DATA[7:0]** contains a synthetic image generated in the component itself, which is a simple gradient pattern, so that it is easily recognisable when received and displayed in the PC software. This entire block is controlled by a state machine.

In this first figure, you can see how the sensor emulator starts transmitting the first frame, activating the **`FRAME_VALID`** signal. After the **SOFBLANKING** period, defined in the MT9V111 datasheet, it activates the transmission signal of its first row or line of the image. This means that the image data is now valid, which is why the status changes from **SOFBLANKING** to **ACTIVE**.

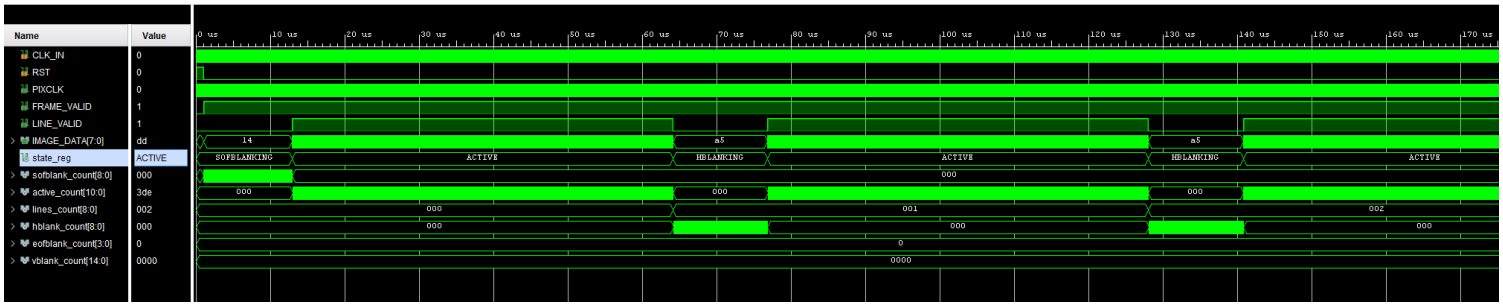


Figure 4.8 Image capture simulation with SENSOR_EMU (part 1).

In this second capture, you can see how the component's FSM switches continuously from the **ACTIVE** state to **HBLANKING**. This is due to the **BLANKING** periods between valid lines of the active image. This period is counted with its **`hblank_count`** counter, and during this period the **`LINE_VALID`** signal is deactivated.

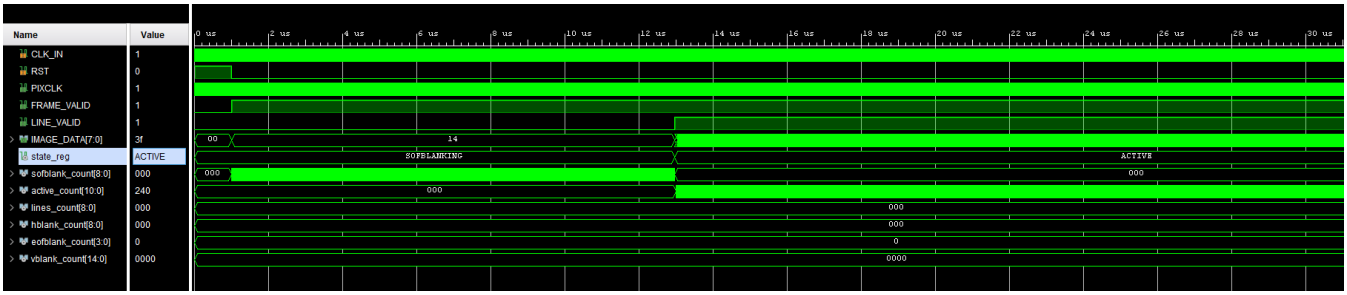


Figure 4.9 Image capture simulation with SENSOR_EMU (part 2).

These blanking periods generally allow consumers or image receivers to know where lines and frames begin and end, i.e., to synchronise with the sensor.

Finally, when the counter of valid lines transmitted reaches the image resolution value (**`NUM_LINES = 480`**), the image transmission is complete and the system moves to the end-of-field blanking state, **EOFBLANKING**. When this state is completed, the current frame is terminated and this is indicated by the **`FRAME_VALID`** signal, which was active throughout the transmission.

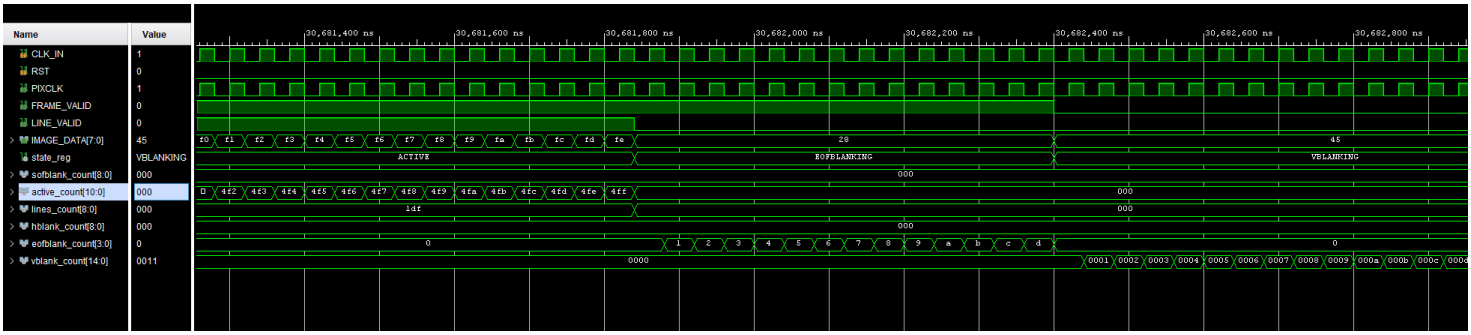


Figure 4.10 Image capture simulation with SENSOR_EMU (part 3).

Now, it moves on to another blanking period, **VBLANKING**, which is a pause between frames to ensure their synchronisation with the receiver.

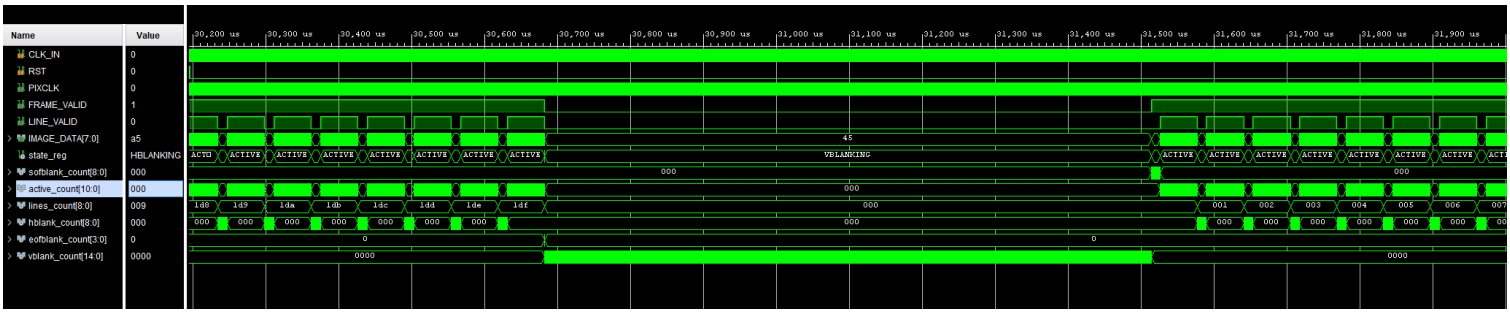


Figure 4.11 Image capture simulation with SENSOR_EMU (part 4).

Once this period has ended, transmission of a new frame begins, starting again with the start blanking period (**SOFBLANKING**). As can be seen in the images, each period or state of the FSM has its own counters, which are defined with values taken from the datasheet, as indicated in section 3.1.4 Image Sensor.

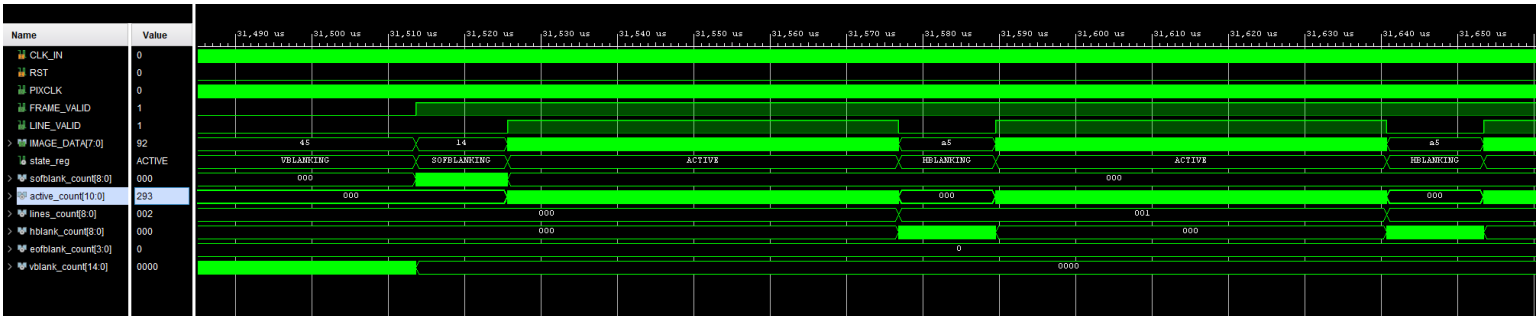


Figure 4.12 Image capture simulation with SENSOR_EMU (part 5).

MT9V111_IF

This block implements a state machine or FSM similar to that of the sensor emulator, as it is the component that interfaces with it and with the actual sensor. In addition to the different **BLANKING** states, it has an additional **FLUSH** state, which is reached when the FIFO is full, i.e., `FIFO_FULL = '1'` or when the reception of a frame has been completed. In this state, the FIFO is emptied, thus ensuring that no data is lost.

4.1.5 TOP

With this in mind, the entire system is simulated. First, the capture of an image in colour format at 25MHz, the maximum configurable frequency in this system, is emulated. All this is done by modifying the capture process of the ***.vhdl*** file of the sensor interface (MT9V111_IF), as well as the input clock frequency to the **`XCLK`** emulator. Once the simulation script is executed, the following is obtained:

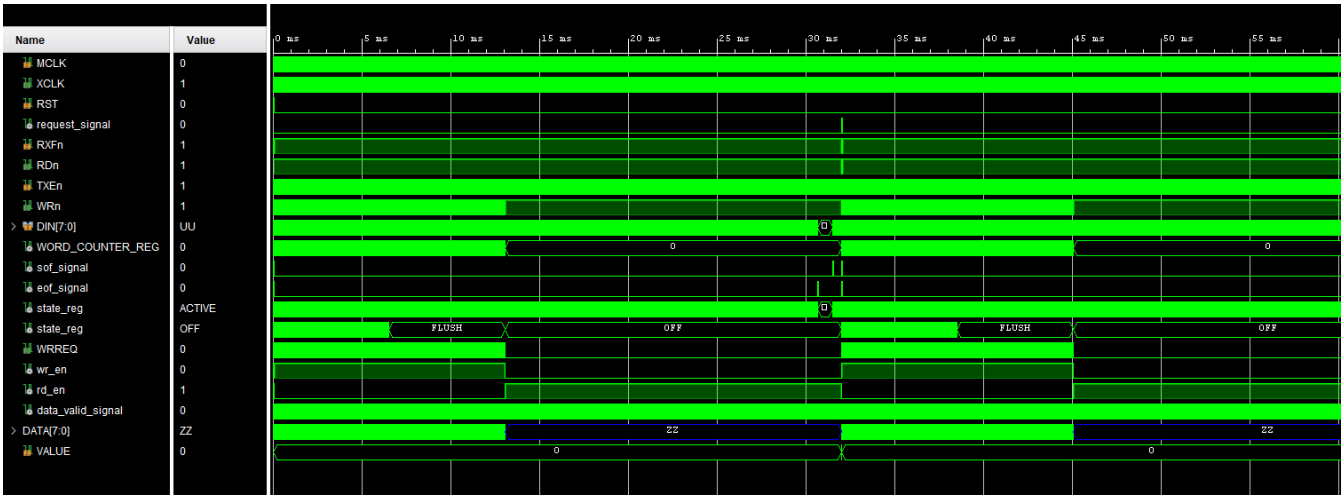


Figure 4.13 Full system simulation (TOP) at 25MHz and colour frame

Here, you can see how, with this data width and frequency, the system is unable to capture the entire frame and store it, causing the reception interface to enter **FLUSH** status before the frame is complete (indicated by the activation of **`eof_signal`**).

As a test, it is decided to maintain the frequency but reduce the amount of data to be captured, i.e., capturing the image in greyscale instead of colour.

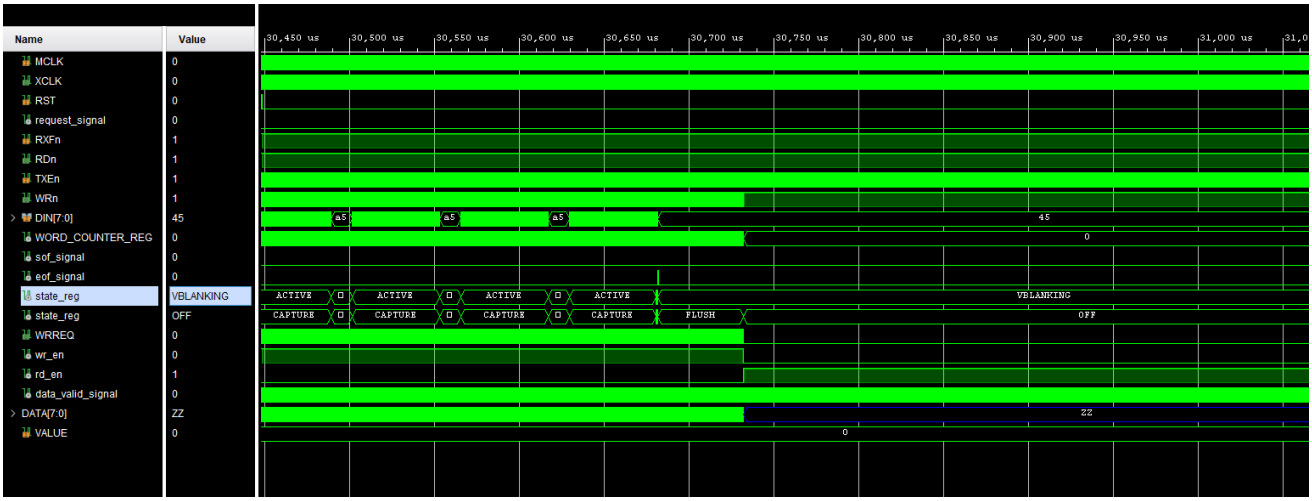


Figure 4.14 Complete system simulation (TOP) at 25MHz and grayscale frame

Now we can see how the system is capable of capturing a complete frame without becoming saturated.

However, this is not useful to us, as the neural model trained and used for this project was trained with colour images, so it will not interpret images in this format. It was therefore decided to leave it in colour format (YUV4:2:2), but to reduce the frequency by half, i.e. to **12.5MHz**. Changing this gives the same result, and we can see that the system is now able to capture the frame.

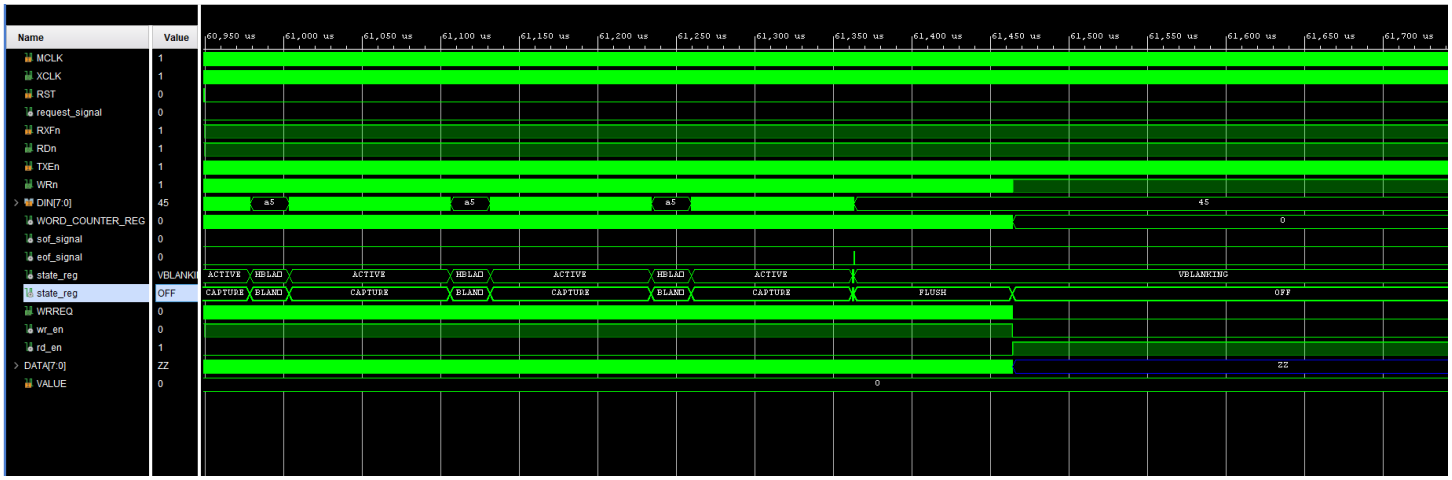


Figure 4.15 Full system simulation (TOP) at 12.5MHz and colour frame

Calculating the simulation and comparing it with the sensor datasheet, we can see that the frame time will be approximately 65ms. Therefore, in an ideal case, the system will have a maximum rate of 15FPS. In greyscale, it is even higher, since the frame time at 25MHz is 32ms, as seen in Figure 4.14, which translates to an ideal rate of 32FPS.

Only when the sensor interface block has finished capturing the frame and is at rest is reading from the PC permitted (**`rd_en`** signal). In this image of a grayscale capture simulation (frame time 32ms), an asynchronous transmission from the PC to the FPGA has been simulated, once the capture and transmission of the frame to the PC has been completed and the FSM of the sensor interface is in IDLE state. The request to send data from the PC is reflected in the **`RXFn`** signal edge at 32ms. The **`VALUE`** signal is what reaches the 7-segment display, i.e. the value 80 (`0x50` in hexadecimal, as seen in the `DATA` buffer) is shown:

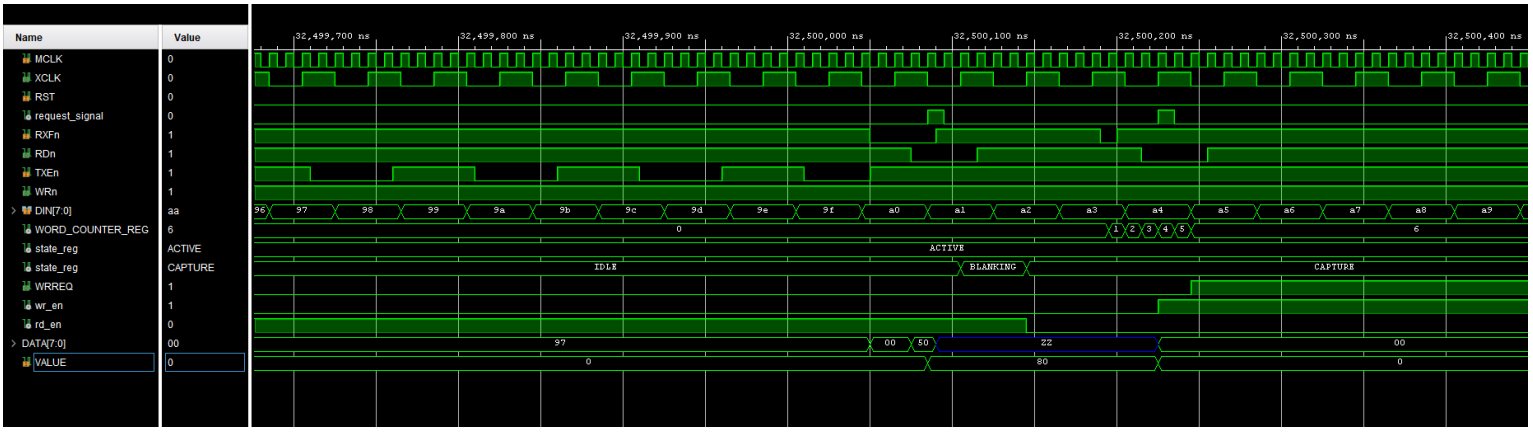


Figure 4.16 Complete system simulation (TOP) upon receiving data from PC

This same data entry is interpreted by the system as a new frame request. Thus, once the PC processes the current frame, it recognises the speed and sends it to the FPGA for display, while also telling it that it is ready to receive the next frame.

For more information about the simulations of the different hardware modules, as well as the location of the simulation scripts corresponding to each block, please refer to the project repository on GitHub, where the complete documentation is included in the README file in the ***Hardware/FPGA_Modules*** folder.

4.2 Software Module

Once the software part had been developed, various tests were carried out to verify its correct functioning together with the hardware part, thus validating the entire system. These tests allow the reliability of the graphical user interface (GUI), communication with the FTDI module and the performance of the neural model for traffic signal recognition to be validated.

4.2.1 User Interface in DEBUG Mode

To illustrate these tests, images were captured in different test cases. Initially, this debug interface (*"SpeedTrafficSignRecognitionApp_debug.ui"),* was used, which is not the one that the user will have, but which helped to validate the different hardware and software blocks separately. From this interface, both single (*"Manual Send Mode"*) and periodic (*"Auto Send Mode"*) transmissions were tested from the PC to the FPGA through the indicated control, verifying that it was received by the FPGA, displaying the data in binary by turning on the BASYS 3 LEDs (**`LD0`** to **`LD7`**), as well as displaying it on the screen.

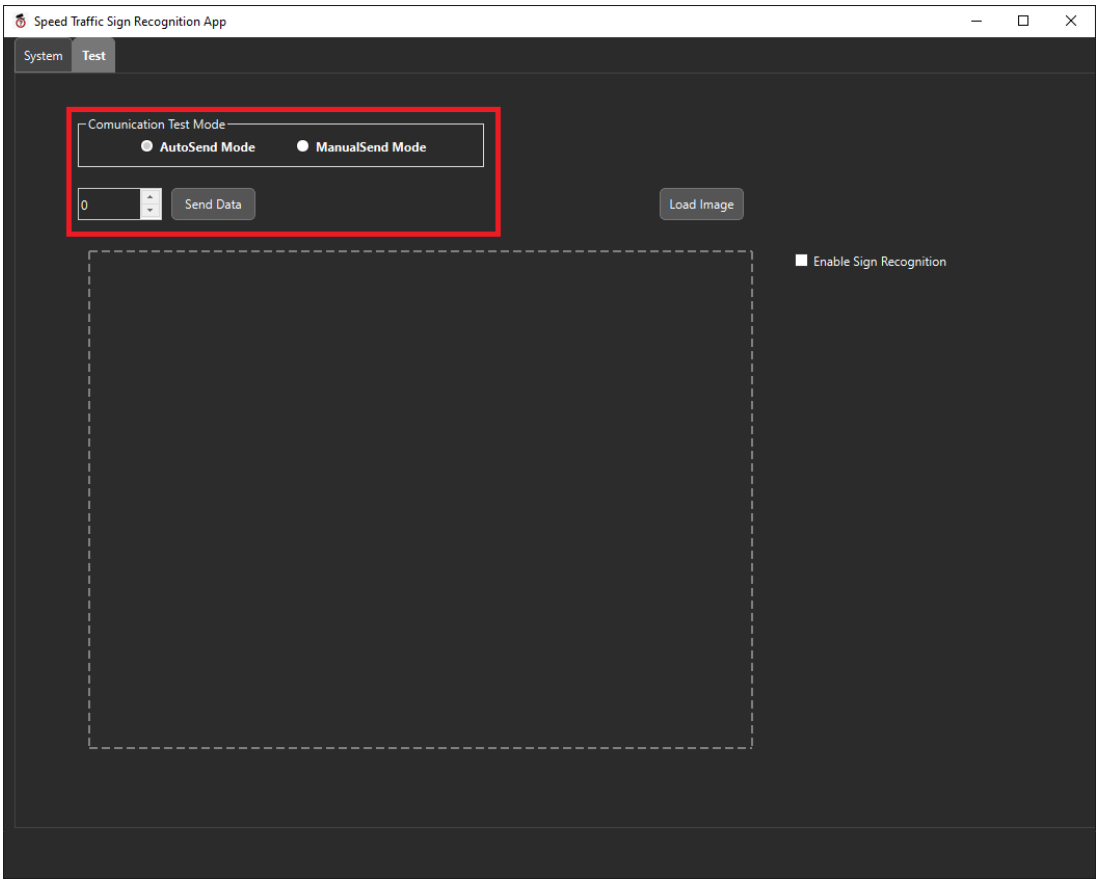


Figure 4.17 DEBUG user interface view communicating with FPGA

Subsequently, reception from the FPGA was verified by implementing a simple free-running counter on it and observing how it reached the application via the Visual Studio Code console.

4.2.2 Neural Model with Static Images

Once the communication has been validated, the correct functioning of the neural model is checked by loading an image from the test dataset from the PC's file system and clicking on the "*Load Image*" button. The results were found to be correct and highly reliable, as indicated by the performance data after the training process.

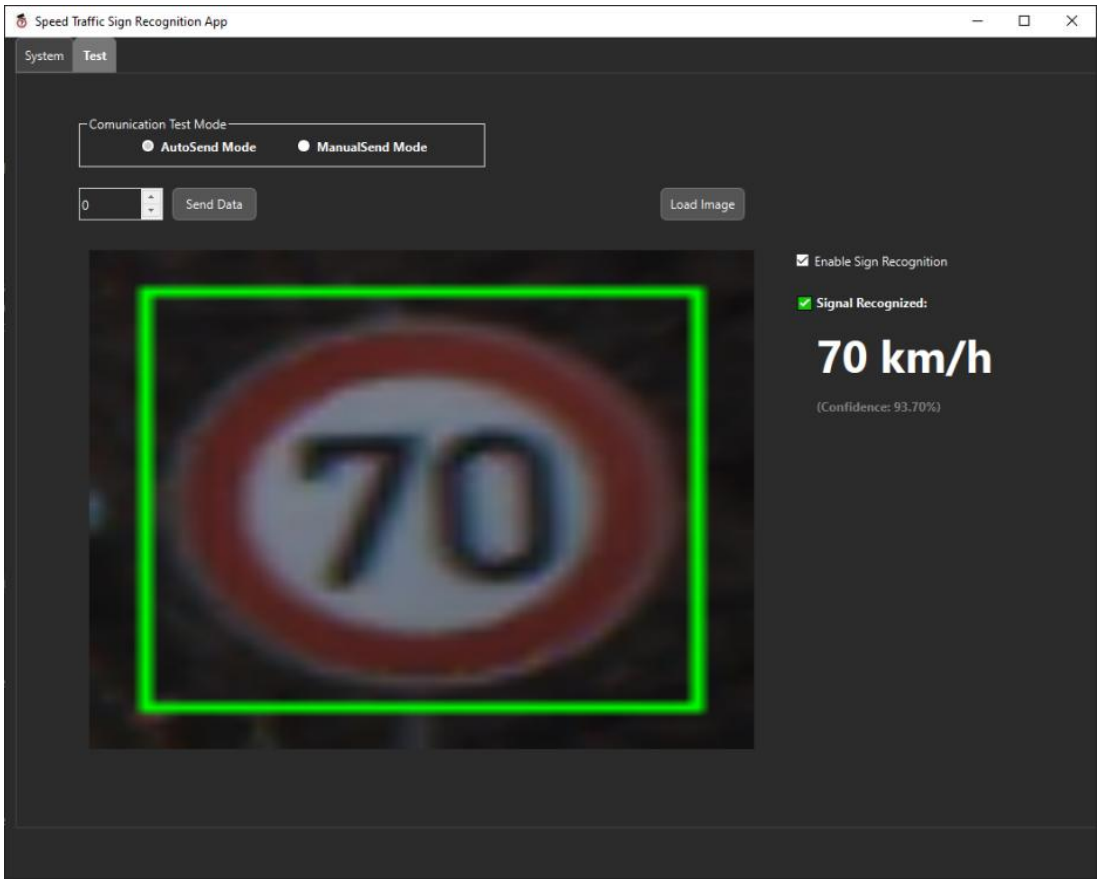


Figure 4.18 DEBUG user interface view after signal recognition

4.2.3 Image Reception

SYNTHETIC

After this, images are sent from the FPGA in synthetic form using the sensor emulator, activating the corresponding switch on the development board (**SW0**). First, they are sent in colour format (***TOP_12MHz_COLOR.bin***), checking that the image with the hardware-defined pattern arrives and is displayed correctly on the interface.

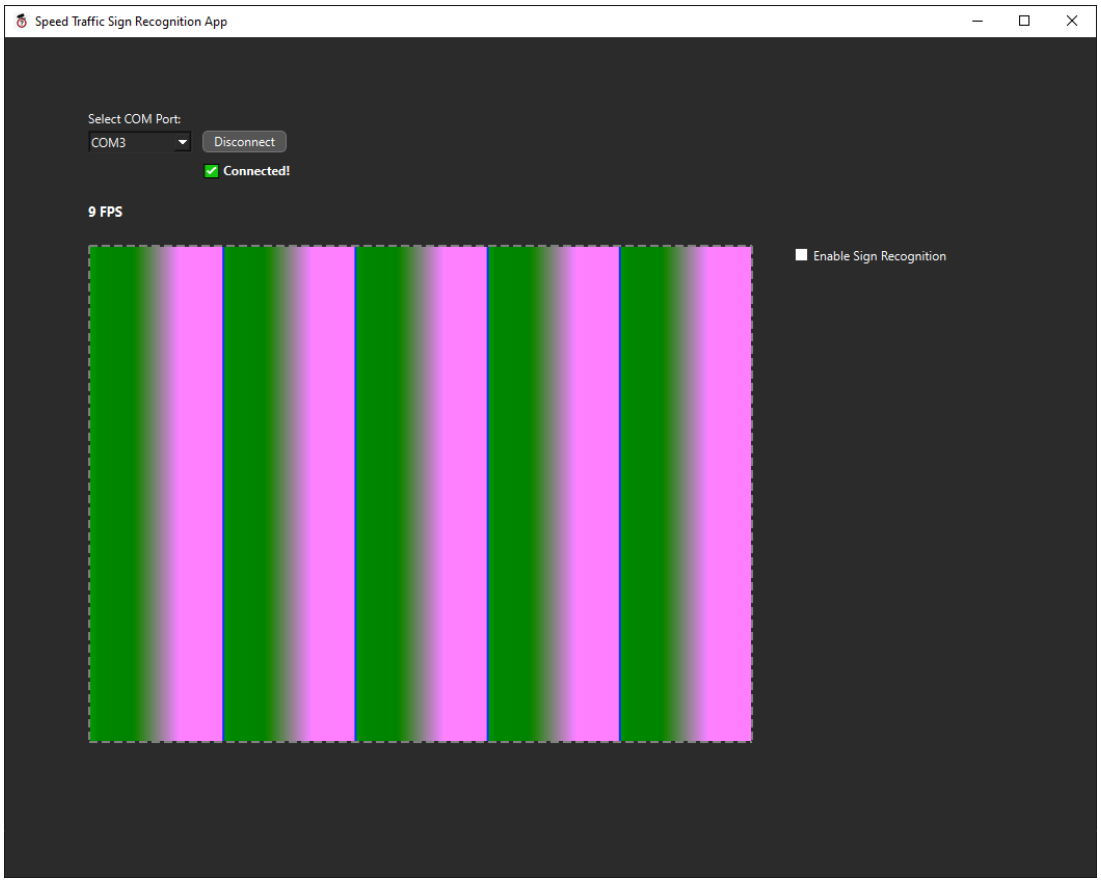


Figure 4.19 Sending synthetic colour images to the PC

The same test is performed in greyscale and at a higher frequency (***TOP_25MHz_GRAY.bin***), obtaining a higher FPS value:

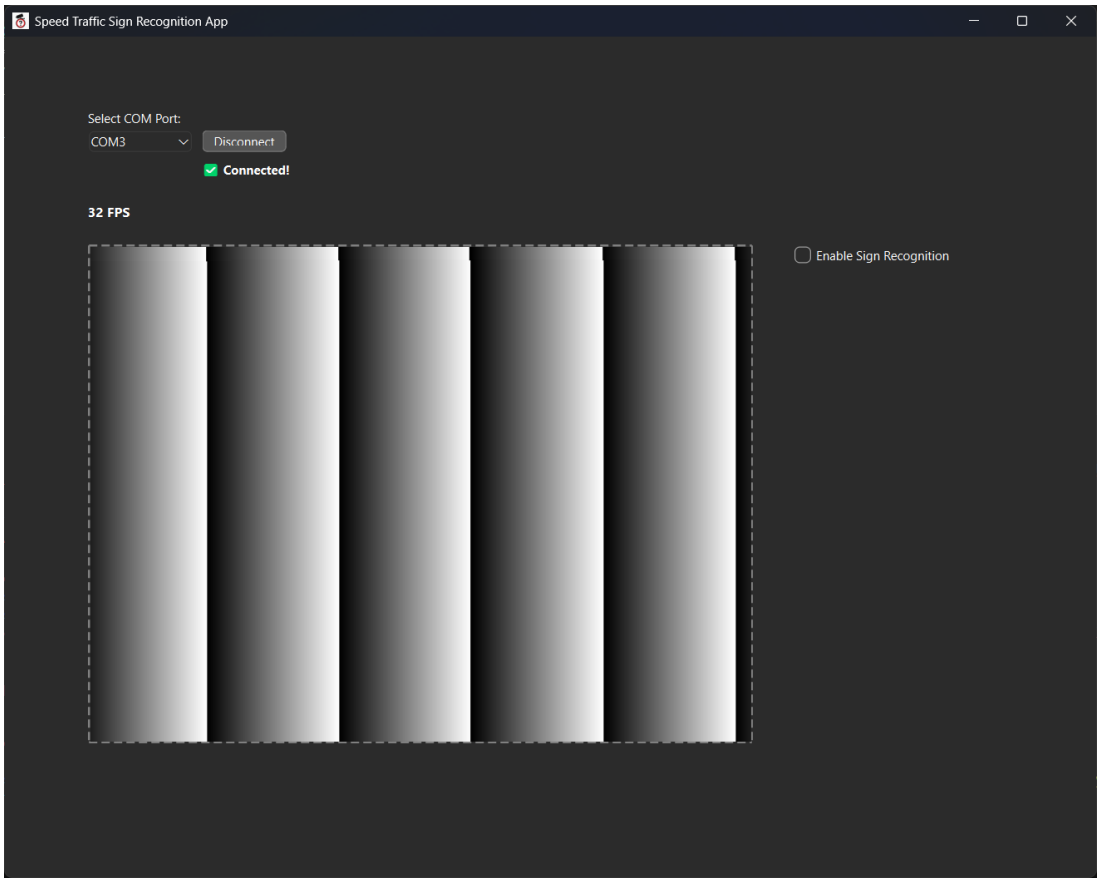


Figure 4.20 Sending synthetic greyscale images to the PC

REAL

Finally, we try connecting the real sensor and disabling the emulation to see the real system in operation. Processing is disabled to check only the communication. We can see that the images arrive correctly, with good quality and at an acceptable rate (12FPS).

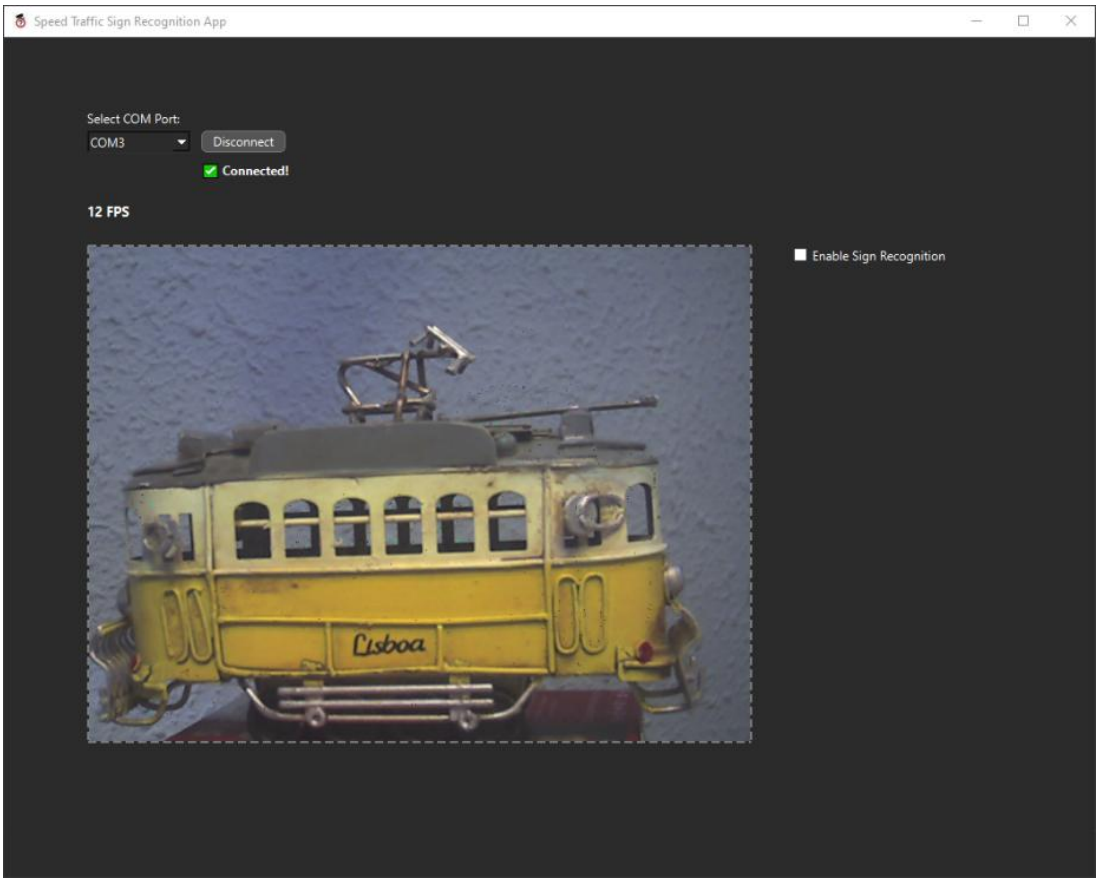


Figure 4.21 Sending real colour images to the PC

To perform the same test that was simulated on the FPGA, it is tested at a higher frequency and greyscale, thus confirming that the number of images received is higher, as mentioned in the hardware section.

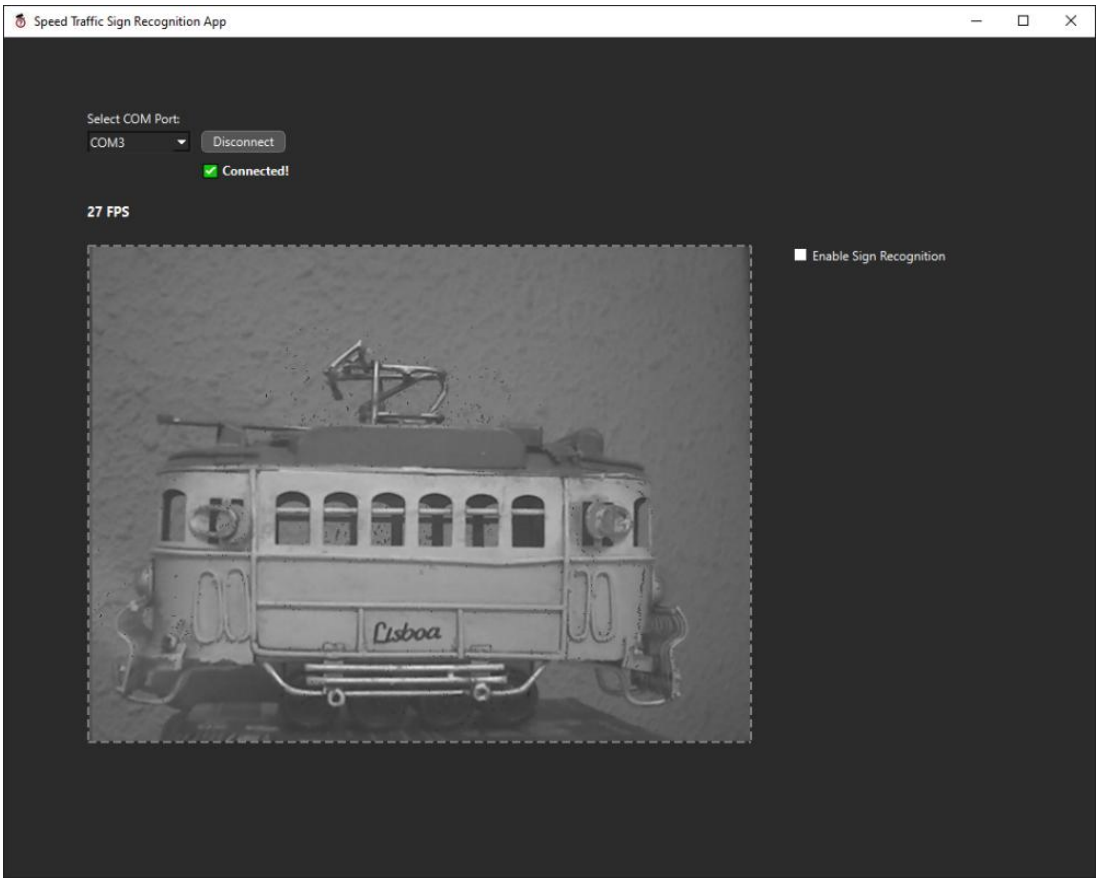


Figure 4.22 Sending real grayscale images to the PC

Both **bitstreams** used, for colour and greyscale formats, are available in the project repository on GitHub, in the /bitstreams folder.

4.2.4 Neural Model with Real Images

Finally, the detection and recognition of speed limit traffic signs is enabled to see the real-time performance of the process. The FPS decreases, but it is still acceptable and gives a sense of “real time”. The sensor is exposed to different signs, showing how it is able to accurately detect the indicated speed (30 km/h) with a high confidence value (91.01%).

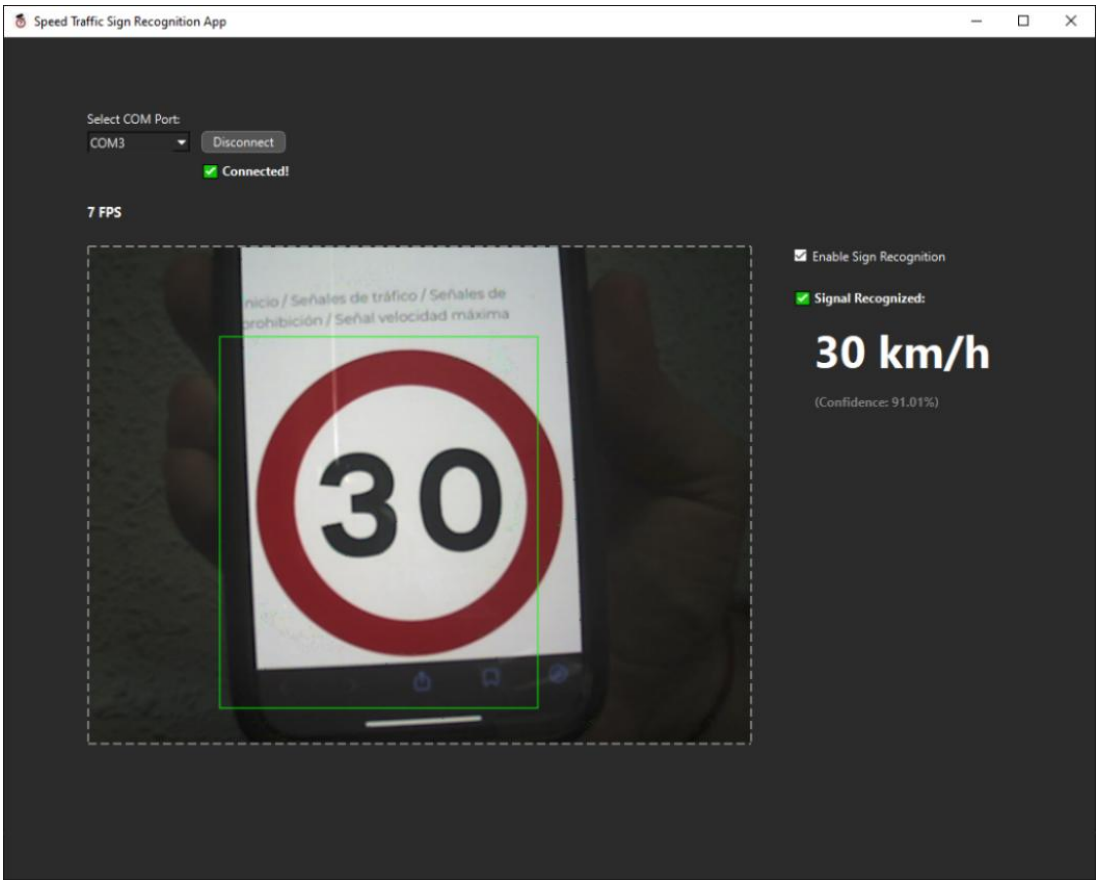


Figure 4.23 Real-time traffic signal detection and recognition

Chapter 5 Conclusion and Future Work

This project has comprehensively addressed the design, implementation, and validation of a system for recognising traffic signs related to speed limits. All of this is based on reconfigurable hardware and software processing using neural networks. From image acquisition to sign detection, a functional, efficient, and scalable solution has been achieved that integrates multiple technologies and meets all the requirements presented in the first chapter of this document.

At the hardware level, a modular architecture has been developed based on blocks designed in VHDL, including synchronous and asynchronous communication interfaces, control systems, FIFO, display visualisation and a complete image acquisition system. This architecture has been verified through functional and post-synthesis simulations and successfully deployed on the FPGA on the BASYS 3 board.

At the software level, an application has been developed in Python using PySide6 (Qt) that acts as a user interface, system controller and inference engine, executing a trained neural model (YOLOv8n). The FPGA communicates with the PC through a FIFO interface exposed by the FTDI module (UM232H-B), which abstracts the complexity of the USB protocol.

However, although the objectives set have been successfully achieved, the project has revealed certain challenges in terms of latencies, software dependency for processing, etc.

One of the most significant improvements proposed is the possibility of **integrating the neural model directly into hardware**, eliminating the dependence on software for signal recognition. To achieve this, it would have been ideal to have an FPGA platform with SoC, such as the Xilinx **Zynq** family, which combines reconfigurable logic with ARM processors and allows the inclusion of embedded **NPUs (Neural Processing Units).**

This approach would allow the performance of software processing to be compared with that of hardware, metrics such as latency and energy consumption to be evaluated, and, in addition, a **completely embedded solution** to be achieved, where the software would act solely as a display and control interface, delegating all intensive processing to the FPGA.

Furthermore, future lines of work could include:

- **Expansion of the dataset** to include other types of signals.
- Improvement of the acquisition and pre-processing pipeline to support colour formats and higher resolutions, and even more interestingly, to adapt the captured image to the needs of neural processing. Currently, the system has been developed and tested with a standard resolution of 640x480 pixels provided by

the **MT9V111** sensor, whose interface has been emulated in this project. Future improvements could also consider the use of more advanced and faster sensors, as well as taking full advantage of the colour information captured.

- Extension of the system to mobile platforms or edge devices.
- Exporting the neural model to optimised formats such as **ONNX** or **TensorRT** for integration into lighter systems.

Chapter 6 Bibliography

[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.

[3] FTDI Chip, "FTDI Chip - Datasheet FT232H," [Online]. Available: 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.

[5] FTDI Chip, "FTDI Chip - Datasheet UM232H-B Module," [Online]. Available: 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=AfmBOort6FlKb4Zw07d4OlHaT7oH_me-lmq13OJGIiqmkd9W9VInDCiy.

[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.

[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/.

Appendix A. GitHub Repository

All source code for the system, including hardware modules in VHDL, simulation scripts, neural network training, and the desktop application, is available in the official project repository: https://github.com/ManuelSN/Speed_Traffic_Sign_Recognition_System

In addition, this repository includes the ***STSRAppInstaller.exe*** installer, which allows the application to be run directly without having to manually install Python or dependencies, designed for **end users**. The installation process is described in **Appendix B. Software Installation**.

Appendix B. Software Installation

The process for installing the project software is shown below. To do this, you will need to download and run the *STSRAppInstaller.exe* file for Windows.

This file has been generated using the "Inno Setup Compiler" utility, so it is a self-contained package with all dependencies already installed. Once this is done, you will be asked if you want to allow this application to make changes to the device. After confirming, the following installer windows will appear:

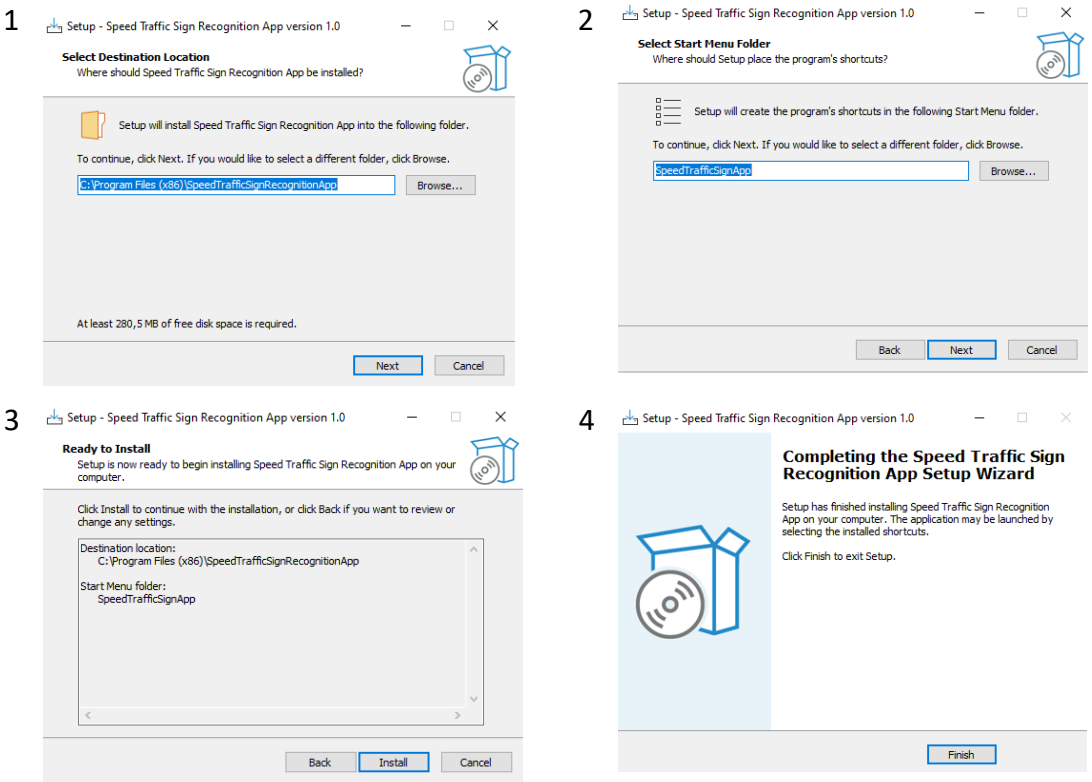


Figure 0.1 Software installation process

In the first window, select where you want to install the application. After clicking "Next", you will be taken to the second window where you can indicate where to create the shortcut to the application. After this, in the third window, click on "Install" to begin the installation process, which will take a few seconds.

Once finished, close the fourth and final window, and you can now search for and open the *"Speed Traffic Sign Recognition App"* in the Windows bar:

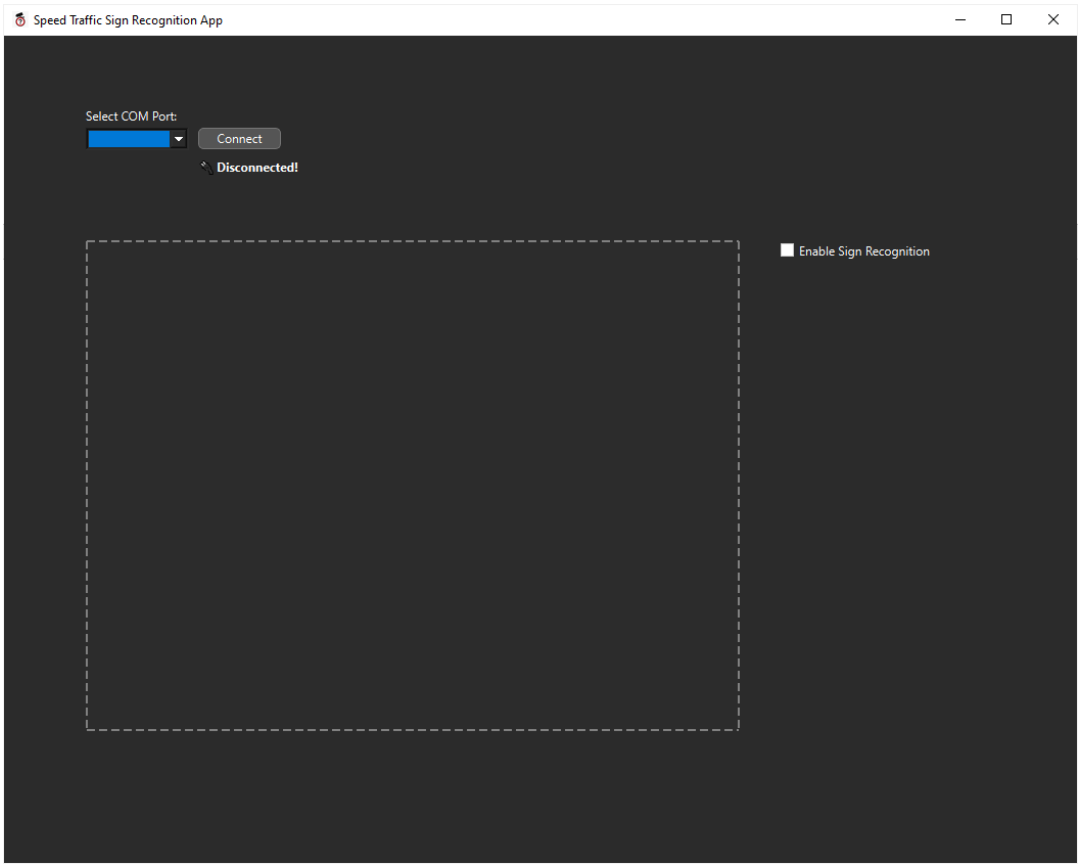


Figure 0.2 Initial view of the application