University of Thessaly

Department of Electrical and Computer Engineering

Fall 2022/2023

Integrated Master Thesis

I^2C to UART-GPIO-I^2C Bridge on FPGA with Zephyr RTOS Driver

Charalampos Patsianotakis

Supervisors: Christos Sotiriou, Dimitris Karaberopoulos, Georgios Stamoulis

I^2C σε UART-GPIO-I^2C Γέφυρα σε ΣΕΠΠ με Πρόγραμμα Οδήγησης σε Λειτουργικό Zephyr

Acknowledgements

I am grateful to the numerous individuals who aided me in my academic journey, although the completion of my thesis project was a solitary endeavor. It would be challenging to express my gratitude to everyone within the constraints of one page, so I will highlight a few key individuals.

I would like to extend my heartfelt appreciation to Prof. Christos Sotiriou for his unwavering guidance and mentorship throughout my time in the Circuits and Systems lab. Also, he was instrumental in helping me launch my first job, where I was able to gain valuable knowledge and experience. I am also grateful to Prof. D. Karaberopoulos and Prof. George Stamoulis for their participation in the three-member committee for my thesis. In addition to these professors, I would like to thank all of the professors who imparted their knowledge and experience during my academic journey, particularly Prof. N. Bellas, Prof. C. Antonopoulos, Ms. V. Doufexi, and Prof. N. Evmorfopoulos.

I would also like to acknowledge the members of the Circuits and Systems lab, with special thanks to Dr. Nikos Sketopoulos and Stavros Simoglou. I would like to express my appreciation to everyone I collaborated with on academic projects and the AgroJason Project. I would like to thank the company I work, Centaur Analytics, for lending me the CozIR-A sensor that I used for my physical experiments. I would also like to thank my current and former colleagues, including my first manager and mentor Nikos Oikonomou, and my current teammate Antonis Sioutas.

I would not have been able to achieve my goals without the support of the people around me. I would like to express my gratitude to my parents, grandparents, godparents, aunts, and the rest of my family for their emotional and financial support throughout my studies. I would like to especially thank my father for instilling in me an interest in the field of electrical engineering through our joint electrical projects as a teenager. I would also like to thank Thomas Marinou and Olga Tsima for their unwavering support during the final years of my studies. Last but not least, I would like to express my gratitude to Maria for her love, support, and patience as I worked on my thesis project while balancing work.

Abstract

The integration of devices into the Internet of Things (IoT) network is experiencing significant growth. With the constant evolution of application demands, it is necessary to continuously develop new designs to accommodate these changes. To facilitate the integration of new features into the IoT system, the electronics of End node devices are comprised of two parts. The first includes the Main Board, which incorporates the core Microcontroller Unit (MCU), various Radio-Frequency (RF) modules, and other essential components. The includes the Peripheral Boards, designed specifically for the specific application requirements of the device. The goal is to maintain a single Main Board for all devices, thereby reducing logistics management effort and costs. However, this is not always feasible due to variations in peripheral device interfaces. This thesis presents the development of an FPGA device that bridges these different Peripheral Boards, simplifying the connection of multiple devices to the Main Board, accompanied by its software driver.

Περίληψη

Ο αριθμός των συσκευών που συνδέονται στο Διαδίκτυο των Πραγμάτων (ΔτΠ) αυξάνεται με γρήγορο τρόπο. Οι απαιτήσεις των εφαρμογών αλλάζουν και διαφορετικές σχεδιάσεις απαιτούνται για να τις πληρούν. Έτσι, οι τελικοί κόμβοι σε ένα σύστημα ΔτΠ αποτελούνται από δύο μέρη. Την κύρια πλακέτα που περιέχει τον κεντρικό Μικροεπεξεργαστή, διάφορες μονάδες ράδιου και άλλα κύρια κομμάτια και τις περιφερειακές πλακέτες, ειδικά σχεδιασμένες για την εκάστοτε εφαρμογή. Ο στόχος είναι η διατήρηση της κυρίας πλακέτας ως μοναδικής και να έχει την δυνατότητα να εφαρμοστεί με όλες τις πιθανές περιφερειακές πλακέτες για την μείωση διαχείρισης επιμελητείας, με το ελάχιστο όμως κόστος. Αυτό δεν είναι πάντα εφικτό λόγω των πολλαπλών διεπαφών που ενδέχεται να έχουν οι περιφερειακές συσκευές. Σε αυτήν την διπλωματική εργασία παρουσιάζεται η ανάπτυξη μίας συσκευής που σκοπεύει να διευκολύνει την διασύνδεση πολλαπλών συσκευών στην κύρια πλακέτα.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 IoT Application Description

The Internet of Things (IoT) is a well-known trend in the current era, with multiple architectures designed and developed for various applications. The majority of IoT architectures consist of three key components: a cloud application, a user interface application, and edge devices. The cloud application, which runs on data centers with unlimited resources, serves as the main logic center for the IoT application. The user interface application, on the other hand, runs on end-user devices such as smartphones or desktops and provides access to the end-users. The edge devices, which act as the interface between the physical world and the cloud, collect data from sensors or actuate based on commands received. In some cases, intermediate gateway devices are required as the environment of edge devices may not allow direct connection to the cloud.

In applications such as agriculture, power consumption is a major concern, and to mitigate this, edge devices operate in a duty cyclic sleep mode. The sensors sleep for a specified time, sample the desired measurements, and wirelessly transmit them to the gateway before sleeping again. Similarly, actuators may turn on their RF module, wait for a command, actuate, and then sleep again. The core system of the device is not always on but sleeps and handles the peripheral element in a specific time slot.

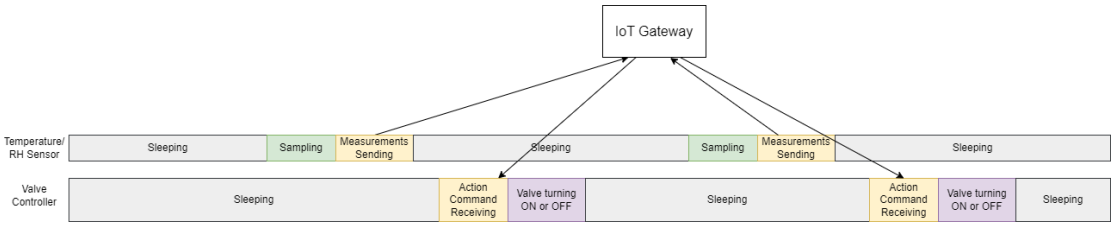

Figure 1.1.1: Network Duty Cyclic Sleep Operation

An example of an IoT application in agriculture is a smart farming system for watering plants. The farmer can access the system through a user interface application on his smartphone, while the cloud application handles the core functionality. A wireless network is established in the farm and temperature and relative humidity sensors provide the environmental conditions to the system. The system controls the

flow of water and the environment of the plants through valves that are actuated based on the data received.

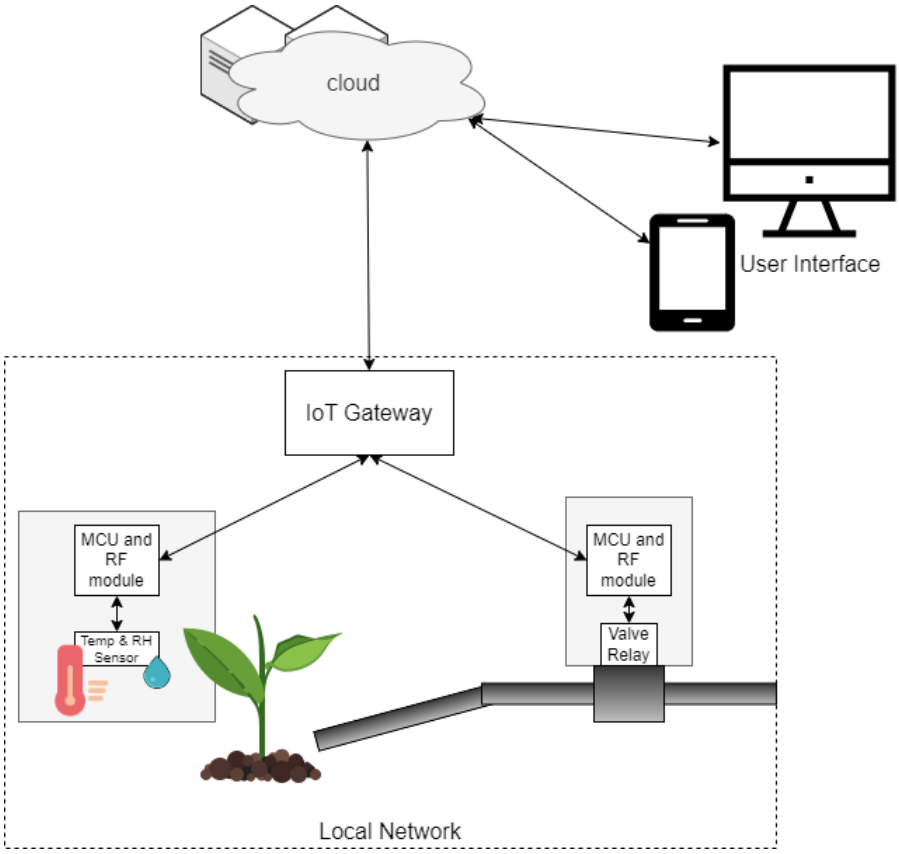

Figure.1.1.2: Smart Watering IoT Architecture

The architecture of the Edge Device comprises of a microcontroller unit (MCU), a Radio-Frequency (RF) module, and various peripheral devices such as sensors, relays, solar charger controllers, battery managers, among others. Some components, such as the MCU and RF module, are utilized in all products, while others are specific to certain products. For example, a temperature sensor may share the same MCU and RF module with a valve controller, but it would not require the relays that the latter would use and vice versa. The widely produced components are incorporated onto the Main Board, while the components with limited production are integrated onto Peripheral Boards.

A Stock Keeping Unit (SKU) code is generated for each specific design of the described boards. Two boards can only have the same SKU code if their hardware

design is identical. Having multiple SKU codes leads to increased complexity in logistics operations and field support.

1.2 Defining the Logistics Problem for multiple designs

The connection between the Main Board and the Peripheral Boards might be established through the use of a header interface and interconnection cables. In a specific example of a valve device that includes the Main Board and a Peripheral Board with a valve and an indicator LED, the Peripheral Board is attached to the Main Board via a 2-pin header interface, which is connected through a 2-wire interconnection cable. The connection between the Main Board and the microcontroller unit (MCU) is finalized by the connection of two MCU pins to the header on the Main Board.

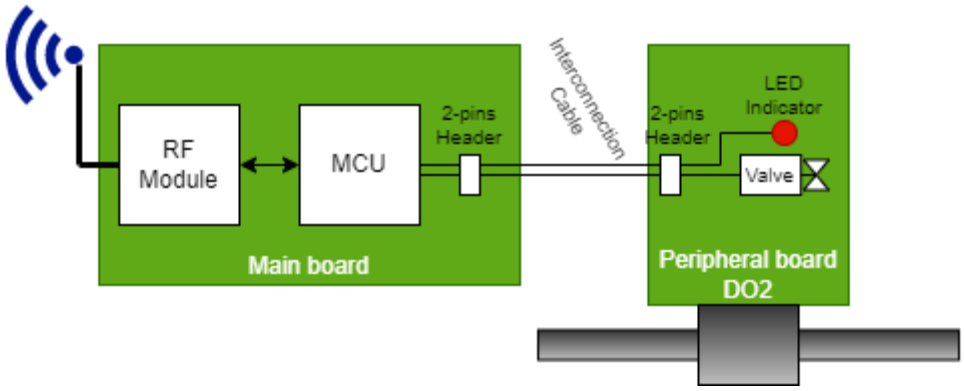

Figure 1.2.1: Simple 2 Digital Outputs Design

In order to accommodate a new product request for a device with two valves and an indicator LED, two approaches are considered. The first approach involves replacing the current Main Board design with an updated version that has three Digital Input/Output (DIO) pins. This would result in all newly produced devices, including those with only one valve, incorporating the more expensive 3-pin Main Board. However, this approach would also require updating the Main Board design with additional Digital Outputs as more complex designs emerge.

The second approach entails maintaining the existing 2-pin Main Board design for devices with a single valve and utilizing a separate 3-pin Main Board for the new device with two valves. While this approach adheres to the principle of a single Main Board design, it will result in the creation of multiple Stock Keeping Unit (SKU)

codes for the Main Boards that are produced in parallel, and may complicate logistics and field operations.

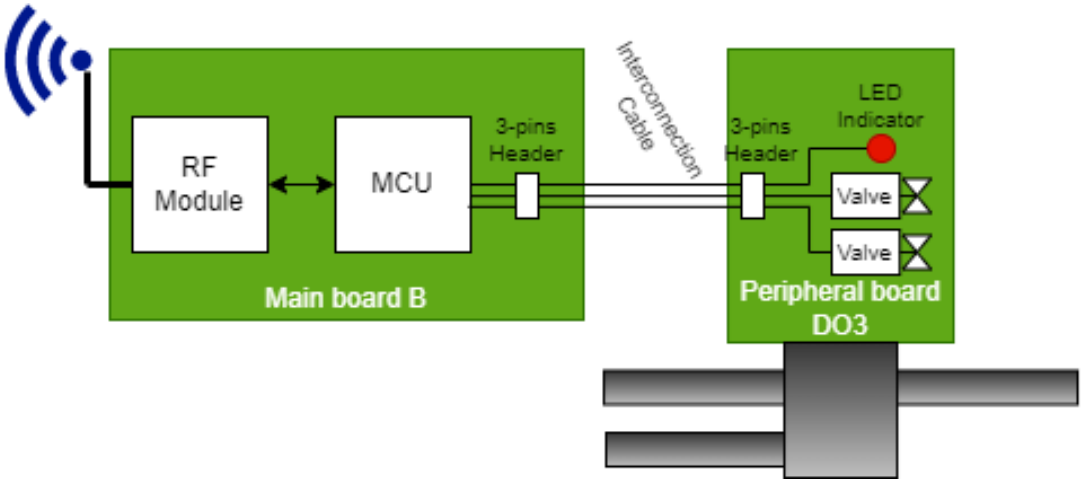

Figure 1.2.2: 3 Digital Outputs Design

A requirement has arisen for a new product that will determine the presence or absence of water using water electrodes. The product will require a digital input to obtain its status and, as per the specified requirements, it must be powered by a solar panel. To effectively manage the solar panel, battery, and load output, a Solar Panel Management Integrated Circuit (IC) is necessary. This will provide two additional status messages: Charging Status and Solar Panel Connected Status. To gather this information, two additional digital inputs are required on the Main Board, as they are crucial for the calculation of the product's lifetime and error indication. Therefore, it is necessary to design a Main Board with a 1-pin header and a 2-pin header.

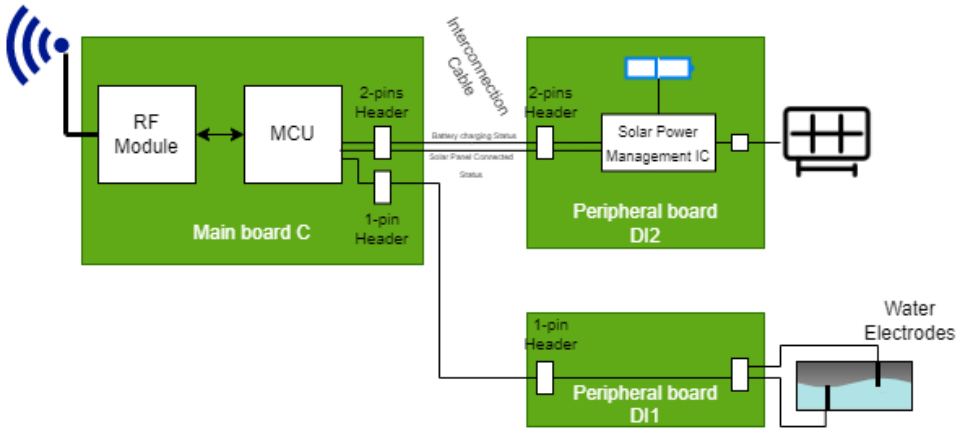

Figure 1.2.3: Water Existence checker powered by solar.

The business has requested a new sensor device to measure the Temperature and Relative Humidity (RH) of the air near to the plant. The sensor element will be utilizing a UART interface, and the firmware designer should have designated two UART pins on the MCU to be connected to a 2-pin header that connects to the UART interface of the sensor element. The current 2-pin Main Board design can be used for this purpose.

However, the product manager requires a device that can measure both the temperature and RH of the air and moisture near the plant, which would necessitate the use of two sensors connected to the Main Board. In this scenario, the UART interface sensors used in the previous device cannot be utilized with the same Main Board as two additional pins are required to instantiate two buses.

One approach to resolve this issue is to implement a new Main Board with two 2-pin headers. Another option is to replace the sensors with I2C interfaces. As described in the Technical Background Chapter, the I^2C bus consists of masters and slaves, each with a unique address, receiving and sending data as requested. Most of the sensors with I^2C interface are set by factory to a specific address. However, there some whose last one or two bits of the address might be configured by specific pins. So, two I^2C slave sensors might be connected on the same bus if they are included in the last category.

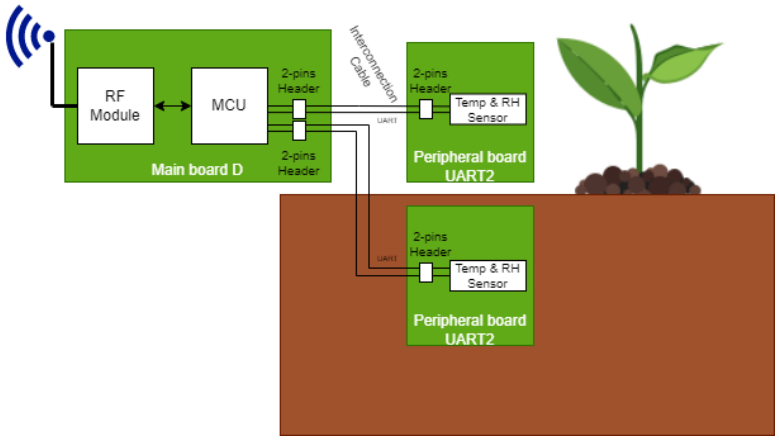

Figure 1.2.4: New Main Board needed to support multiple UART interface sensors.

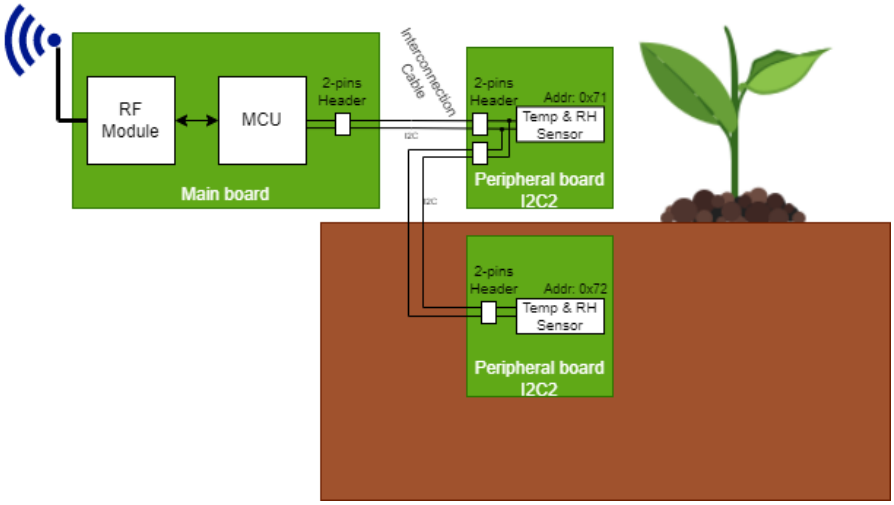

Figure 1.2.5: I^2C Interface sensors for air and moisture conditions

The watering system must be updated to accommodate the requirements of a new plant that necessitates the measurement of both air and high and low levels of moisture. Due to the scarcity of sensors with more than two I^2C addresses that are readily available, it has been determined that a separate bus will be required to support the three sensors needed for this application.

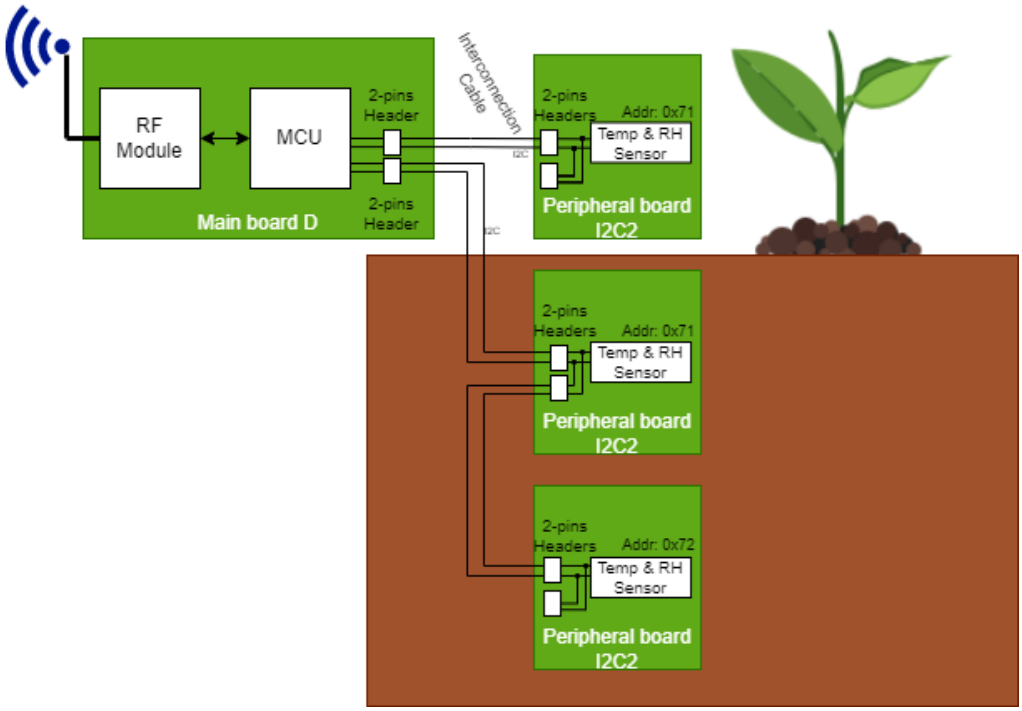

Figure 1.2.6: New Main Board required to add more I^2C sensors.

1.3 Serialization on the same Bus

What if an additional element is instantiated on the Peripheral Board to manage the peripheral device signals? For example, a Serial-Digital Output module could be set on the Peripheral Board to control the valves. The software running on the MCU would send a command over the serial bus to this module, which would set the output value through its GPIO pin.

A first approach of the serial bus could be the UART one. The UART protocol is simple, a simple amount of bytes can be sent as is and translated in the converter module to Digital Output. So, 2 wires are needed, independently of the supported outputs on one Peripheral Board, removing the complexity of designing specific Main Board, but also reducing the wires of the interconnection cables.

As far as the Digital Inputs, the UART could also be used. The updated value of an input can immediately be sent to the MCU on the UART after the signal edge.

If a device is requires to include both the Peripheral Board with valves and the Solar panel, an updated Main Board with support for two UART interfaces is needed. To address this, the UART bus can be replaced with another bus where each Peripheral

Board can be individually targeted. This way, the software can determine where to send or receive data.

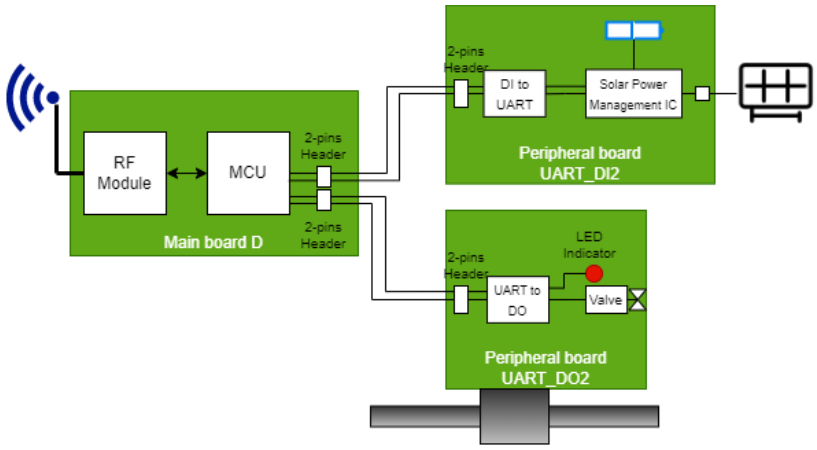

Figure 1.3.1: UART to D I/O still cannot support all possible cases.

A widely used bus that meets these requirements is the I^2C bus. In this case, the serial to DI/O converter might be a I^2C bus slave device, with a configured address, unique in the same bus. However, a disadvantage for the Digital Input interface is that the information about the edges is not directly communicated to the MCU software. But for the applications described, where the input is simply sampled to obtain a status at a specific time, this is not a problem.

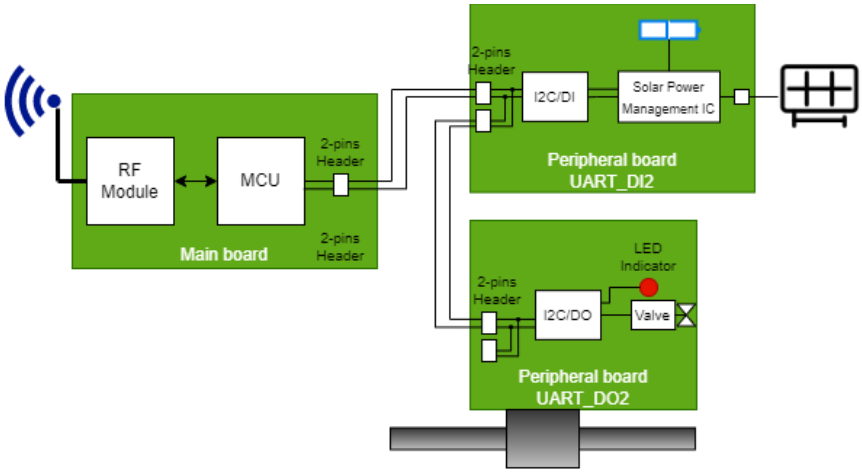

Figure 1.3.2: I^2C to D I/O can operate for the application needs.

In the same way, instead of replacing the UART interface sensors with I^{2}C ones, a UART to I^{2}C converter would simplify the designer life. This is a significant benefit if the UART interface sensor can easily be found in the market, or it has a better performance.

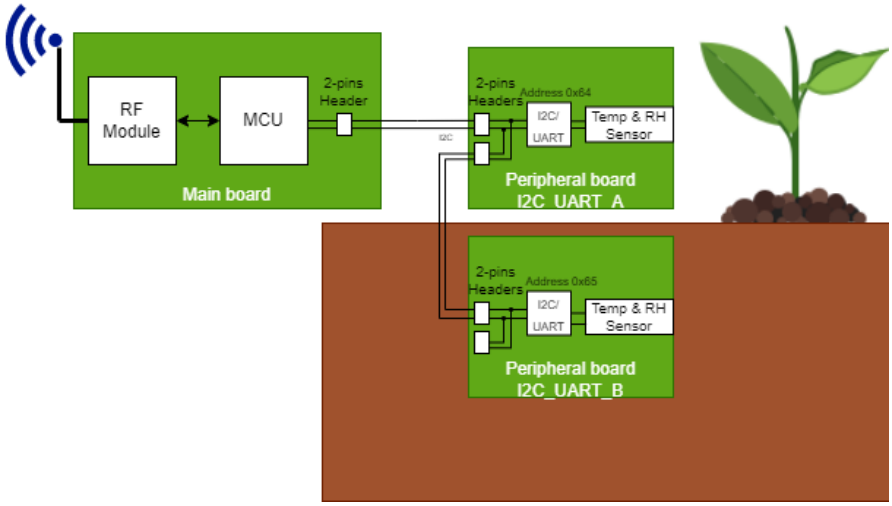

Figure 1.3.3 UART interface sensors communicating to the MCU from the same wires.

Regarding the I^{2}C interface sensors, the number that can be attached to the same bus is limited based on the available pins for address configuration. To increase the capacity for connected devices, an I^{2}C-to-I^{2}C bridge can be utilized as an intermediary between additional devices and the MCU. The MCU will recognize the bridge device and the bridge will transfer data from its sub-bus to the MCU.

1.4 I^{2}C Bridge

By delegating the management of peripheral devices to the Peripheral Board, the logistics effort required for supporting a variety of devices has been significantly reduced, as only one design for the Main Board is necessary. This can be further reduced by integrating all I^{2}C bridge modules into a single component. During the initialization process, software commands enable the I2C Bridge to identify itself as I^{2}C-DO, I^{2}C-DI, I^{2}C-UART, or I^{2}C-I^{2}C. As a result, the same hardware component can be utilized on all Peripheral Boards.

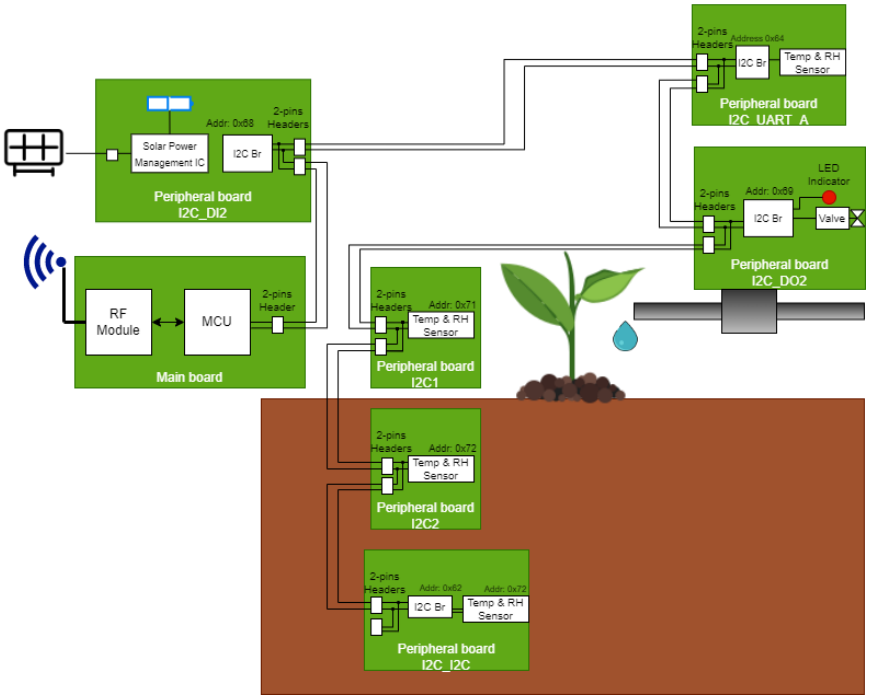

Figure 1.4.1: Multiple types of Peripheral Boards communicating with Main Board on the same bus.

One of the objectives is to minimize design costs, and thus, the I^2C Bridge component should be as cost-effective as possible. Tiny FPGAs offer a potential solution, as some are available on the market for less than 10 euros with industrial specifications. These devices are capable of executing data transactions quickly, making them virtually invisible at the application level.

Chapter 2 Technical Background

2.1 IC’s Communication Interfaces

2.1.1 General Purpose Digital Input / Output

The Digital Signal is a binary representation of information, defined by two voltage levels: the "High" level, which is close to the supply voltage, and the "Low" level, which is close to the ground or reference value. This type of signal is used to convey simple information, such as a status, by representing a "true" or "1" state with a High voltage and a "false" or "0" state with a Low voltage. Unlike an analog signal, which represents a continuous range of values, a Digital Signal conveys information in a discrete manner. It can be used to transmit information as a single bit or as a group of bits in communication protocols, forming data packets.

The General Purpose Digital Input/Output (GPIO) is an interface that controls simple digital input/output signals. It enables devices like microcontrollers to connect with other devices such as LEDs, buttons, motors, battery chargers, FPGAs, PLCs, or even other microcontrollers. This versatility makes GPIO pins a valuable component in a variety of applications, including robotics, home automation, and Internet of Things (IoT) projects.

2.1.2 UART

The Universal Asynchronous Receiver Transmitter (UART) is a hardware communication protocol that employs asynchronous serial communication with adjustable data transfer speed [1]. The UART protocol involves two signals between the two communicating devices: the Receiver (Rx) and the Transmitter (Tx).

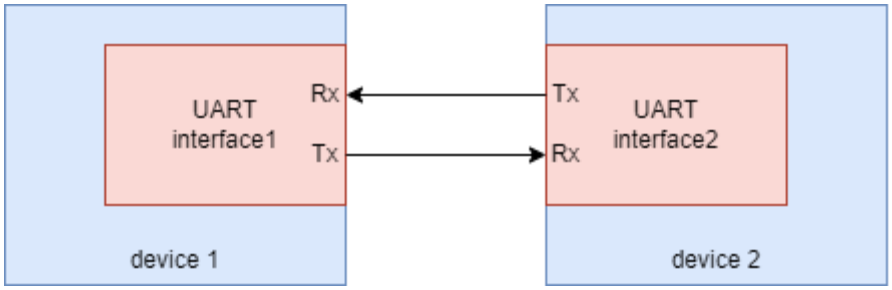

Figure 2.1.1: UART Lines

The absence of a clock signal to synchronize the transmission of output bits from the transmitting device to the receiving end is a defining characteristic of the UART protocol. Instead, the two devices communicate with each other at a pre-determined baud rate to achieve synchronization. The most commonly used baud rates are listed in a table.

Baud Rate	Bit Duration
300 bits/s	3.333 ms
1200 bits/s	833.333 us
4800 bits/s	208.333 us
9600 bits/s	104.167 us
19200 bits/s	52.083 us
38400 bits/s	26.042 us
57600 bits/s	17.361 us
115200 bits/s	8. 681 us

The mode of transmission in UART is packet-based. Each packet consists of a Start Bit, a Data Frame, a Parity Bit, and a Stop Bit.

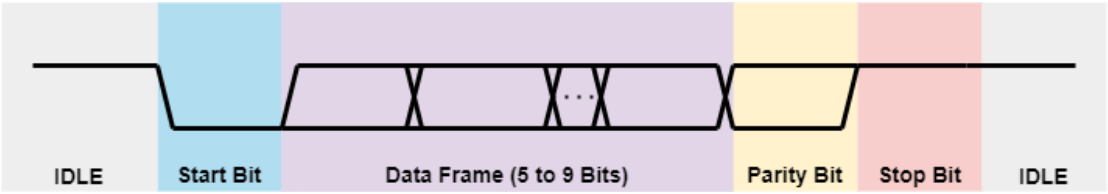

Figure 2.1.2: UART Protocol Packet

The UART signal is usually at a high level during the idle state, when there is no transmission. When the Transmitter wishes to send a packet, it lowers the signal value

for a specified time period, which is recognized by the Receiver as the start bit and signals the Receiver to prepare to receive the packet.

The Data Frame, which follows the Start Bit, represents the data being transferred, and its size can range from 5 to 8 bits, or 9 bits if a parity bit is not used. In this case, it is assumed that the Data Frame has a size of 8 bits.

The Parity Bit serves as a first-level mechanism for validating the transmitted Data Frame bits, indicating whether the number of 1s in the Data Frame is even or odd. The UART Transmitter generates this information by applying an XOR function to the Data Frame and transmits it, along with the Parity Bit. The Receiver then generates this information in the same way and compares it to the received Parity Bit. If the received and generated parities do not match, the receive operation is considered to have failed.

Finally, the Stop Bit signals the end of the transmitted packet. If the Stop Bit is sampled as high, the receive operation is deemed successful.

2.1.3 I^2C

The Inter Integrated Circuit (I^2C) is a simple bidirectional 2-wire bus for efficient inter-IC control. [2] It requires two bus lines: a serial data line (SDA) and a serial clock line (SCL).The SDA line is used for data transmission, while the SCL for synchronization between the 2 nodes. During the SCL high state, the SDA should keep its value stable, unless a START or STOP condition should be triggered.

The Inter-Integrated Circuit (I^2C) bus consists of controllers, also known as Masters, and targets, also referred to as Slaves. Some devices have the capability to function as both Master and Slave. The Master is responsible for generating the clock signals and for either providing or requesting data from a Slave. For the Master to target a Slave, the latter must have a unique 7-bit address on the bus. In the event that multiple Masters are present on the same bus, an arbitration procedure exists to determine which Master will control the bus. [2] In most IoT edge devices, the I^2C bus only has one Master, which is typically the microcontroller. As a result, the feature of multiple Masters is not always necessary and is omitted to simplify the design and reduce costs for smaller and less expensive Field-Programmable Gate Arrays (FPGAs).

The I2C bus specification outlines four available speed modes: standard mode with rates up to 100 kbit/s, fast mode with rates up to 400 kbit/s, fast-mode plus with 1Mbit/s, and high-speed mode with 3.4Mbit/s.

In the idle state, the SCL and SDA signals are not driven by any node on the bus and are pulled up to a high voltage by pull-up resistors. When the Master wants to initiate communication, it pulls the SDA signal low while the SCL signal remains high, creating a START condition. The Master then pulls the SCL signal low to begin writing the first bit. The packet is divided into bytes and each byte can be transmitted by both the Master and Slave. Each byte starts with the most significant bit and ends with the least significant bit. After the byte is written, the receiver acknowledges the transfer by pulling the SDA signal low during the next cycle. The first byte, driven by the Master, contains the 7-bit address of the Slave device and the read/write operation in the Least Significant Bit (LSB).

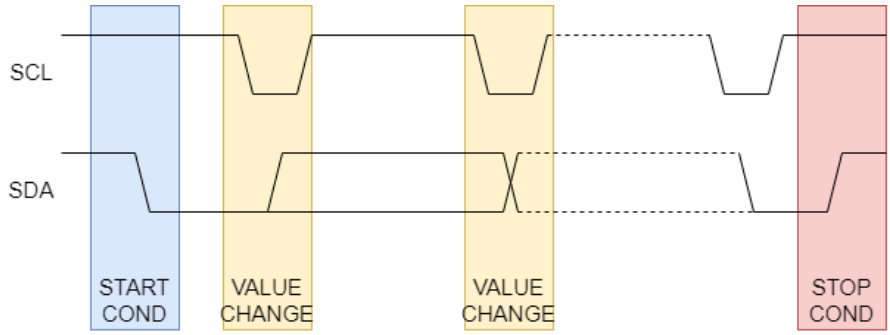

Figure 2.1.3: I^2C Start-Stop conditions.

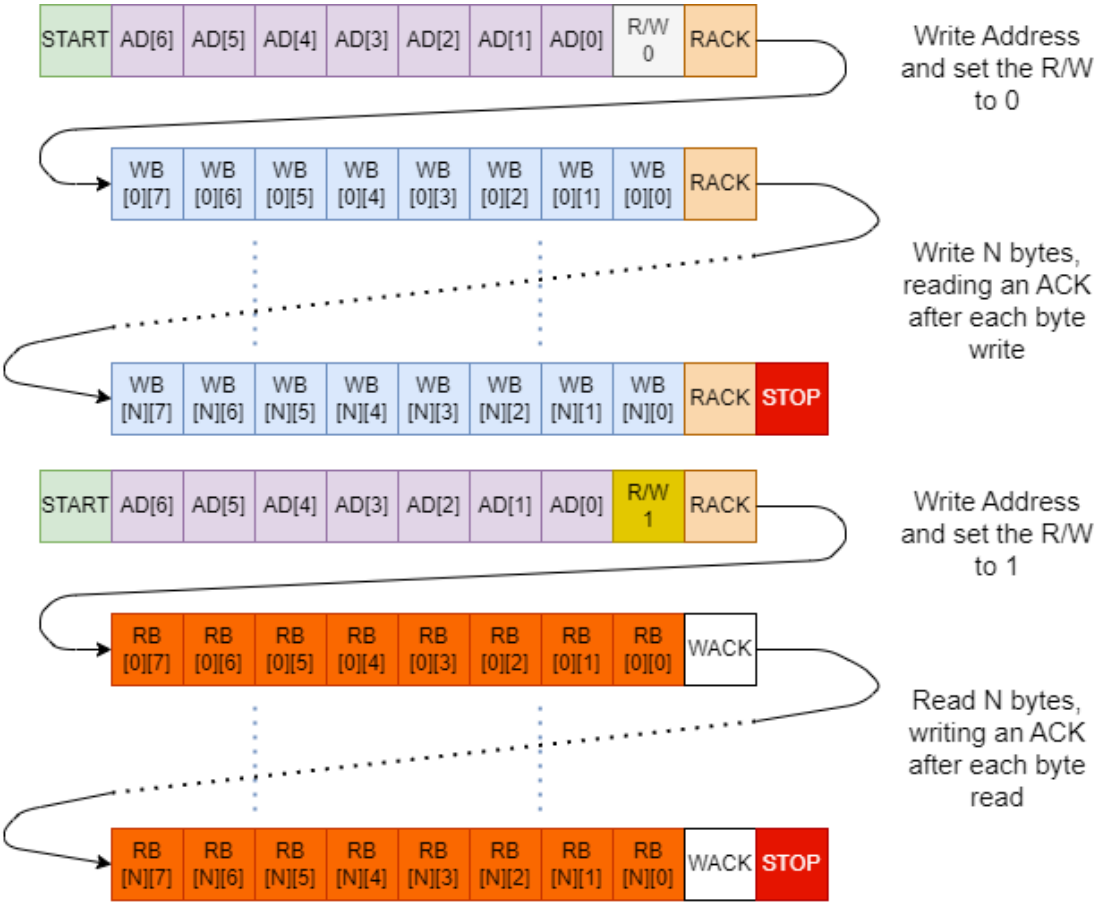

Figure 2.1.4: I^2C read and write packets.

In most cases of peripheral Integrated Circuits (ICs) for IoT applications, communication between the master and slave devices involves specific registers. For write operations, the master sends the address of the register along with the R/W bit set to write mode, followed by the data to be written. For read operations, two I^2C transactions must occur. The first transaction sends the address of the desired register, with the R/W bit set to write mode, allowing the slave device to locate the desired register. In the second transaction, the master retrieves the data from the pointed register by sending the address along with the R/W bit set to read mode. In the event of multiple reads from the same register, most devices only require the first register pointing process. In some cases, simply pointing to a specific register can trigger an operation on the slave device.

.

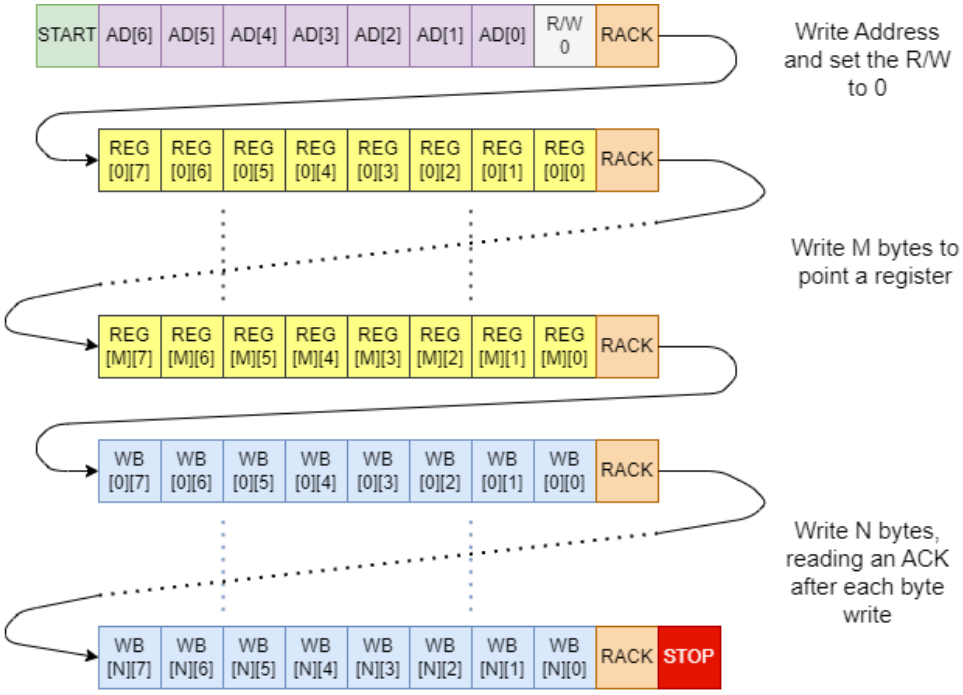

Figure 2.1.5: Write to a register of an I^{2}C slave device.

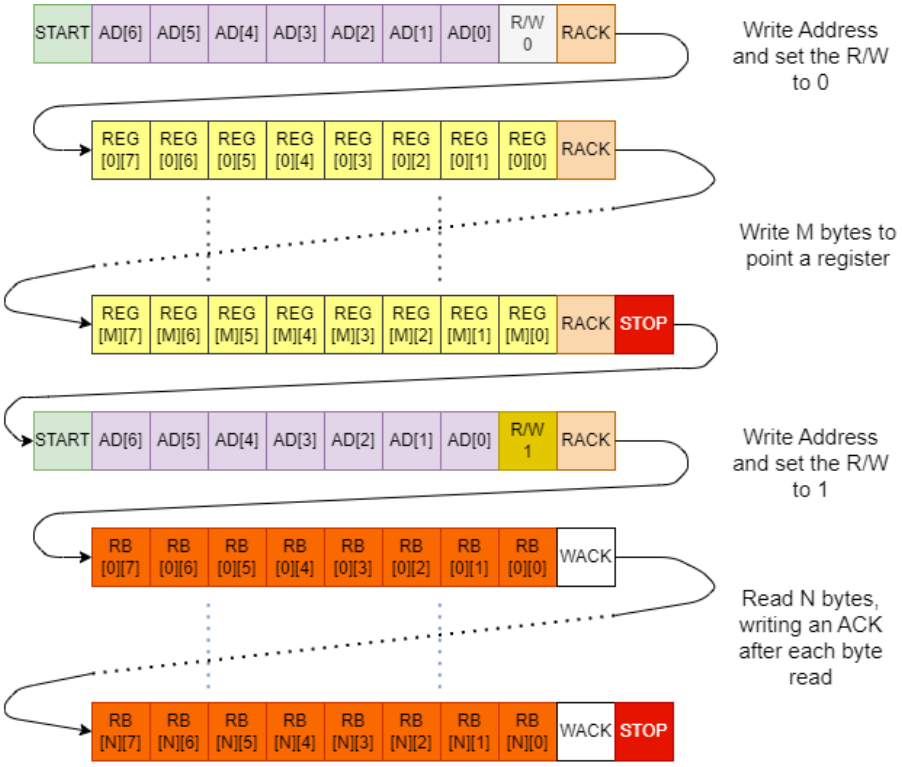

Figure 2.1.6: Read from a register of an I^{2}C slave device.

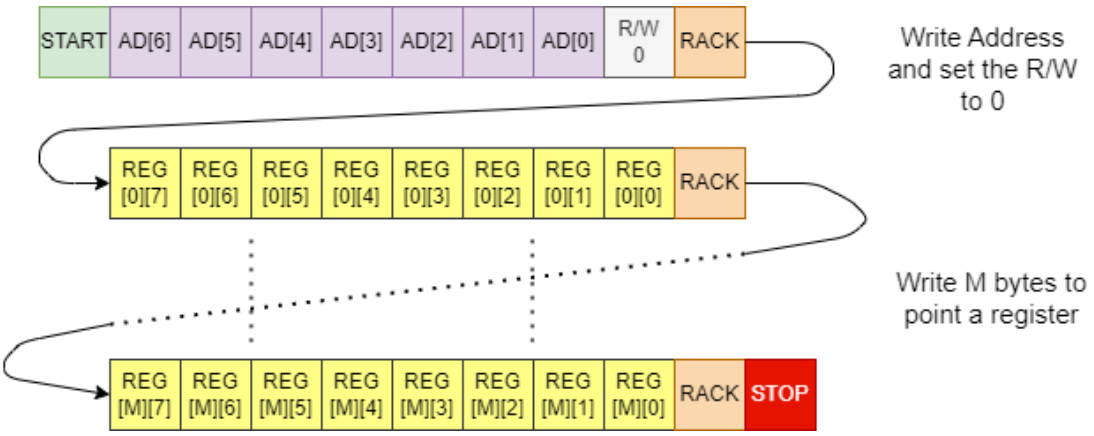

Figure 2.1.7: Point a register of an I^2C slave device.

For the implementation of the project, the method of writing and reading data from the registers of a slave device through the I^2C protocol will be utilized. It should be noted that, as of the current implementation, the capability of simply pointing to a register will not be supported.

2.2 Digital Hardware Development

2.2.1 FPGA

A Field-Programmable Gate Array (FPGA) is a type of integrated circuit that can be reconfigured after manufacturing [3] to perform various digital functions. It comprises an array of Programmable Logic Blocks (PLBs) and reconfigurable interconnections, allowing for the implementation of complex combinatorial functions or simple logic gates.

The configuration of an FPGA can be accomplished through the use of a Hardware Description Language (HDL), such as VHDL or Verilog. An Electronic Design Automation (EDA) tool takes the specified design and generates a bitstream, which is then loaded onto the internal volatile memories and used to control the interconnections.

While FPGAs can be used to implement the same functions as an Application-Specific Integrated Circuit (ASIC), they are generally less efficient in terms of power consumption and performance. However, the development cost of an FPGA is much

lower than that of an ASIC, making them suitable for prototyping and low-volume production.

.

2.2.2 Digital Hardware Design Flow

The process of programming a digital hardware design onto a Field-Programmable Gate Array (FPGA) consists of several phases, including Register-Transfer Level (RTL) design, synthesis, implementation, and programming. Each phase must be subject to a Quality Assurance (QA) flow to ensure its validity. In the event of a failure during any phase, the designer must either revisit the design or revert to a previous step and recommence from that point.

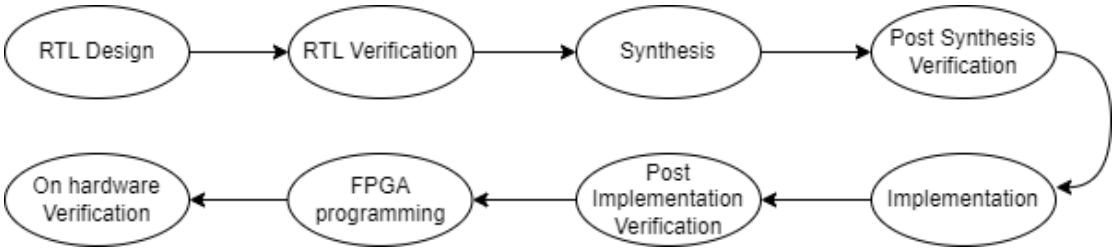

Figure 2.2.1: Digital H/W design flow.

The RTL design phase involves the creation of an abstraction that models synchronous digital circuits and describes the events that drive registers and logic. During this phase, the behavioral logic of the modules is defined, and Functional State Machines (FSMs) are implemented.

The next phase is synthesis, where the behavioral logic is transformed into a netlist. The netlist represents the design of a circuit as a list of interconnected elements, such as gates, flip-flops, adders, and multipliers. The synthesis process performs a syntax check, optimizes the logic, eliminates redundant logic, and converts the design into a netlist. Additionally, based on the target technology, some EDA tools may produce an initial estimate of the FPGA utilization, power analysis, and static timing analysis (STA).

Physical implementation follows, where the netlist is mapped to the FPGA design. This process involves the placement and routing of components on the IC blocks. The

placer aims to fit the components within the available blocks, while the router seeks the optimal routes for connecting the instantiated elements. Upon completion of the physical implementation process, the FPGA utilization, power analysis, and STA should be evaluated. The utilization should ensure that the design does not consume more resources than available on the FPGA. The power analysis should provide an estimate that meets the power requirements specified in the design. The STA should guarantee that the timing analysis is without setup and hold time violations.

The setup time is the minimum time before the active edge of the clock at which the input data line must be valid for reliable latching [4]. The hold time represents the minimum time that the data input must be held stable after the clock edge. In cases where the setup and hold times are not respected, it may result in metastability or other errors in the circuit.

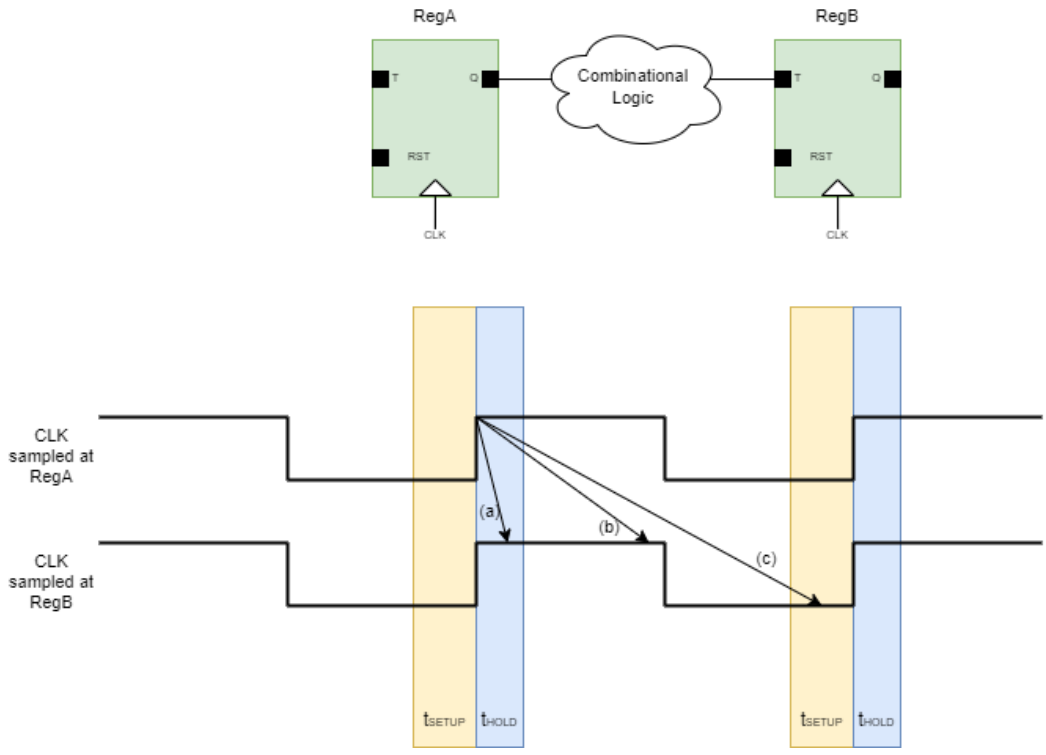

Figure 2.2.2: Setup and Hold violations.

In transmission (a), the transfer of value from one register to the next occurs during the hold time, resulting in either a successful transfer of the data to the output of RegB in the same cycle or the potential for metastability. Conversely, case (c) has the potential to result in RegB entering a metastability state in the following cycle. On the

other hand, in case (b), both the setup and hold times are respected, avoiding any potential issues.

2.2.3 iCE40UP5K and iCE40UL1K

The I^2C Bridge module will be instantiated on the ICE40UP5K and the ICE40UL1K, which are produced by Lattice Semiconductor. These FPGA families comprise an array of Programmable Logic Blocks, two oscillators (10 kHz and 48 MHz), Embedded Block RAMs, Single Port RAMs, and Programmable I/Os. [5] Each Programmable Logic Block contains eight Logic Cells, with each Logic Cell incorporating a 4-input Look-Up Table (LUT) that drives a Flip-Flop or the Logic Cell Output directly. The ICs can be configured for industrial operation and feature a power supply voltage of 1.2V, 2.5V, or 3.3V, and are capable of operating at temperatures ranging from -40 to 100°C.

The ICE40UL1K is part of the iCE40 UltraLite family and includes 1248 Logic Cells and 36 Programmable I/Os. Its dimensions are 1.409mm x 1.409mm and it has an estimated cost of 3-4 euros. The ICE40UP5K is part of the iCE40 UltraPlus family and includes 5280 Logic Cells and 60 Programmable I/Os. Its dimensions are 2.15mm x 2.15mm and it has an estimated cost of 10 euros. The ICE40UL1K will be evaluated using an iCE40 UltraLite Breakout board, while the ICE40UP5K will be evaluated using an UltraPlus Breakout board.

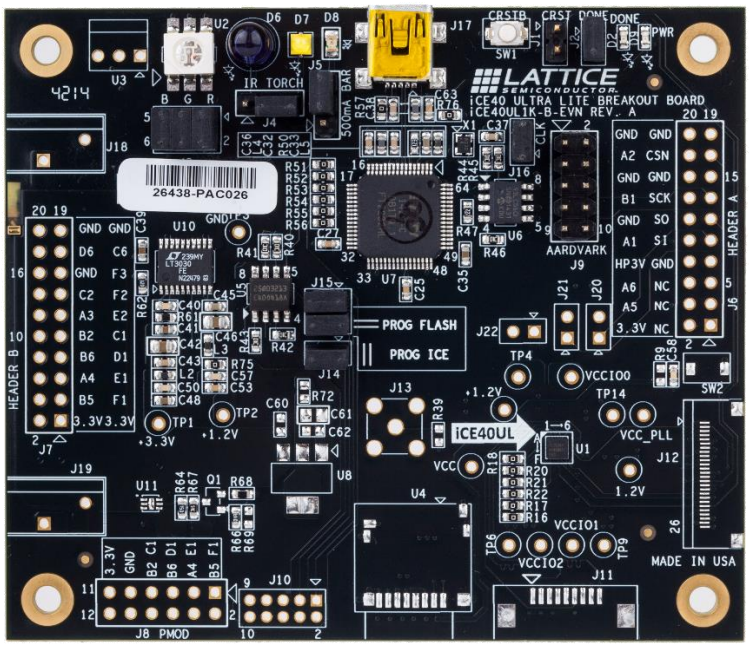

Figure 2.2.3: iCE40 UltraLite Breakout board

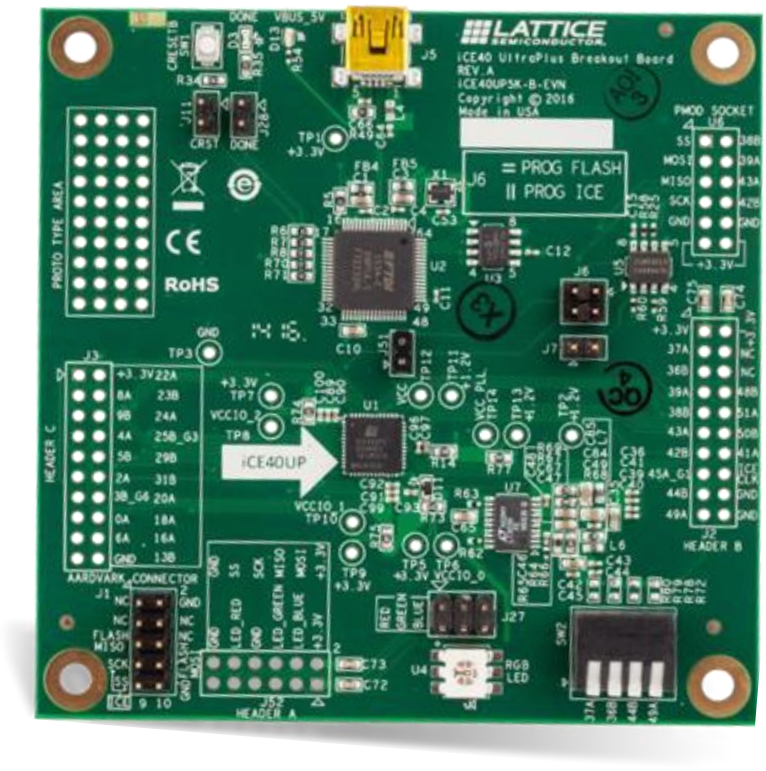

Figure 2.2.4: iCE40 UltraPlus Breakout board

2.2.4 iCEcube2

The iCEcube2 software tool is necessary to run the Synthesis and Physical Implementation flow and produce a Bitstream for the targeted FPGAs. The software

supports the Synplify Pro synthesis tool and the Lattice Synthesis Engine for synthesis. The resulting netlist and Physical Constraints File (PCF) are then fed into the Placer and Router process, following which the Bitstream is ready for programming onto the device.

2.3 Software Development

2.3.1 Device Driver

Each peripheral device requires specific initialization operation and management. Each software that communicates with it has to handle it, increasing its complexity and making it less portable. To solve this, an intermediate layer between the target device and the other parts of software has been introduced, the device drivers. [6] They act as translators providing to the rest of the software an Application Programming Interface and handle the device based on its specific needs. For instance, to send a message via a peripheral RF module, only a send data command is needed, as the serial bus, gpio and other are handled by the driver.

2.3.2 Zephyr RTOS

A Real Time Operating System (RTOS) is a specialized OS designed for real-time applications with critical timing constraints. It is widely used in various industries, such as automotive and aircraft, and is particularly popular in embedded systems due to its light design and ability to operate with low power and minimal resources.

One of the most well-known RTOSs is the Zephyr RTOS, which supports multiple boards, has multiple drivers implemented, and is licensed under Apache 2.0. The Zephyr RTOS is highly configurable, both at the software level (defining the code to be inserted and parametrizing the values) and the hardware level (by constructing the device tree). The system comes with its own meta-tool, called "west," to initialize a project, fetch desired external modules, compile and flash an MCU.

2.4 Testing Tools and Environments

2.4.1 Verilog Simulation

Icarus Verilog is an open-source tool for Verilog simulation and synthesis. It can compile Verilog (IEEE-1364) source code into various formats for simulation and generate a netlist for synthesis. [7] The resulting file can be simulated using the "vvp" command. The combination of iverilog and vvp is a useful tool for evaluating Verilog code, particularly during the pre-synthesis flow, as special libraries are not required. The tool is compatible with both Windows OS and popular Linux distributions (such as Ubuntu 18.04 and Linux Mint 20.04), making it easily integrable into automated testing processes.

During simulation, desired signals can be recorded in a Value Change Dump (VCD) file, which can be plotted using the gtkwave tool, a waveform analyzer for digital and analog data. This tool is not used for production purposes (such as for stable code and verification tests) but rather for debugging purposes.

2.4.2 Docker

Docker is a software platform that facilitates the building, testing, and deployment of applications by virtualizing containers on a configured operating system. A container is a controlled and isolated environment that is separated from other processes on the host machine. [8] It is created from a Docker image, which contains the configuration and information necessary for generating the container. The recipe for constructing the Docker image is contained in a DockerFile.

2.4.3 CI/CD

Continuous Integration, Continuous Delivery (CI/CD) are automation tools used for testing and delivery processes. By continuously integrating code changes after automatic testing and verification, the risk of code conflicts and bug insertion is reduced. Continuous Integration manages the Quality Assurance flow by keeping logs and tracking changes, allowing for seamless updates to be merged into the master branch. Continuous Delivery takes over after the CI flow, deploying the infrastructure to testing or production environments.

Chapter 3 Application Description

3.1 Architecture

The I²C Bridge module can be separated in three major parts:

1. I²C Slave Interface (communicating with MCU).
2. Register File
3. Interfaces to Peripheral devices

Four pins are provided for on board configuration of the I²C address, achieving the existence of 16 modules on the same bus. Those represent the Least Significant Bits of the module, with the three MSB to be statically defined in it (3'b010).

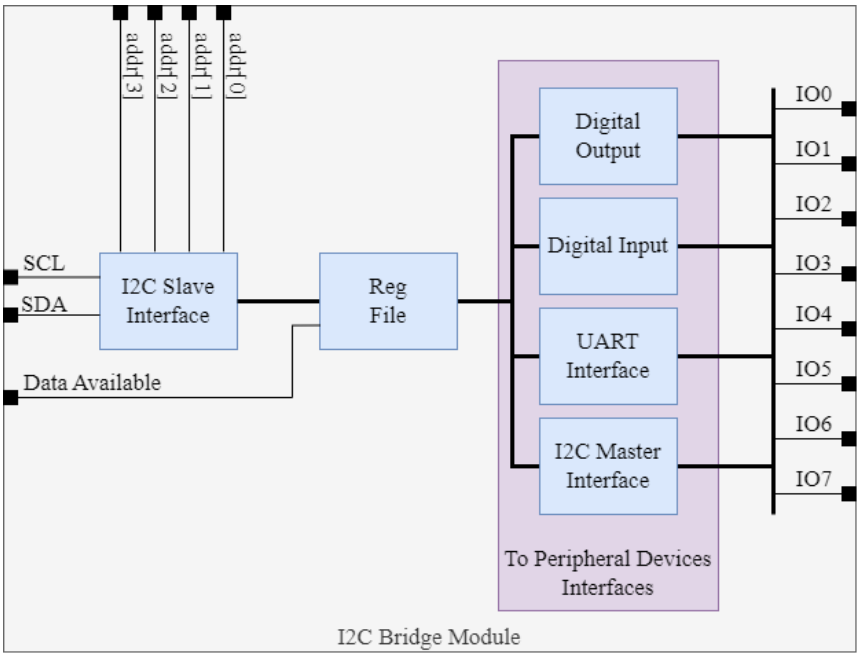

Figure 3.1.1: Top Level Diagram

The I²C Slave Interface is responsible for managing the communication with the MCU (Master) Device and facilitating the transfer of data to and from the Reg File. The Reg File serves as a repository for the configuration data that determines the behavior of the module and buffers the data during both transmission and reception.

Additionally, four interfaces have been implemented for communication with Peripheral Devices, with only one being activated at a time based on the configuration data received, allowing control over the desired Input/Outputs.

3.2 Register Map

The Master Device communicates with the I^2C bus and accesses a specific register for transmitting and receiving data. In order to perform this operation effectively, it is necessary for the it to have knowledge of the register addressing, the meaning of the internal operations, and the read and write permissions. This information is provided in the Register Map Table (Table 3.2.1).

Table 3.2.1: Register File Components.

Address	Description	R/W Permissions	Data
0x0	Interface Configuration	Read/Write	[1:0] Interface • 2'b00: Digital Output • 2'b01: Digital Input • 2'b10: I^2C • 2'b11: UART [4:2] Speed If Protocol is UART: • 3'b000: Baud Rate: 300 bps • 3'b001: Baud Rate: 1200 bps • 3'b010: Baud Rate: 4800 bps • 3'b011: Baud Rate: 9600 bps • 3'b100: Baud Rate: 19200 bps • 3'b101: Baud Rate: 38400 bps • 3'b110: Baud Rate: 57600 bps • 3'b111: Baud Rate: 1152000 bps If Protocol is I^2C: • 3'b000: Standard Mode: 100 kbps • 3'b001: Fast Mode: 400 kbps

Address	Description	R/W Permissions	Data
			• 3’b010: Fast Plus Mode: 1 Mbps • 3’b011: High Speed Mode: 3.4 Mbps
0x1	Data Size Configuration	Read/Write	[2:0] Data Rx Size packet: • 3’b000: 1 byte • 3’b001: 2 bytes • 3’b010: 3 bytes • 3’b011: 4 bytes • 3’b100: 5 bytes • 3’b101: 6 bytes • 3’b110: 7 bytes • 3’b111: 8 bytes [5:3] Data Tx Size packet: • 3’b000: 1 byte • 3’b001: 2 bytes • 3’b010: 3 bytes • 3’b011: 4 bytes • 3’b100: 5 bytes • 3’b101: 6 bytes • 3’b110: 7 bytes • 3’b111: 8 bytes [7:6] Data Tx Size packet: • 2’b00: 1 byte • 2’b01: 2 bytes • 2’b10: 4 bytes

Address	Description	R/W Permissions	Data
			• 2'b11: 8 bytes
0x2	I^2C Slave Address Configuration	Read/Write	[0] I^2C Read Operation [7:1] I^2C Slave address
0x8	Receive Data Byte[0]	Read Only	
0x9	Receive Data Byte[1]	Read Only	
0xA	Receive Data Byte[2]	Read Only	
0xB	Receive Data Byte[3]	Read Only	
0xC	Receive Data Byte[4]	Read Only	
0xD	Receive Data Byte[5]	Read Only	
0xE	Receive Data Byte[6]	Read Only	
0xF	Receive Data Byte[7]	Read Only	
0x10	Transmit Data Byte[0]	Read/Write	
0x11	Transmit Data Byte[1]	Read/Write	
0x12	Transmit Data Byte[2]	Read/Write	
0x13	Transmit Data Byte[3]	Read/Write	
0x14	Transmit Data Byte[4]	Read/Write	
0x15	Transmit Data Byte[5]	Read/Write	
0x16	Transmit Data Byte[6]	Read/Write	
0x17	Transmit Data Byte[7]	Read/Write	
0x18	Slave Register Byte[0]	Read/Write	
0x19	Slave Register Byte[1]	Read/Write	
0x1A	Slave Register Byte[2]	Read/Write	
0x1B	Slave Register Byte[3]	Read/Write	
0x1C	Slave Register Byte[4]	Read/Write	
0x1D	Slave Register Byte[5]	Read/Write	
0x1E	Slave Register Byte[6]	Read/Write	
0x1F	Slave Register Byte[7]	Read/Write	

Notes:

1. The I2C Read Operation Bit (Bit 0) in the I2C Slave Address Register (Reg 0x2) is used to initiate an I2C read from the slave device to the specified register (Slave Register). The bit will return to 0 once the read operation is completed.

2. The configuration registers, which include the Interface Configuration, Data Size Configuration, and I^2C Slave Address Configuration, support 1-byte read and write operations. When writing to the Transmit Data (0x10) or Slave Register (0x18), the I2C Bridge Module requires the specified number of bytes as defined in the Data Size Configuration. Similarly, it will provide the appropriate number of bytes during transmissions from Receive Data (0x8), Transmit Data (0x10), or Slave Register (0x18).

3.3 Device Operation Flows

3.3.1 Basic Flow

The primary utilization of the device can be separated into three key components: Initialization, Writing Data, and Reading Data. During the Initialization process, certain parameters are written that typically do not change during operation. For example, the interface used is typically determined by the board design and remains constant. The interface is set only once during the Initialization process, thus eliminating the need for reconfiguration. Similarly, if the interface is UART and connected to a specific device, the baud rate would not be modified during operation. Based on the application, data may then be written or read. Some parameters may require reconfiguration, such as the data size or the I^2C address of the slave device if the I^2C protocol is in use. Once reconfiguration is complete, it is time to write or read data.

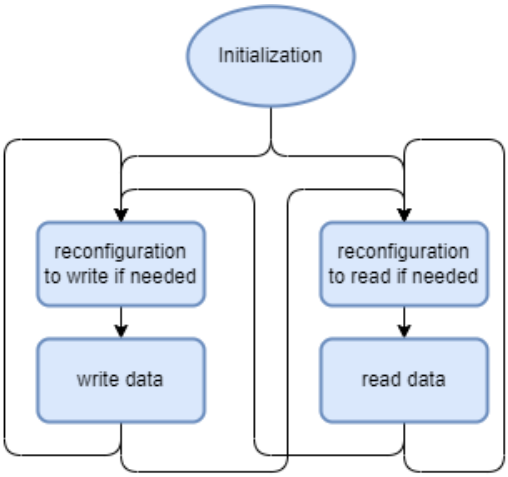

Figure 3.3.1: I²C Bridge Basic Operation Flow

3.3.2 Digital Output Operation Flow

The operation of Digital Output consists solely of writing a single byte to the output when necessary. It does not involve a read data component or its configuration. During the initialization process, the protocol is set to Digital Output and the Write Data Size is configured to 1 byte. Whenever the output value needs to be changed, a command must be sent to address 0x10 with the appropriate bitstream. The MCU can then verify the written value by reading from the same register.

3.3.3 Digital Input Usage Flow

The Digital Input process involves a straightforward initialization step and reading a byte that represents the input bitstream. The initialization stage entails setting the interface to Digital Input and specifying the Read Data Size as 1 byte. With no further configuration required, the external master can simply issue an I²C command targeting the Receive Data address (0x8) as the final step of the initialization process. Whenever a new input sample is required, an I²C read operation will retrieve the input bitstream.

3.3.4 UART Operation Flow

To communicate with a peripheral device that has a UART interface, the interface must first be configured as UART. As the baud rate of most peripheral devices is constant, the interface configuration register can be set once during the initial setup. However, the Data Size may not be constant, so it should be configured prior to the read or write operation. To initiate a write operation, the Transmit Data Size should be set as desired and an I^2C command should be sent to address 0x10, containing the desired data. The peripheral device will receive the data. In the case of a read operation, the Data Receive Size should be set prior to expecting data to be sent from the peripheral device to the bridge device. After a short delay, the MCU should access address 0x8 of the bridge device and retrieve the received data.

3.3.5 I^2C Operation Flow

The initialization process for the I2C operation flow involves configuring the interface with the appropriate interface and speed code. To initiate a write operation, the Data Tx Size and the Register Address size should be set if either of these parameters have changed since the last operation. Then, the target register on the peripheral device should be specified in the "slave register" register of the bridge device. If the address of the peripheral device has not been set previously, it should be set in the 7 most significant bits (MSB) of register 0x2, with the least significant bit (LSB) set to 0. Finally, the bridge device can send the data to the peripheral device by providing it to register 0x10.

For a read operation, the Data Rx Size and the Register Address size should be set. Then, the peripheral device register to be read from should be specified. To initiate the read operation, the I2C address should be provided at address 0x2, similar to the write operation, but with the LSB set to 1. This must be done each time a read operation is requested, as setting the LSB to 1 triggers it. After a brief waiting period, the data from the peripheral device will be stored in register 0x8 of the bridge device, and the microcontroller unit (MCU) can retrieve it.

Chapter 4 Digital Hardware Design

4.1 RTL Implementation

4.1.1 Util Modules

4.1.1.1 Edge Detector module

The Edge Detector module is designed to detect changes in the input signal and generate a pulse signal upon the detection of an edge. This is achieved by utilizing a flip flop circuit to capture the previous cycle value of the input signal and comparing it to the current state. If the previous cycle value is high and the current cycle value is low, a negative edge is detected and the "Neg Edge" output will be set to high. Conversely, if the current cycle value is high and the previous cycle value is low, a positive edge is detected, resulting in a high signal on the "Pos Edge" output.

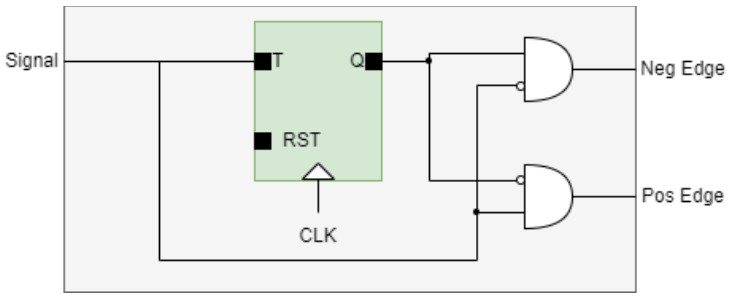

Figure 4.1.1: Edge Detector Schematic

4.1.1.2 Bidirectional Splitter module

This module serves the purpose of separating a bidirectional signal into its input and output components. It is utilized in situations where a pin of the I2C Bridge can be utilized as either an input or output, while the internal logic handles the input and output signals differently. An example of this is the I/Os to the slave device or the SDA wire of the I2C interface. The INOUT wire is connected to the desired pin, while the "To_output" signal drives the desired output and the "From_Input" signal provides the input. The "Output_En" signal enables the "To_output" signal to be transmitted to the output.

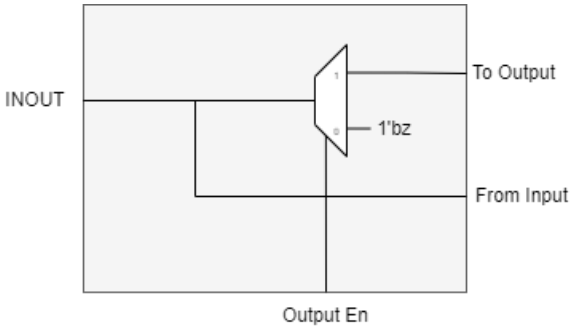

Figure 4.1.2: Bidirectional Splitter Schematic

4.1.1.3 input_synchronizer module

The input signal, originating from the external environment of the I2C Bridge module, may not be synchronized with its internal clock, which can result in metastability issues. The lack of control over the input signal may cause it to oscillate during the clock's sampling edge. To mitigate this, it has been recommended in [9] to pass the input signal through two flip-flops. The implementation of this design carries the tradeoff of increased signal latency due to the synchronization process, however, this is deemed acceptable for the needs of the project.

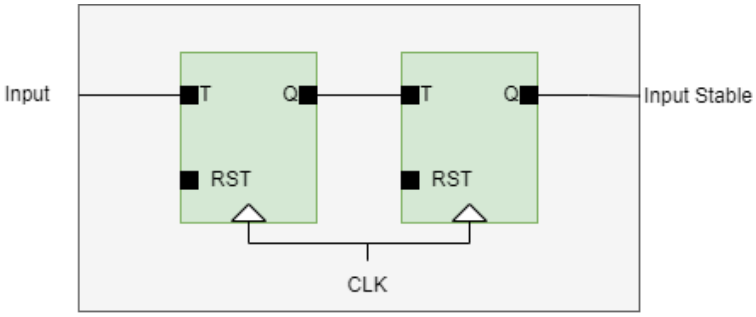

Figure 4.1.3: Input Synchronizer Schematic

4.1.1.4 debouncer module

The digital inputs that are connected to external devices may experience signal bounce when toggled. This is due to the presence of mechanical components in devices such as switches, buttons, or relays, which can introduce natural bounces. If left unfiltered, signal bounce can cause a range of issues in a digital design, including unnecessary energy consumption and multiple triggering of internal logic, rather than a single triggering event. For example, a button press may cause a counter to increment by five values instead of just one. To address these problems, it is necessary to implement a debouncing module for the digital inputs.

The debouncer module consists of several key components, including an edge detector, a counter, a multiplexer, and a flip-flop that maintain the information being transmitted to the output. Before being processed by the internal logic, digital inputs must first pass through the debouncer module for synchronization. The input signal undergoes detection of any changes in its state by passing through the edge detector, which is achieved by passing both the positive and negative edges of the input signal through an OR gate. The resulting edge-detected signal is then used to reset the counter, which is triggered by the system clock and will stop counting and generate an enable signal after reaching its maximum value, which is determined proportionally to the clock period provided during synthesis. When the enable signal is high, the input drives the flip-flop, otherwise, the output signal (current output) is fed back into the flip-flop's input.

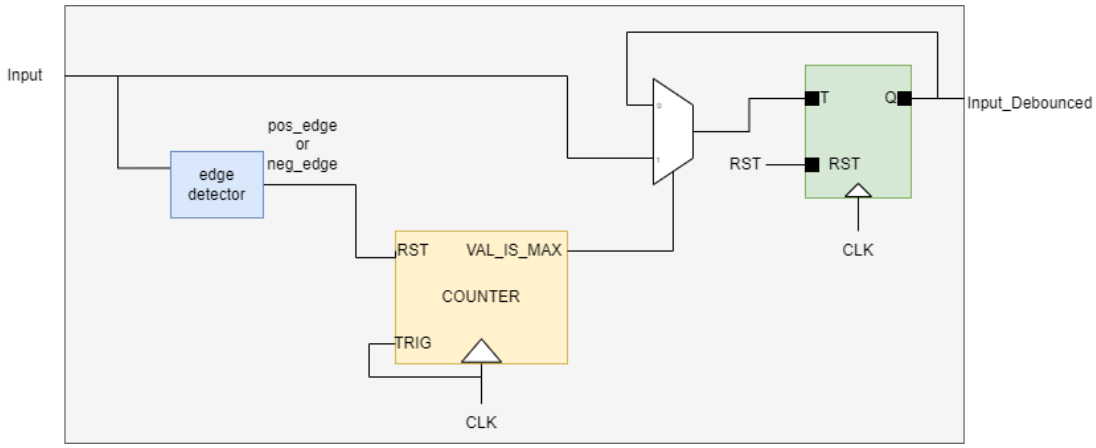

Figure 4.1.4: Debouncer Schematic

With this design, two time periods are achieved. The first period is the "calm period," in which no signal bounces occur, and the input signal is directly transmitted to the output with a delay of one clock cycle. Any transition in the input signal will reset the internal counter, thereby initiating the "non-calm" period in which the output is locked to the last changed value. This period will persist until 255 ns (the debouncing period) have elapsed since the last bounce occurred.

4.1.1.5 reg_to_serial module

The "reg_to_serial" module performs serialization of a byte, utilizing a trigger signal as input. The inputs of this module are an 8-bit Data, a trigger signal, and an enable signal, while the outputs are the serialized signal (TxD) and a "Tx_busy" signal to inform an upper-level module. The module is implemented with a finite state machine (FSM), with each state representing the serialization of a bit and an IDLE state. In the event of the enable signal being pulled down, the state will be updated to the IDLE state. Conversely, a positive edge in the trigger signal will prompt the state to transition as specified in the FSM shown in Figure 4.1.6.

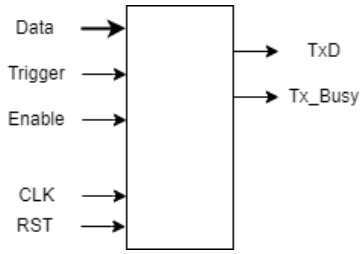

Figure 4.1.5: reg_to_serial inputs and outputs

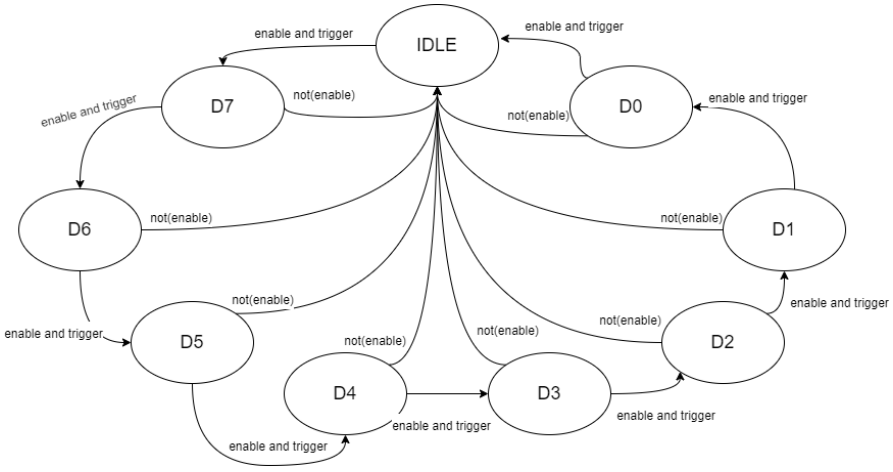

Figure 4.1.6: Serialization and Deserialization FSM

4.1.1.7 serial_to_reg module

The "serial_to_reg" module serves to deserialize an incoming signal. The module employs a finite state machine (FSM) that has the same states and transition conditions as the FSM used in the "reg_to_serial" module. However, rather than writing data at a trigger signal, the "serial_to_reg" module samples the incoming signal and builds the output data. When the FSM reaches its final state, the "data_valid" signal is activated and remains high until the module begins a new sampling process.

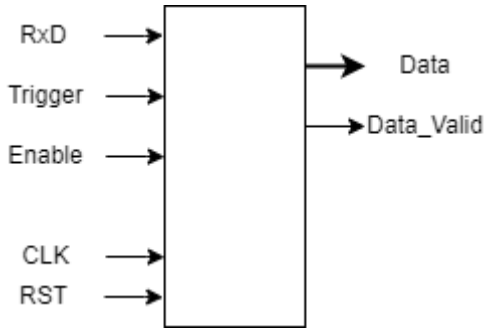

Figure 4.1.7: serial_to_reg inputs and outputs

4.1.1.8 i2c_read_byte module

The I^2C Read Byte module performs the task of reading a single byte of data. It comprises of an internal instance of the serial_to_reg module, responsible for executing the read operation. The module operates under the control of a FSM with three distinct states: IDLE, READ_BYTE, and WRITE_ACK. The inputs to the module include the SDA signal, scl_trig_high, scl_trig_low, should_comp, data_to_comp, enable, clock, and reset. The outputs of the module include the read data, a valid_data signal indicating the availability of the read data, a read_busy signal, and the allocate_sda signal for acknowledging the read operation.

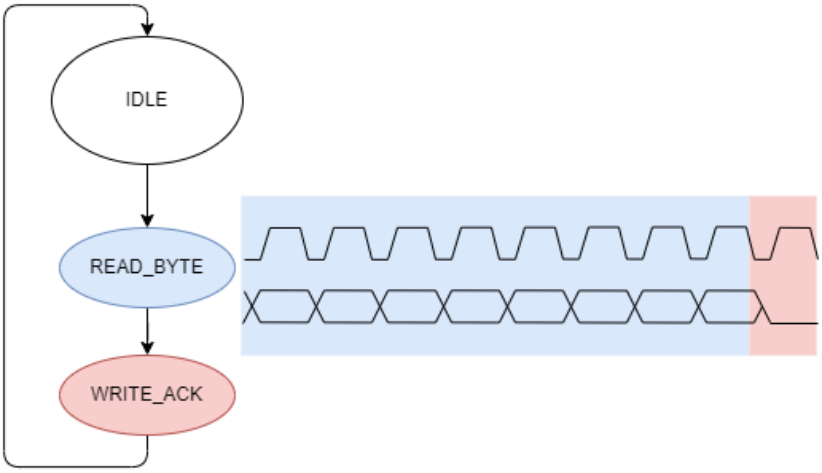

Figure 4.1.8: i2c_read_byte flow and I^2C lines.

The "scl_trig_high" and "scl_trig_low" are signals that are to be provided by a higher-level module. The "scl_trig_high" signal indicates that the SDA signal is high and should be sampled. This signal serves as the trigger input for the serial_to_reg module. The "scl_trig_low" signal indicates that the SDA signal is low and should be written. The FSM utilizes this signal to determine the completion of the I2C read phase (in conjunction with the "data_valid" signal of the serial_to_reg module). The "allocate_sda" signal is raised when the I2C read phase is completed and falls on the next positive edge of this signal, which indicates the completion of the acknowledge phase.

While the FSM is in the "READ_BYTE" state, the serial_to_reg module is enabled and disabled in all other states. The "valid_data" signal, which is exposed to the higher-level module, informs when valid data is available for consumption, while the i2c_read_module continues to transition through the "WRITE_ACK" state.

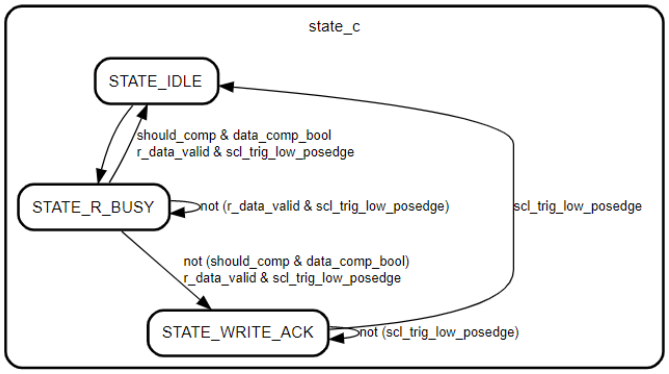

Figure 4.1.9: i2c read byte FSM.

The “should_comp” and “data_to_comp” signals are utilized by the higher-level module to validate if the expected data has been received before proceeding with the acknowledgment step. This feature is particularly useful in the I2C Slave module, where it is necessary to confirm the reception of a specific address before continuing. If the “should_comp” signal (flag) is not activated, this comparison function is inactive, and the FSM will proceed to the acknowledgment step regardless of the received data.

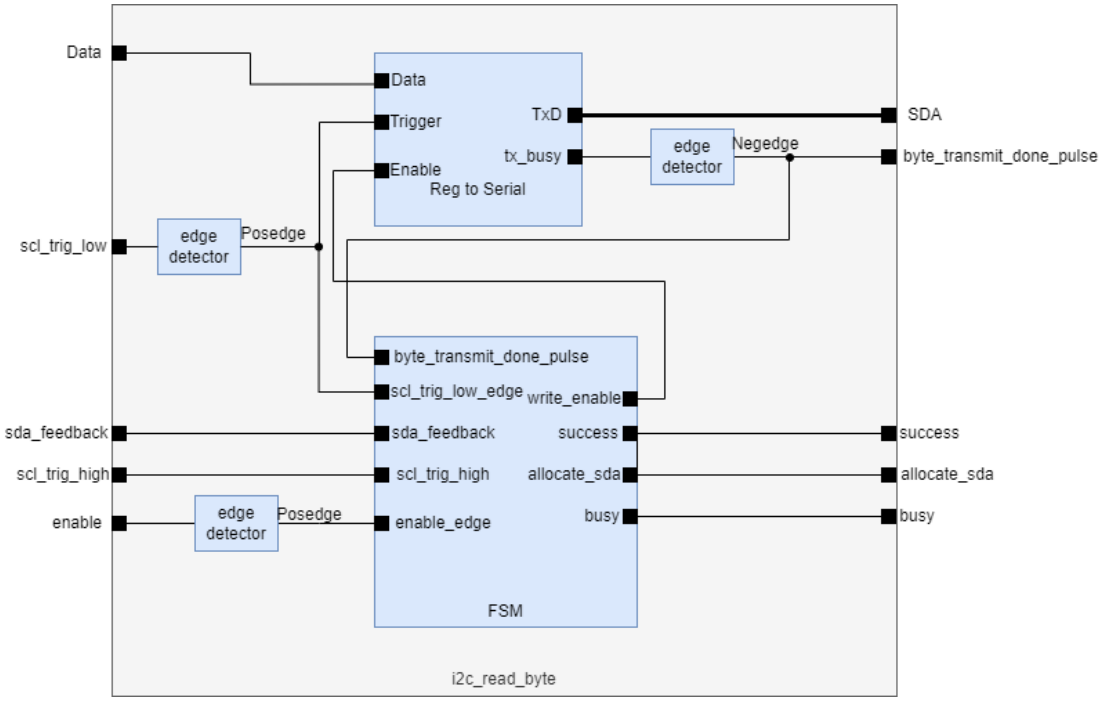

Figure 4.1.10: i2c read byte schematic.

4.1.1.9 i2c_write_byte module

The module "i2c_write_byte" serves the purpose of data writing and acknowledging its successful transmission. It incorporates a "reg_to_serial" module to serialize the input data from the higher-level module and transmit it via the SDA signal. The module operates through a finite state machine (FSM) with four states: IDLE, WRITE_BYTE, WAIT_ACK, and POST_ACK. The inputs to the module are the data to be written, the signals "scl_trig_high" and "scl_trig_low", the signal "sda_feedback", and the enable, clock, and reset signals. The outputs of the module include the SDA signal, the signal "byte_transmit_done_pulse", indicating successful transmission, the signal "allocate_sda", and the busy signal, indicating the module's operational status.

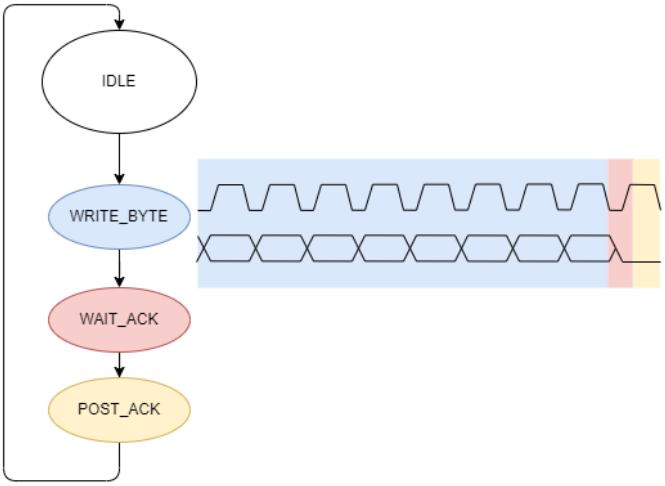

Figure 4.1.11: i2c_write_byte flow and I^2C lines.

The scl_trig_high and scl_trig_low signals are supplied by the higher-level module to the i2c_write_byte module, just like in the i2c_read_byte module. However, in this module, the reg_to_serial serves as a write module rather than a sampling module. Thus, the trigger for the reg_to_serial module is derived from the positive edge pulse of the scl_trig_low signal. This signal is also utilized by the FSM to determine the exit condition during the evaluation of the acknowledgement. The scl_trig_high signal is used by the FSM for the purpose of sampling the acknowledge.

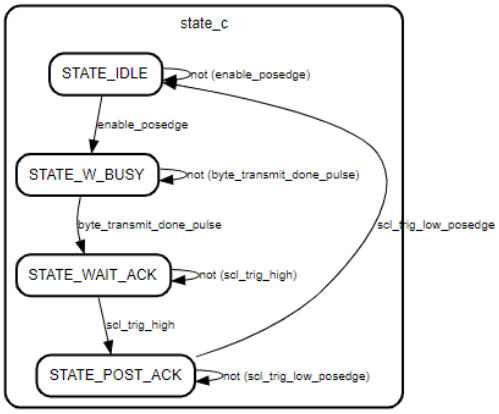

Figure 4.1.12: i2c write byte FSM.

While the FSM is in the WRITE_BYTE state, it enables the reg_to_serial module, allowing it to drive the SDA signal. In the WAIT_ACK state, the FSM waits for the byte transmission to be completed by the reg_to_serial module, and then samples the SDA signal when the scl_trig_high signal is high. If the feedback received from the other device is as expected, the success signal is raised. The FSM then transitions to

the POST_ACK state and waits for the positive edge of the scl_trig_low signal to return to the idle state and drive the busy signal low.

Similar to the i2c_read_byte module, the done signal of the reg_to_serial module is made available to the higher-level module to assist with speeding up any operation. The output provided by this module is in the form of a pulse, rather than the tx_busy state, as an edge detector is usually attached to detect state changes and the tx_busy output is followed by an edge detector.

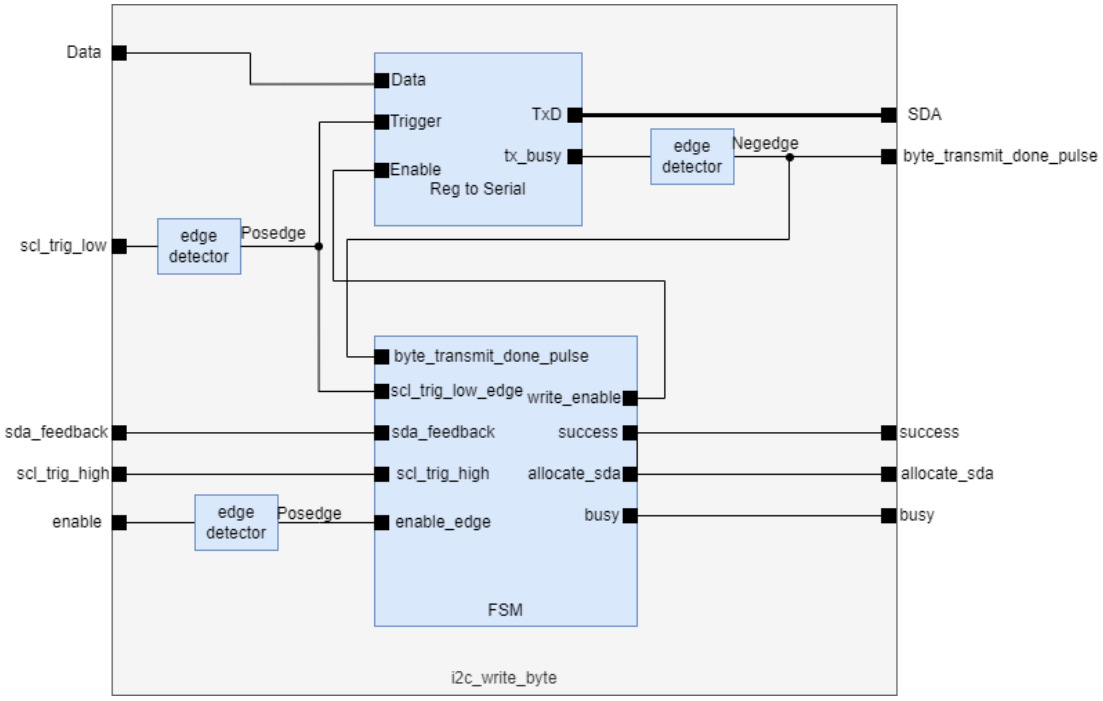

Figure 4.1.13: i2c_write_module schematic

4.1.1.10 i2c_start_stop_detector module

The start_stop_detect module is responsible for detecting the edges of the SDA signal in conjunction with the SCL signal being high. The positive edge of the SDA signal represents the stop of the I2C transmission, while the negative edge represents the start, as depicted in Figure 4.1.14.

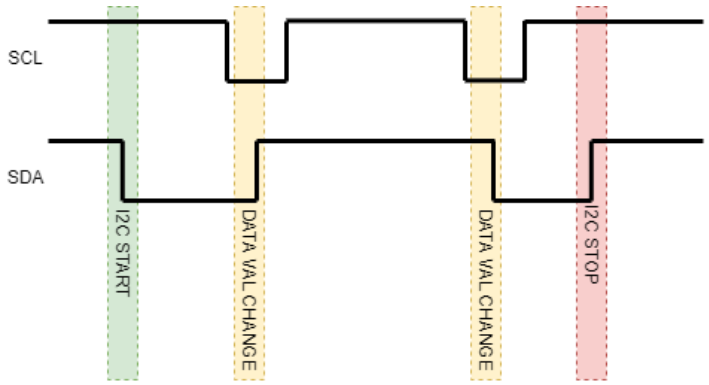

Figure 4.1.14: I^2C Start Stop conditions.

To implement this functionality, an edge detector has been instantiated, which then drives two AND gates. These AND gates enable the i2c_stop and i2c_start signals only when the SCL signal is high. This ensures that the detection of the start or stop of an I2C transmission is correctly synchronized with the SCL signal.

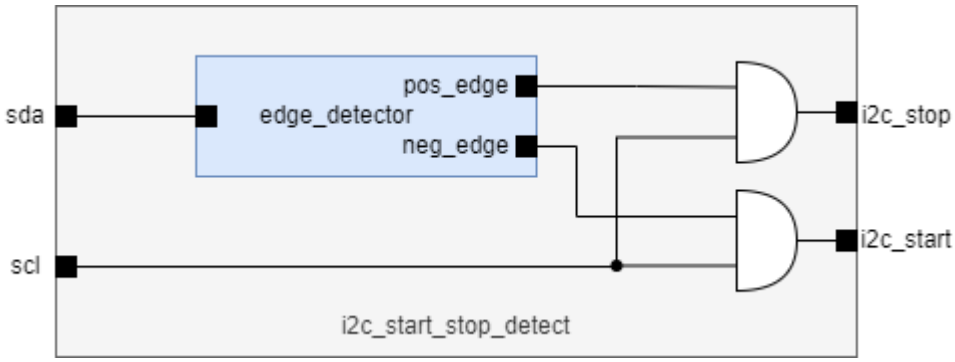

Figure 4.1.15: i2c_start_stop_detect schematic

4.1.1.11 enable_after_priority module

The “enable_after_priority” module provides a signal to the higher-level module to indicate when a trigger event has been received and another signal with higher priority is not currently active. In the event that the priority signal is high at the time the trigger is received, the module waits for the priority signal to fall before activating the enable signal. A delay of one cycle is introduced between the fall of the priority signal and the activation of the enable signal, to ensure that the higher-level module can detect an edge in the combined signal of the two.

This functionality is achieved through the implementation of a finite state machine with three states: IDLE, WAIT, and UPDATE. If the trigger is received and the priority signal is low, the state changes from IDLE to UPDATE, where the enable signal is raised. If the priority signal is high at the time of the trigger, the state changes to WAIT, and the module waits for the priority signal to fall before transitioning to the UPDATE state.

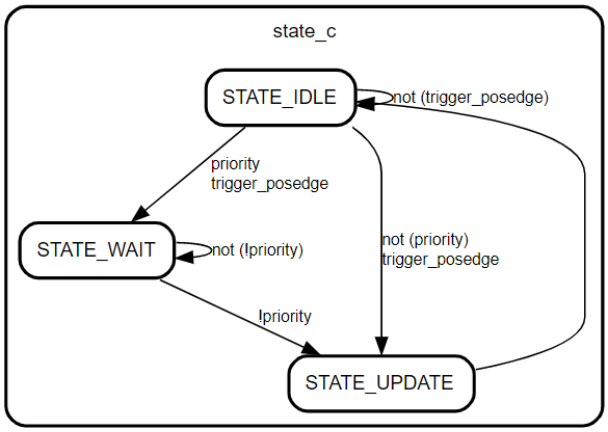

Figure 4.1.16: enable_after_priority FSM

4.1.2 Register File Modules

The Register File consists of four separate partitions, including the Configuration Register File, the Receive Data Register File, the Transmit Data Register File, and the Slave Registers Register File. The Configuration Register File contains information that pertains to the device's operation, such as the protocol, speed, data size, and the I2C Slave address. The Receive Data Register File holds data received from the Slave Module, the Transmit Data Register File holds data intended for transmission to the Slave Module, and the Slave Registers Register File holds the register addresses of an I2C Slave device (this is utilized only if the protocol is I2C). The Transmit Data Register File and the Slave Registers Register File are implemented using the same module, known as the "data_from_master" module, as they both manage data sent by the Master Device. On the other hand, the Receive Data Register File is implemented using the "data_from_slave" module.

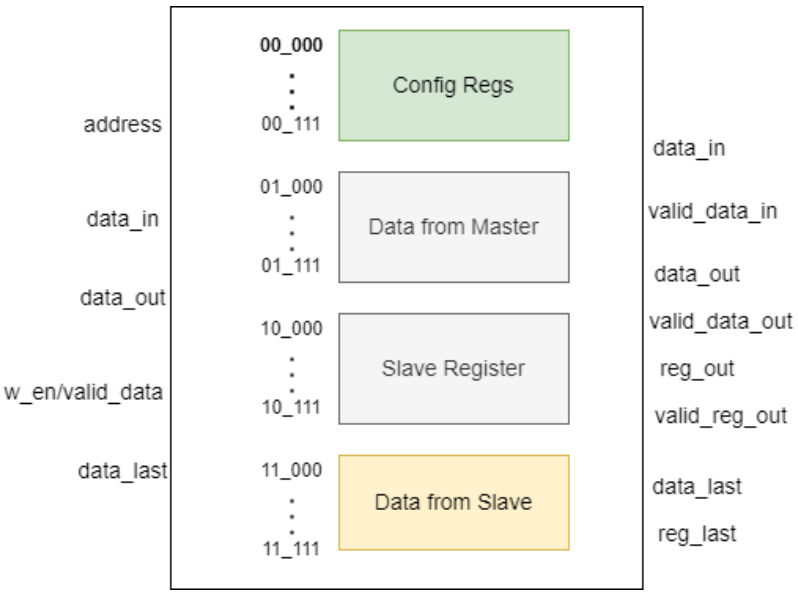

Figure 4.1.17: Register File Abstraction and Inputs/Outputs.

4.1.2.1 Interfaces and Reg File data Algorithm

The communication between the interfaces and the register file requires the implementation of a specific communication protocol. The communication is bidirectional, with data being transferred from the interfaces to the register file and vice versa. The main distinction between the two modules is that the interfaces must communicate with external modules and thus must serve the protocol in a timely manner, while the register file only serves the interfaces and may wait.

The communication from the interface to the register file involves the use of the "w_en" and "data_last" signals, as well as the "data" signal. The "w_en" signal triggers the register file module to write the provided "data" to its registers. If the current register address has reached its limit, the "data_last" signal is raised, signaling the interface to stop receiving data.

The communication from the Register File to the interface is characterized by the presence of three signals: data_available, data_req, and transmit_done. Upon receipt of data from another module, the Register File raises the data_available signal to notify the interface that it has data available for transfer. The interface then initiates the data transmission process. When the interface has completed the transfer of a single byte and requires the next, it raises the data_req signal, causing the Register File to update the data. If the interface terminates its data transmission process, it raises the transmit_done signal, signaling the Register File to reset its counters.

The data from the master module differs from the data from the slave module in that the former requires the consideration of an address received from the input. This is due to the fact that data from the master module is transmitted using the I^2C protocol, which specifies the writing of data to specific registers. As such, the address is an input signal for the data from the master module, while in all other cases the addresses are determined by internal counters that are incremented or reset based on the signals described by the communication protocol. Additionally, the master device may need to read the value of a register, and as such, an output data signal is required to provide the value based on the input address, which is the same signal used for write operations.

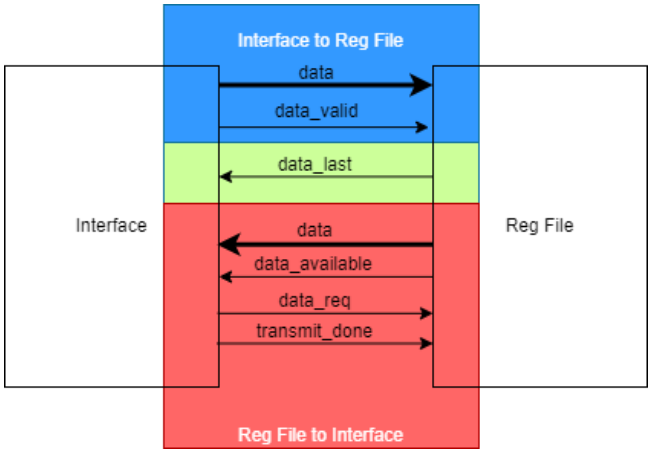

Figure 4.1.18: Reg File to Interface Communication Signals

4.1.2.2 config_regs module

The “config_regs” module is responsible for compliance with the restrictions imposed by the “data_from_master” module, however it does not need to provide data to the slave interface. Instead, it provides signals that convey configuration values. The i_value is written to the appropriate register if the w_en signal and the address match the register. If not, the register output returns to its input. The configuration values are obtained directly from the registers. Additionally, the master device has the ability to read the written value from the I2C interface through an o_value byte that provides the value of the register requested by address.

To support the feature of resetting the least significant bit of Register 0x2 (I2C Slave Address Configuration), additional logic has been implemented. The i2c_receive_done signal is passed through an enable_waiting_for_priority module to prevent triggering while the w_en signal is high. Then, it acts as the w_en signal, replacing the input address with the I2C Slave Address Configuration register address and the i_value with the last register value, but setting the least significant bit to 0.

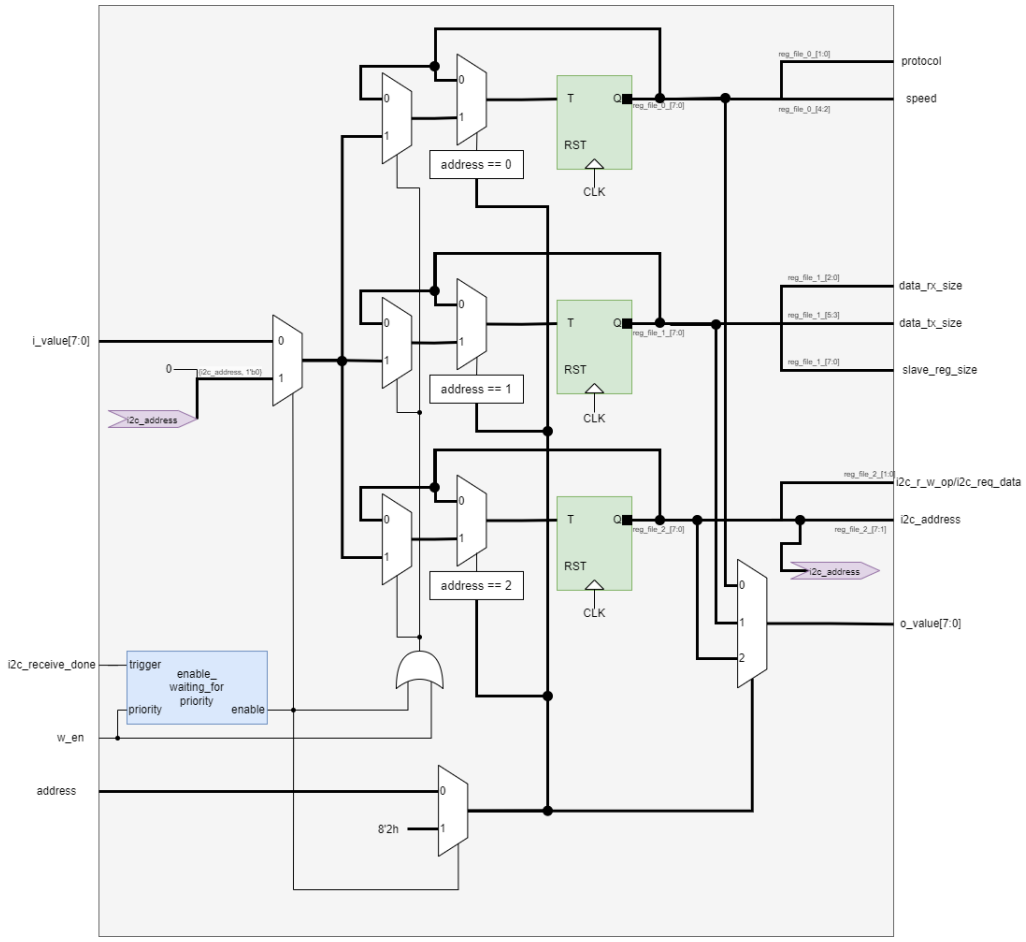

Figure 4.1.19: config_regs schematic

4.1.2.3 data from master module

The "Data from Master" module is responsible for receiving data from the master device, signaling availability of the data to the slave interfaces, and providing the data to them. This module comprises a register file and a manager for this specific function.

The register file is comprised of a simple array of eight 8-bit registers. The inputs to this file are a 3-bit master address (m_address), a 3-bit slave address (s_address), 8-bit data, and the write enable signal (w_en). The register values are updated with the incoming m_input value if the m_address provided matches the address of the register and the write enable signal is active. Otherwise, the register retains its current value. The 8-bit master output (m_output) and the 8-bit slave output (s_output) are obtained from the selected register based on the values of the m_address and s_address, respectively.

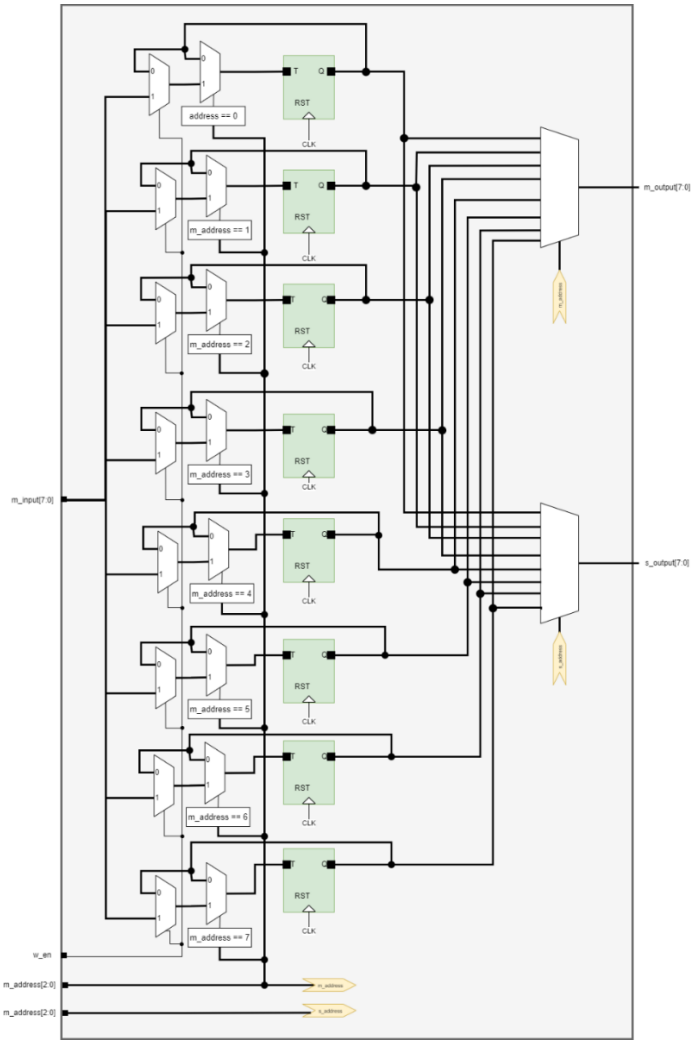

Figure 4.1.20: data_from_master schematic.

The Data from Master Manager oversees the control of the Slave Address (s_address) and the signals utilized for communication with the interfaces. The Master Data Last signal (m_data_last) is set to 1 when the Master Address (m_address) is equal to the input data size. In the event that the Master Address reaches its maximum permissible value and a Write Enable (w_en) signal is received, the data is considered full, and the Data Full edge is the positive edge output of an Edge Detector, which is driven by the AND gate result of the Master Write Enable signal (m_w_en) and the Master Data Last signal. Unconsumed data will persist after the Data Full edge as long as the Slave Address has not reached the data size, and when it does, until a Slave Data Request is received.

The next phase involves the consumption of the data by the Interface to Slave. The Slave Address starts at 0 and increases by 1 with each Slave Data Request Edge. When the Slave Address reaches the data size, the next Data Request will reset the Slave Address. If the Unconsumed Data signal is high, the Data Available output will be set to high. An exception to this rule occurs when the Slave Address does not equal the data size and a Slave Data Request is received. In this case, the Data Available signal will only fall in the next cycle and then rise in the subsequent cycle to provide a positive edge, as some modules (such as UART) receive this information as an enable signal.

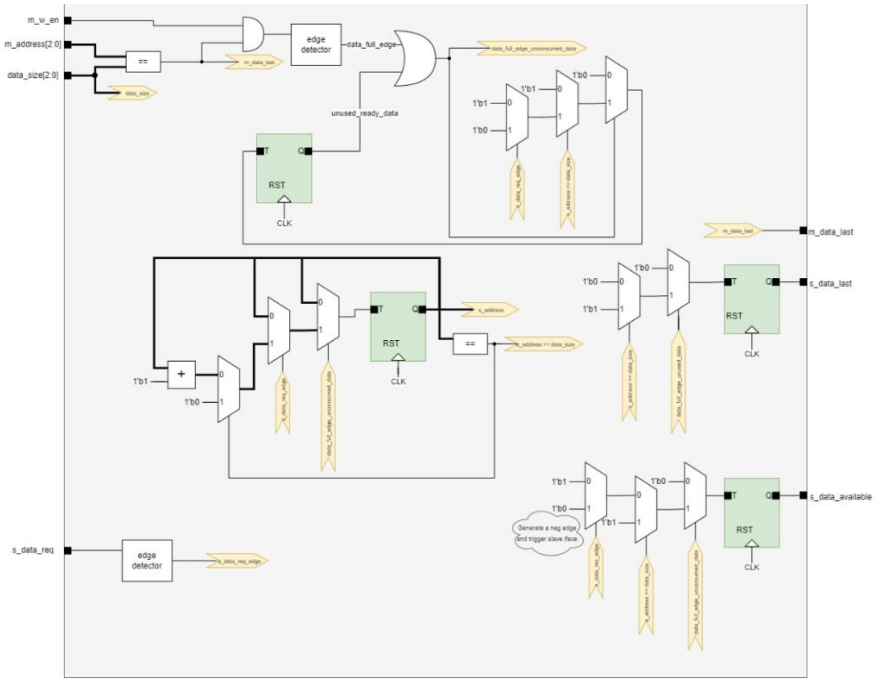

Figure 4.1.21: data from master manager

4.1.2.4 data from slave module

The Data from Slave module serves as an intermediary for the transfer of data from the Interface to Slave to the Interface to Master. The master is unable to write to it through the I2C interface, with the capability of reading the data being the only provided function.

The module comprises a register file and a manager, similar to the structure of the Data from Master module. However, there is a difference in the way the values are written to the registers, as the s_address is utilized instead of the m_address. Additionally, only the master output is accessible, as the Interface to Slave does not require the ability to read the written data.

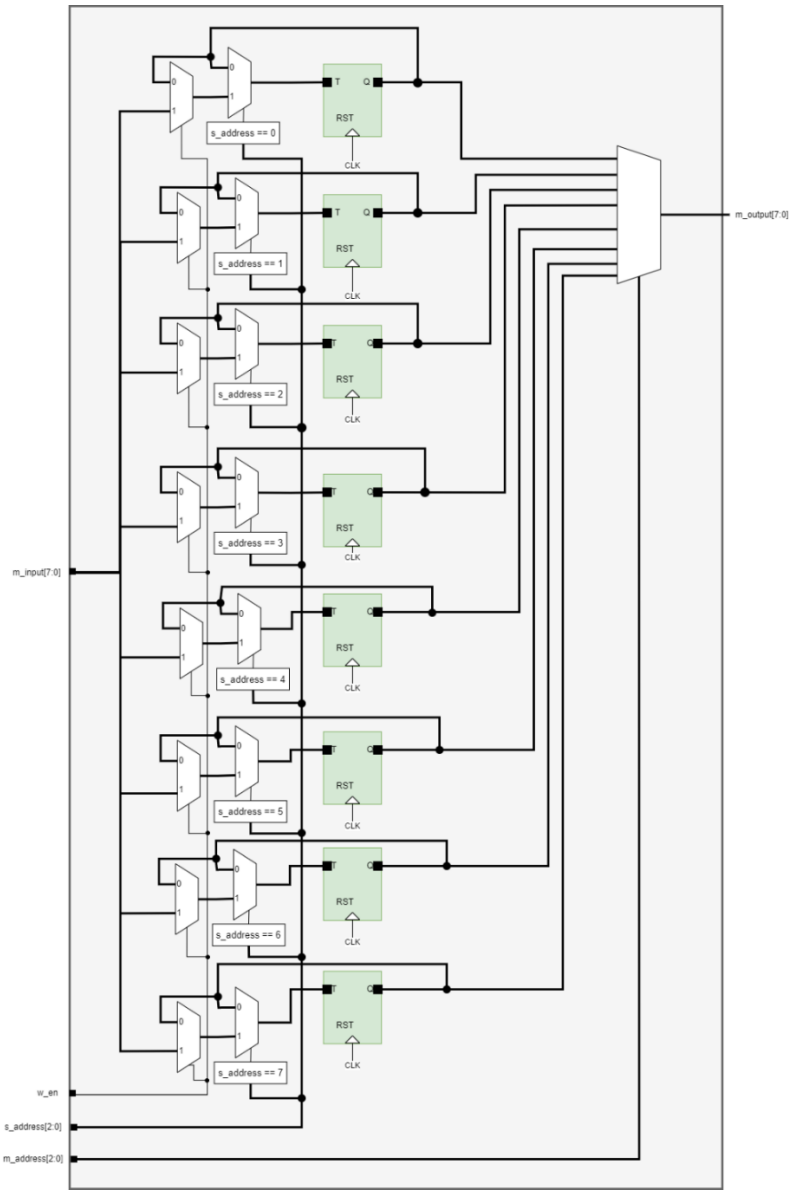

Figure 4.1.22: data from slave reg file

The Data from Slave Manager controls the communication between the Interface to Master and the Interface to Slave. It is responsible for setting the addresses, and controlling the write enable signal. The inputs to the manager include a 3-bit data size, the Slave Valid Data (s_valid_data) signal, the Master Data Request (m_data_req), and the Master Transmission Done (m_trans_done).

The s_address is incremented by 1 if the s_valid_data edge signal is received and the s_address does not equal the data_size. In the event that the m_address reaches the s_address, the s_address will be reset. The write enable signal is triggered one cycle after the s_valid_data edge signal if the data is not full. Data is considered full if the s_address and data size are equal and the s_valid_data_edge signal is received. The write enable signal returns to low when the s_address is reset.

The data available signal is set to 1 after the write enable signal is triggered. It will be reset to 0 if either a m_data_req edge or m_trans_done edge signal is received. The m_address is incremented when the m_data_req edge signal is received and the m_address does not equal the s_address. It will be reset when the m_trans_done edge signal is received. The data_last signal is set to high if the m_address is greater than or equal to the s_address.

.

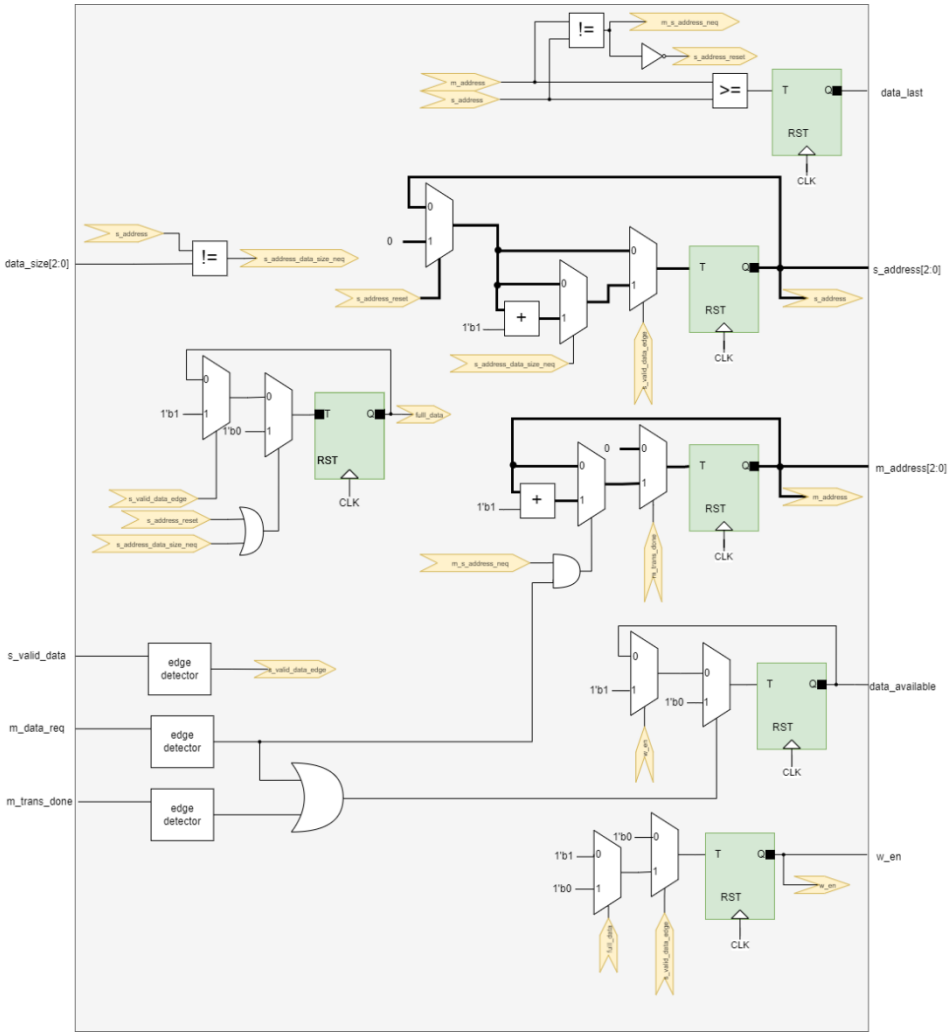

Figure 4.1.23: data from slave manager

4.1.3 Peripheral Device Interface Modules

The section describes the modules responsible for driving communication interfaces to the slave (peripheral) device. A wrapper module, known as the "to_slave_interfaces," integrates all the interfaces and facilitates their connection to the I/O and the Register File.

4.1.3.1 digital_output module

The Digital Output Module is responsible for driving 8 output signals with the input byte upon receiving the last trigger signal. It consists of two inputs, namely the trigger signal and the data byte, and two outputs, the busy signal and the I/O output wires. The busy signal is set to a high state for one cycle immediately after the receipt of the trigger signal to indicate the change of the output to a higher-level module.

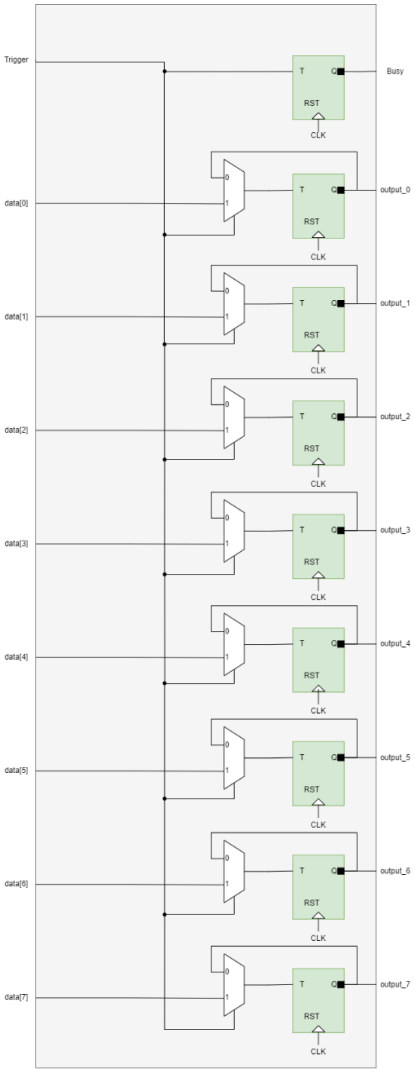

Figure 4.1.24: Digital Output Interface schematic

4.1.3.2 digital input module

The Digital Input module converts the inputs received from the I/Os into a data byte. It has an enable signal and the I/Os inputs as its inputs, and its outputs are the data byte and an update_pulse signal. The module operates as follows: If the enable signal is set to 1, each input passes through a debouncer to eliminate any bouncing that may occur. If the enable signal is not activated, the debouncers are driven by a constant zero, to reduce energy consumption when the Digital Input module is disabled. The outputs of the debouncers drive the data byte and an edge detector. The outputs of the edge detectors, positive or negative, are combined through an OR gate and drive the update_pulse signal. This means that any edge of the received inputs, after bounce filtering, will result in a pulse of the update_pulse signal, allowing the higher-level module to save the data to the register file.

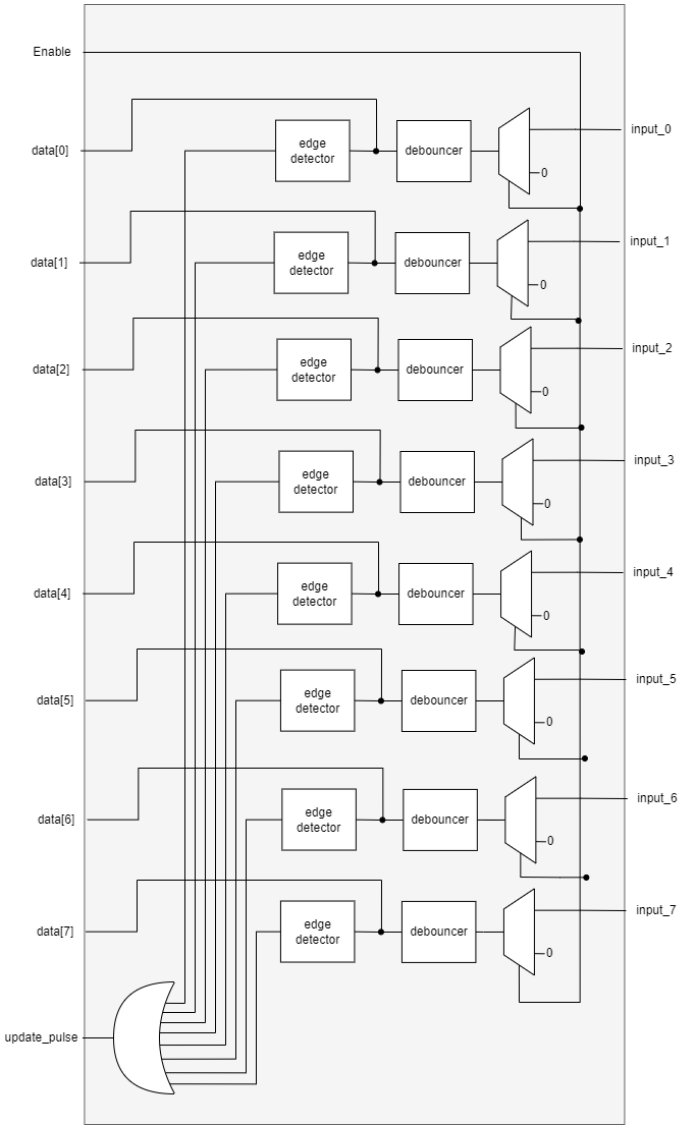

Figure 4.1.25: Digital Input Interface schematic

4.1.3.3 uart_interface

The uart_interface handles the communication of the UART bus. The module is a wrapper of the uart_transmitter and the uart_receiver who operate independently. Each module has internally instantiated a baud_rate_sample_pulse_generator for synchronization with the UART protocol specification rates.

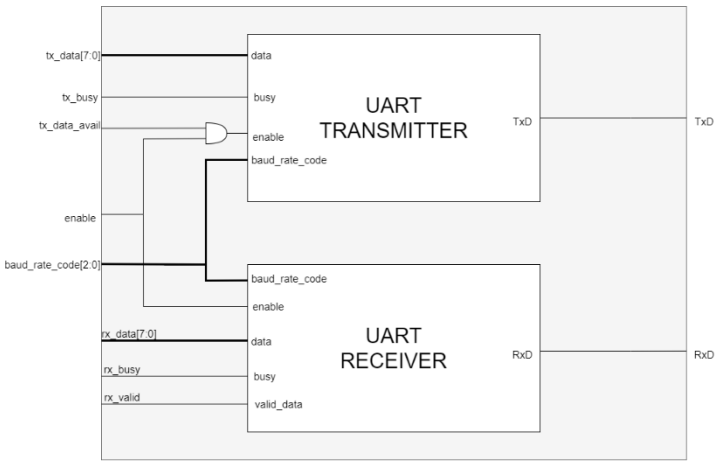

Figure 4.1.26: uart_interface Schematic.

4.1.3.4 uart_transmitter module

The UART Transmitter module is responsible for transmitting data through the TxD signal, based on the input byte. The inputs of the module include the Data Byte, the Baud Rate Code, and the Enable, Clock, and Reset signals. The outputs of the module are the TxD signal and the Tx_busy indicator.

The module implements a Finite State Machine (FSM) consisting of eleven states that represent the various stages of UART transmission, including IDLE, START_BIT, D0, D1, D2, D3, D4, D5, D6, D7, PARITY_BIT, and STOP_BIT.

Additionally, the module includes a Baud Rate Sample Pulse Generator, which provides the trigger signal at the correct time to ensure proper transmission of the data. During the START_BIT state, the TxD signal is in a low state. During the D0 to D7 states, the TxD signal takes on the value of the corresponding bit of the input data byte. The parity bit, which is calculated as the XOR function of the data bits, is transmitted during the PARITY_BIT state, and during the STOP_BIT state, the TxD signal is in a high state. The transition to the next state takes place with each pulse of the trigger signal, which is provided by the Baud Rate Sample Pulse Generator module.

4.1.3.5 uart_receiver

The UART Receiver module is responsible for capturing the RxD signal and generating a data byte through the deserialization of the UART message. It comprises an FSM with 11 states, identical to the UART Transmitter, including IDLE, START_BIT, D0, D1, D2, D3, D4, D5, D6, D7, PARITY_BIT, and STOP_BIT.

Upon detecting a negative edge in the Rx signal while in the IDLE state, the module transitions to the START_BIT state, where it awaits the appropriate time to begin sampling the data bits. During the D0-D7 states, the module samples the data bits and constructs the byte output, while also computing the parity bit using an XOR gate on the RxD value and the previous parity bit result.

In the PARITY_BIT state, the module compares the calculated parity bit with the RxD value. In case of equality, the process continues normally. However, if there is an inequality, a parity error signal is raised and the module returns to the IDLE state. Finally, the module awaits the stop bit (RxD high) and raises a functional error signal if the expected value is not captured.

4.1.3.6 baud_rate_sample_pulse_generator

This module generates a pulse to make the receiver and transmitter modules to synchronize to the UART bus. It consists of a counter whose threshold is relative to the baud rate code and the defined clock period.

Table 4.1.1: Baud Rate Code. Sampling Period and Maximum Clock periods

Code	Baud Rate	Sampling Period	Clocks per Pulse	Clocks per Pulse for 21 ns clk period)
000	300	3333333 ns	3333333 / CLK_PERIOD	158730
001	1200	833333 ns	833333 / CLK_PERIOD	39682
010	4800	208333 ns	208333 / CLK_PERIOD	9920
011	9600	104137 ns	104137 / CLK_PERIOD	4958
100	19200	52083 ns	52083 / CLK_PERIOD	2480
101	38400	26042 ns	26042 / CLK_PERIOD	1240
110	57600	17361 ns	17361 / CLK_PERIOD	826
111	115200	8681 ns	8681 / CLK_PERIOD	413

4.1.3.7 i2c_master_interface

The i2c_master_interface handles the communication with Peripheral Devices with I^2C interface. It handles the clock line, transmits data and requests-receives from the slave. It cannot operate in a bus where another master device exists.

To drive the clock line of the bus, it has a scl controller module instantiated. The scl controller consists of an FSM of 5 states, an idle state, a high first part, a high second part, a post scl high and a pre scl high part. During the IDLE state, the SCL line is not asserted, while in the first part and second part high it is set to 1 and in the post and pre scl it set to 0. When the scl controller is activated, it will jump from the idle state to the high second part and then it will follow the FSM described. The FSM is encoded in one-hot format, and each bit is exposed to the i2c master interface module.

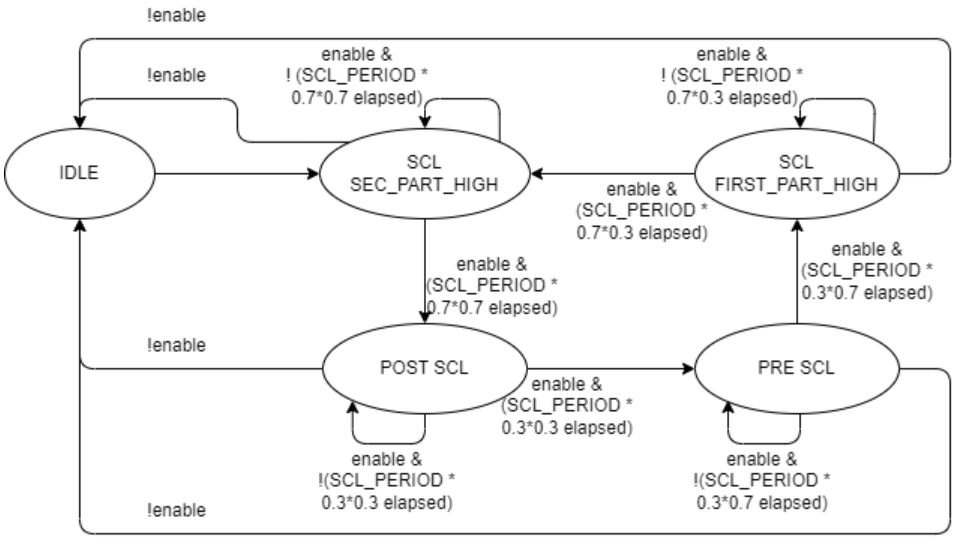

Figure 4.1.27: SCL controller FSM

The i2c_master_interface module consists of a core FSM, the scl_controller, i2c_read_byte and i2c_write_byte modules and a i2c start stop which triggers on the bus the start stop conditions. The FSM consists of 8 states: IDLE, COM_START, WRITE_I2C_ADDRESS, WRITE_REG, WRITE_BYTE, READ_BYTE, COM_STOP, RECEIVE_IDLE.

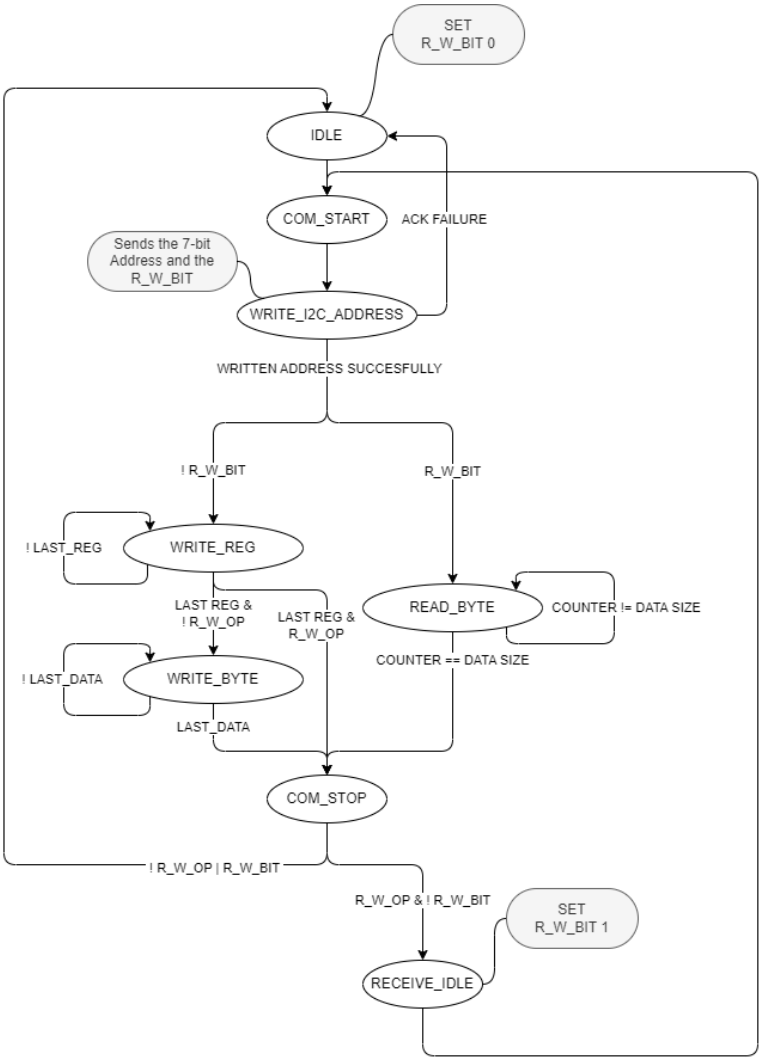

Figure 4.1.28: i2c master interface FSM

The process of write operation begins upon the receipt of an enable signal with the read/write operation (R_W_OP) input set to 0. The I2C communication protocol then proceeds to the IDLE state, followed by the COM_START state. During this state, the i2c_start_stop module is activated, driving the I2C lines and triggering a start condition.

The next stage is the WRITE_I2C_ADDRESS state, where the provided address is transmitted onto the bus and an acknowledgement is expected. In case no acknowledgement is received, it can be inferred that there is no device present with the specified address, and the process returns to the IDLE state. If an acknowledgement is received, the process continues by transmitting the register addresses until the last register signal is raised. It is then followed by transmitting the data bytes until the last data signal is set.

Finally, in the COM_STOP state, a stop condition is triggered on the I2C bus, signaling the end of the write operation. The process then returns to the IDLE state, ready for the next communication request.

The process of the read operation is more complex. When the R_W_OP input is set to 1 and the enable signal is received, the process begins. It starts with the COM_START state, followed by the WRITE_ADDRESS, where the r_w bit is set to 0 in order to initiate a write operation. If an acknowledgment is received, the module will provide the register address of the peripheral device to receive the data. Then the FSM changes to the COM_STOP state to restart the communication. The module will wait in the RECEIVE_IDLE state for a SCL period, before returning to the IDLE state. The module then moves to the WRITE_ADDRESS state after the COM_START, with the r_w bit set to 1, and begins reading until the internal counter reaches the specified DATA_SIZE threshold. A counter is implemented in the interface to keep track of the data size, instead of relying on the last_data signal, as it will be enabled after the decision of reading the last byte. Once the threshold is reached, the module will transition to the COM_STOP state to complete the read operation.

.

4.1.3.8 peripheral_interfaces_wrapper

The peripheral_interfaces_wrapper module encapsulates all the interfaces described. In addition to the interfaces, it also includes internal instantiations of I/O interfaces, along with bidirectional splitters and input synchronizers. Debouncing is applied exclusively for the Digital Input operation, and as such, this functionality is integrated within the relevant module.

The I/O0 output is enabled when the interface is either a Digital Output or when the I^2C interface requires access to the SDA line (as the I^2C interface utilizes this pin for the SDA line). In the case of the UART interface, this pin represents the Rx signal and is never allocated, as for Digital Input, it represents the first bit of the 8-bit bitstream. The I/O1 output is enabled for the UART interface (representing the Tx line), I^2C interface (representing the SCL line, which is not bidirectional), and Digital Output (representing the second bit). The remaining I/Os are utilized solely by the Digital Output interface and are enabled only when this interface is in use or for Digital Input. In addition to directing the wires to the I/Os, the slave wrapper also merges the wires

to the Reg File, as the latter uses the same signals regardless of which interface is enabled.

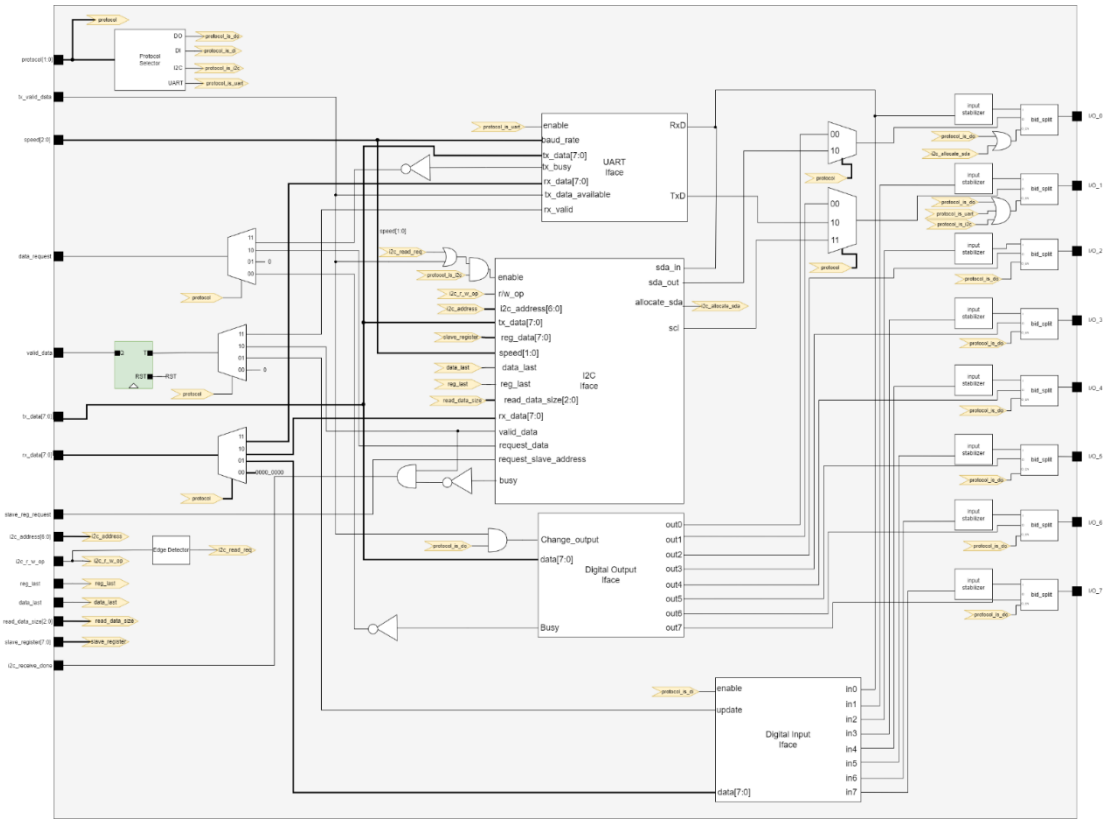

Figure 4.1.29: To Peripheral wrapper schematic

4.1.4 MCU Communication Module

The I^{2}C Slave interface is instantiated for communication with the MCU device. It comprises of the following components: a start_stop_detect module, a i2c_write_byte module, a i2c_read_byte module, a bidirectional splitter for the SDA wire, input synchronizers for the I2C wires, 3 edge detectors, and its basic finite state machine (FSM).

The inputs to the I^{2}C Slave interface include the device address, the data byte from the register file, the last data signal, the I^{2}C bus lines (SDA as inout) and of course, the clock and the reset signals. The outputs of the interface are the register address and data, the register write enable signal, the data from register file request, and the busy and transmission done signals.

Regarding the design, the SDA signal passes through a bidirectional splitter to separate the output and input. The SDA input and the SCL wires are then routed through input synchronizers, as these signals originate from outside the digital design.. The output of the input synchronizers drive the i2c_start_stop module. The synchronized scl signal also passes through an edge detector to obtain the scl_negedge and scl_posedge signals. The scl_negedge signal drives the scl_trig_low signals of the i2c_read_byte and i2c_write_byte modules, as the low value of SCL enables writing on the SDA signal. The scl_posedge signal drives the scl_trig_high signal, as the SDA has received its high value, and the master device should have already written the desired value to the SDA signal.

The SDA out signal (input of the bidirectional splitter, which is driven to output when enabled) will take on the value of the i2c_write_byte SDA output if the SDA is allocated by the i2c_write_byte module, and will be constantly low (representing the SDA value when allocated by the i2c_read_byte module). The bidirectional output enable signal is high if either the i2c_read_byte or i2c_write_byte module allocates the SDA signal.

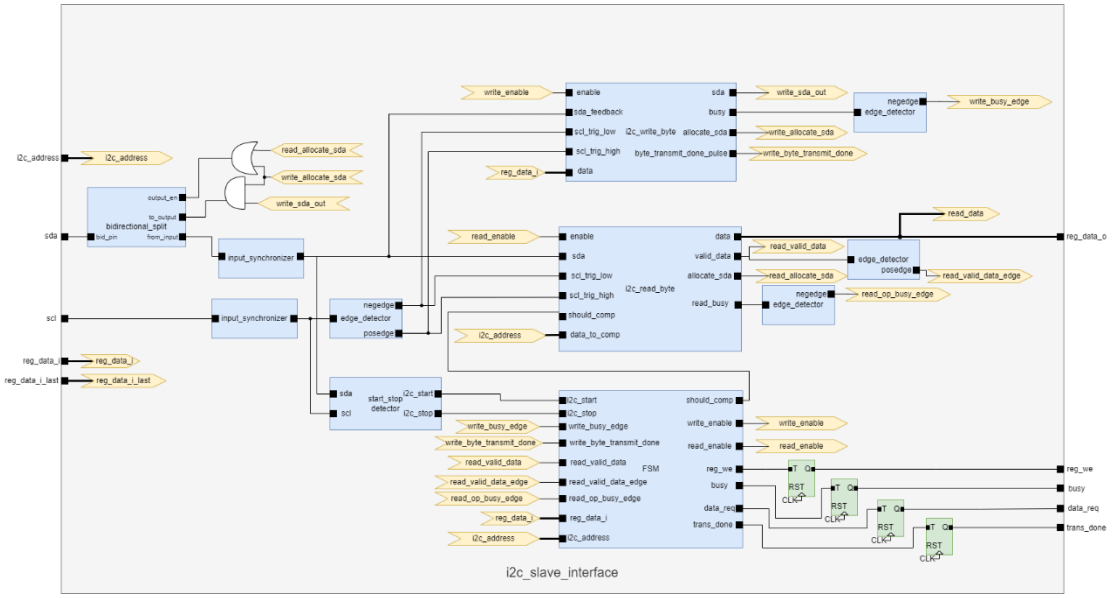

Figure 4.1.30: I^2C Slave Interface Schematic.

The FSM in the I^2C Slave interface is comprised of five primary states: IDLE, ADDRESS, REGISTER, READ, and WRITE. During the IDLE state, no operation is carried out. Once the start signal is triggered, the state transitions to the ADDRESS state after two cycles. In this state, the i2c_read_byte module is enabled with the

"should_comp" pin enabled. If the 7 most significant bits of the received data do not match the I2C address, the read operation is considered a failure, and the FSM returns to the IDLE state. If the received bits match the address, the least significant bit is checked to determine if the operation is a "write" or "read". If the bit value is 0, the operation is considered a "write" from the master device, and the FSM transitions to the REGISTER state. In this state, the desired register address is received from the master device. The FSM then transitions to the READ state, and data is read until the register file signals that the last data has been received. If the least significant bit of the byte received in the ADDRESS state is 1, the state transition from ADDRESS is to the WRITE state. The module continuously outputs the data stored in the register that was previously specified in a write operation, until the "last data" signal is activated.

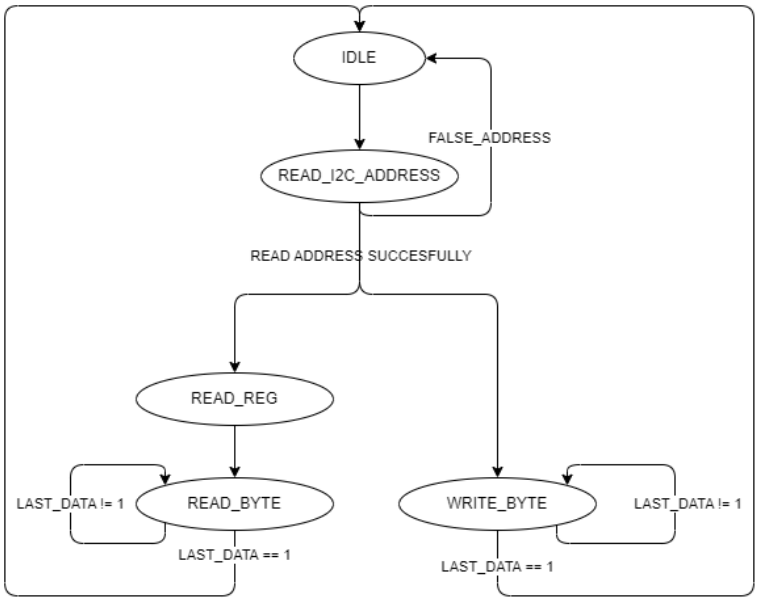

Figure 4.1.31: I^2C Slave FSM

4.2 4.2 RTL Verification

The RTL Verification process is divided into two stages: module tests and functional tests. Both stages utilize Verilog for test writing and are managed by separate Python scripts. To ensure a streamlined testing process, the tests are automated and run at least once for each Merge Request within the GitLab environment through the use of CI/CD flows.

4.2.1 Module Tests

The objective of the module tests is to evaluate the functionality of individual modules. Each test is comprised of a directory containing a Verilog testbench and a corresponding YAML file. The YAML file may include the following fields.:

- src_files: A list of the source files provided to the Verilog compiler to compile the test.
- enable: A Boolean variable indicating if this test should run or not.
- only: A Boolean variable, indicating whether or not this test (and any others with "only enabled") should be executed. It is not recommended to enable this in any tests in the stable branch, but it proves valuable during the development phase for debugging purposes..

fpga > test > module_tests > interfaces_i2c_master > ! simulation_src.yml

```yaml
src_files:
  - rtl/interfaces/i2c_master_interface/i2c_master_interface.v
  - rtl/interfaces/i2c_master_interface/i2c_start_stop.v
  - rtl/interfaces/i2c_master_interface/scl_controller.v
  - rtl/utils/i2c_write_byte.v
  - rtl/utils/i2c_read_byte.v
  - rtl/utils/edge_detector.v
  - rtl/utils/bidirectional_split.v
  - rtl/utils/serial_to_reg.v
  - rtl/utils/reg_to_serial.v
enable: true
only: false
```

Figure 4.2.1: Example of a module test yaml file

The testbench implements various scenarios and samples signals or data. At each sampling interval, the testcases variable is incremented. If the sampled values do not match the expected values, the fails variable is incremented. To simplify the testbench file, Verilog tasks for development and debugging are utilized to run the different scenarios. Upon completion, a message displaying the score (PASS/TOTAL) is printed, and if the fails variable is equal to zero, the simulation is terminated with the

$finish command. If the fails variable is not equal to zero, the simulation exits with the $fatal command, allowing the simulation caller to be aware of the result.

The testing process of the Verilog testbench is managed by a accompanying Python script. At the start, the script scans the directories within the "$I2C_BRIDGE_PRJ_ROOT/fpga/test/module_tests" folder in search of "enable" and "only" keys in the associated yaml files. Tests that have the "enable" key are placed in a "lowPriorityTests" list, while those with a "only" key set to "true" are placed in the "highPriorityTests" list. After the scanning process, the final tests list is determined based on the presence of elements in the "highPriorityTests" list; if it is not empty, the tests list is set to that, otherwise, it is set to the "lowPriorityTests" list.

The script then compiles each element in the tests list using the iverilog tool along with the specified testbench and source files. If an error occurs during the compilation process, the script exits with a "testbenchFailure" exception and returns a code of 1. Otherwise, the simulation process is initiated by calling the "vvp" tool with the compiled file as an argument. If the simulation finishes successfully, indicated by a return value of 0 and the Verilog testbench ending with the "$finish" command, the script proceeds to the next element in the tests list. Otherwise, the script exits in the same manner as the compilation error.e

4.2.2 Functional Tests

The functional tests are designed to verify the overall functionality of the system. Unlike the module tests, the Design Under Test (DUT) remains constant, but the number of different flows and behaviors to be tested increases. To simplify the process, these flows are separated into distinct Verilog files containing only tasks and variables, and a Python script is utilized to gather the desired flows, combine them with a base testbench, and generate a unified testbench for the simulation. This approach of generating a concise testbench is to facilitate its reuse for post-synthesis testing.

The base testbench file consists of the instantiation of the Design Under Test (DUT) and its associated wires, as well as an initial block containing the $finish command. Additionally, it includes three general-purpose tasks, namely the data test, the signal test, and a delay sync task which serves to synchronize with the clock. The file also

contains two comments, the first of which is positioned above the initial block, providing a location for the desired tasks to be copied. The second comment is located within the initial block, serving as the header of the last task of each test, acting as the main task of each test.

Each test is instantiated in a directory and contains the Verilog file and a yaml file. The yaml file has the enable and only keys, with similar functionality of those in the module tests. In case no test has its “only” key true, those with the “enable” true will have their Verilog code been integrated in the generated testbench, else only those with the “only” true. Each test should have a task at the bottom without arguments which will play the role of the main task. Its header will be instantiated in the initial block of the generated testbench. It should reset the total_testcases and fails variables and let them increase accordingly during the test. Then it will call any other tasks to run the test. Finally, it will print the score and in case the fails variable is not 0, it will run the $fatal command exiting of the simulation. Else it will let the next test to run.

The tests do not directly interact with the DUT's inputs and outputs, but instead create scenarios and assess their results. For example, when testing the read/write operation of registers, the focus should be on the return value of the register, rather than the I2C signals. The I^2C signals are handled by an intermediary layer, known as the drivers, which are always included in the generated testbench. The test tasks call the drivers to facilitate communication with the DUT and to participate in the testing process.

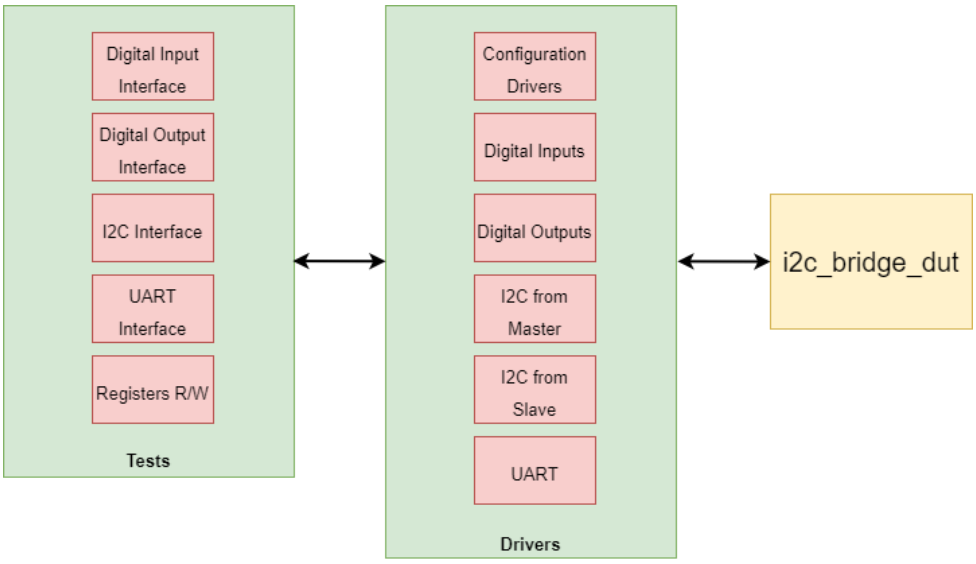

Figure 4.2.2: Functional Tests Block Diagram

4.2.3 RTL Verification Automation

In order to maintain consistent verification of the RTL, an automated process is necessary. During development, tests are conducted continuously, focusing on specific parts by enabling the "only" key or conducting general tests to ensure that changes to one flow do not negatively impact others. However, this alone is not sufficient, as there must be a record of the verification tests to ensure that each branch to be merged into the master (stable) branch has been properly verified.

For this purpose, GitLab CI/CD is utilized. This tool runs the tests in its environment and will not allow a branch to be merged into the master branch if the verification process fails. In Figure 4.2.3, Branch 1 passes the verification and is successfully merged into the master branch, while Branch 2 fails and GitLab blocks the merge process. Branch 2 must address the error and, upon re-testing, may be merged. This ensures that the RTL in the master branch is always verified.

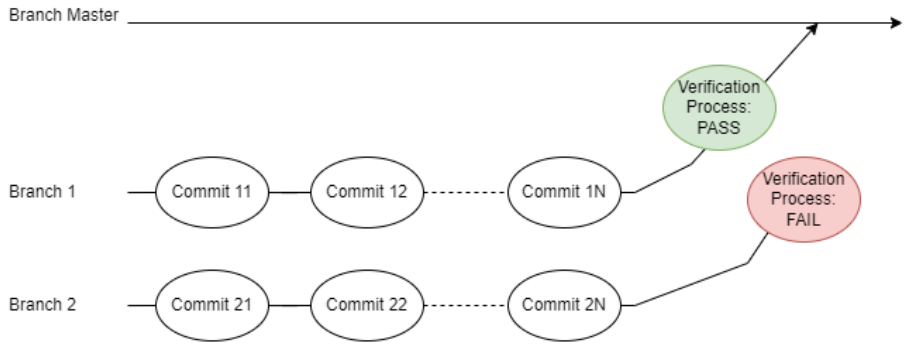

Figure 4.2.3: Code Quality Maintenance flow

GitLab CI/CD utilizes a docker image to run the verification process. To accommodate this, a docker image is created using a DockerFile located in the repository. The image is based on Debian and includes additional software, such as python3, python3-pip, iverilog, and gtkwave. The inclusion of gtkwave allows for the image to also be used for development and testing from another workstation if necessary.

4.3 4.3 Physical Implementation

4.3.1 Clock and Reset

Both FPGAs have two on-board oscillators: one with a low frequency of 10kHz and another with a high frequency of 48MHz. [10] To support the high speeds of the I2C bus communication with the slave device, the high frequency oscillator was selected. The high-fanout clock signal is passed through a global buffer, along with the reset signal. Unfortunately, the Power-On reset takes place before the design is loaded onto the FPGA, so it cannot be utilized in the design. As a result, a RESET_N signal must be assigned to a chip pin and the user must provide their own power-on reset for the FPGA design.

4.3.2 I/Os Assignment

The requested Inputs/Outputs pins are provided with the PCF. The pinout for the iCE40UP5K and for the iCE40UL1K is described in the Figures 4.3.1 and 4.3.2.

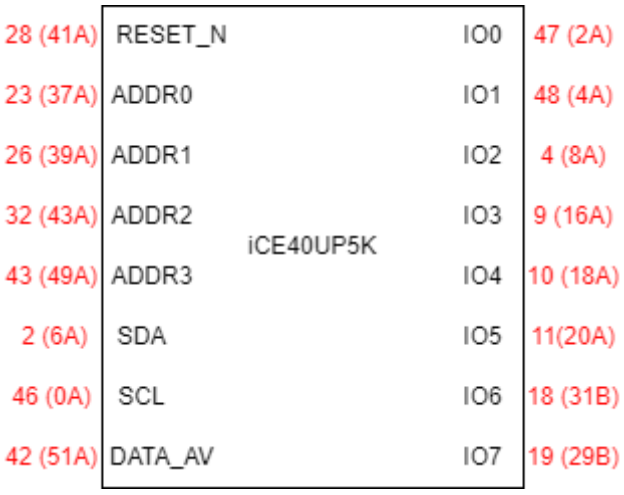

Figure 4.3.1: iCE40UP5K Pinout

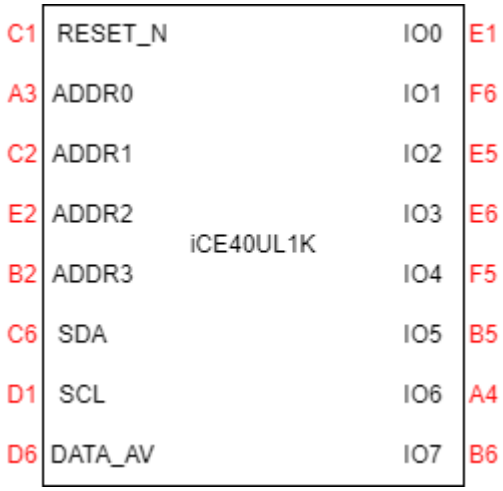

Figure 4.3.2: iCE40UL1K Pinout

4.3.3 FPGA Utilization and Floor Planner

The Utilization summary after the Placer and Route operation for the iCE40UP5K is presented in Table 4.3.1 table. Also, the design floorplan for this FPGA is demonstrated in Figure 4.3.3.

Table 4.3.1: iCE40UP5K Utilization

Elements	Used	Total
LogicCells	1065	5280
PLBs	187	660
BRAMs	5	30
IOs and GBIOs	16	36
PLLs	0	1
DSPs	0	8
LFOSCs	0	1
HFOSCs	1	1

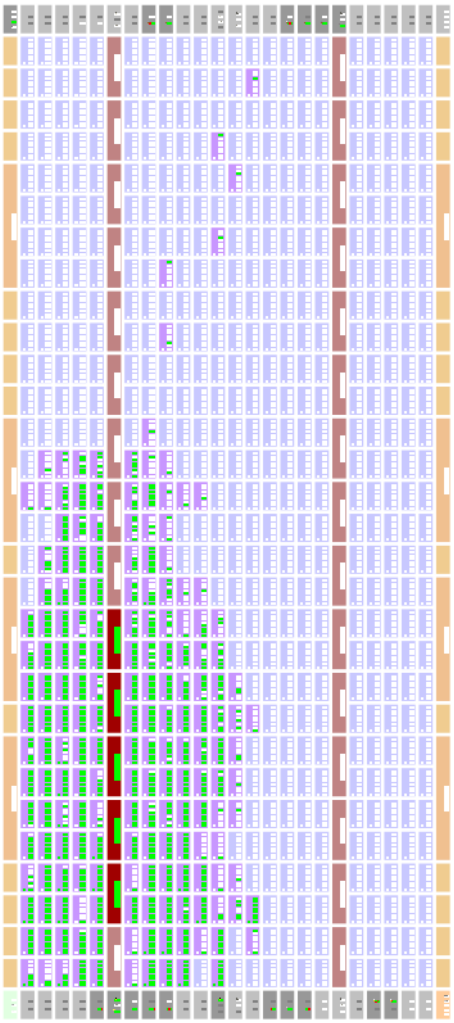

Figure 4.3.3: iCE40UP5K Floorplan

For the iCE40UL1K FPGA, the Utilization summary is presented in Table 4.3.2 table, while the relative floorplan in Figure 4.3.4.

Table 4.3.2: : iCE40UL1K Utilization

Elements	Used	Total
LogicCells	1065	1248
PLBs	153	156
BRAMs	5	14
IOs and GBIOs	16	21
PLLs	0	1
DSPs	0	8
LFOSCs	0	1
HFOSCs	1	1

Figure 4.3.4: iCE40UL1K Floorplan

4.3.4 Static Timing Analysis

The design on the ICE40UP5K has a slack of 989 ps. The critical path is located in the counter of the SCL controller (part of the I^2C Master interface). The path on the floorplan is provided in Figure 4.3.2.

```
Start
i2c_bridge_inst.to_slave_wrapper_m.i2c_master_interface_m.scl_controller_m.counter_0_LC_3_7_0/lcout
End
i2c_bridge_inst.to_slave_wrapper_m.i2c_master_interface_m.scl_controller_m.counter_0_LC_3_7_0/in0
Reference        OSCInst0/CLKHF
Setup Constraint 20833(p)
Path Slack       989(p)
```

Figure 4.3.5: iCE40UP5K STA Log Output

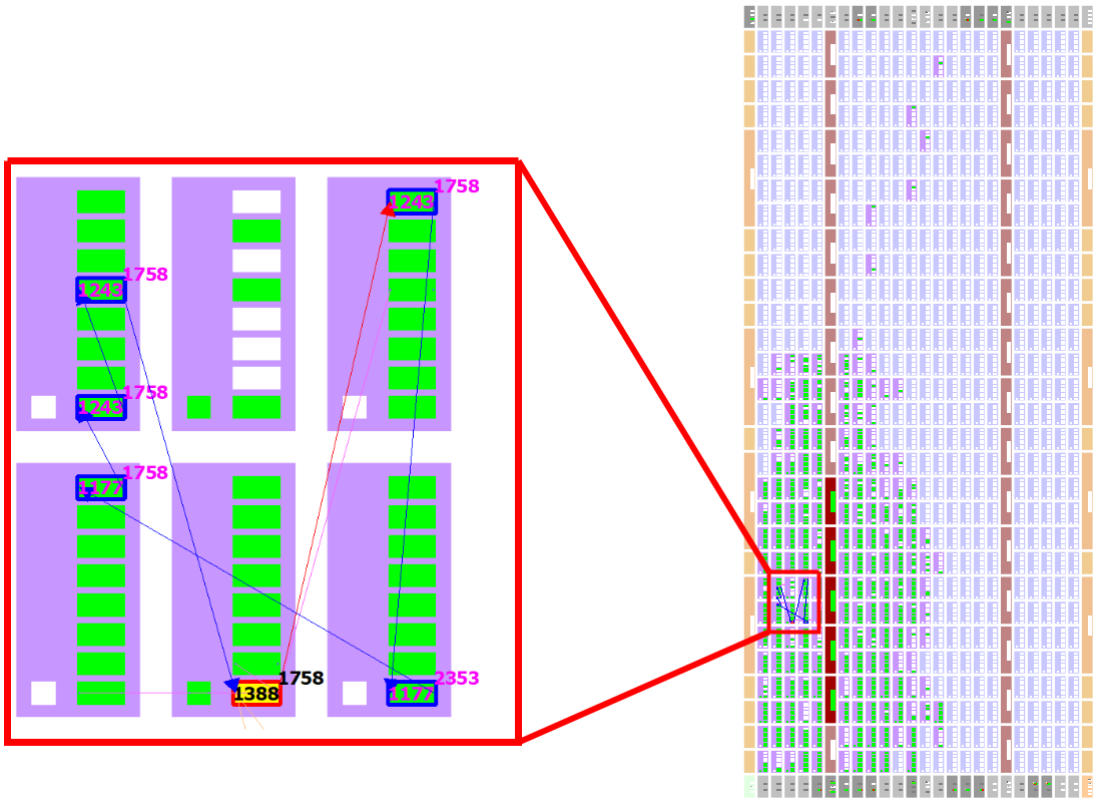

Figure 4.3.6: iCE40UP5K Critical path in Floorplan

The slack of the ICE40UL1K design has a value of 1637 ps. The critical path starts from the transmitter busy signal in the I^2C Slave interface and reaches until the data from slave manager in the reg file. The path on the floorplan is provided below.

```
Start
i2c_bridge_inst.i2c_slave_interface_m.edge_detector_transmitter_busy.prev_state_LC_5_12_7/lcout
End
i2c_bridge_inst.reg_file_m.receive_data_reg_file_m.data_from_slave_manager_m.data_last_LC_5_11_3/in3
Reference        OSCInst0/CLKHF
Setup Constraint 20833(p)
Path Slack       1637(p)
```

Figure 4.3.7: iCE40UL1K STA Log Output

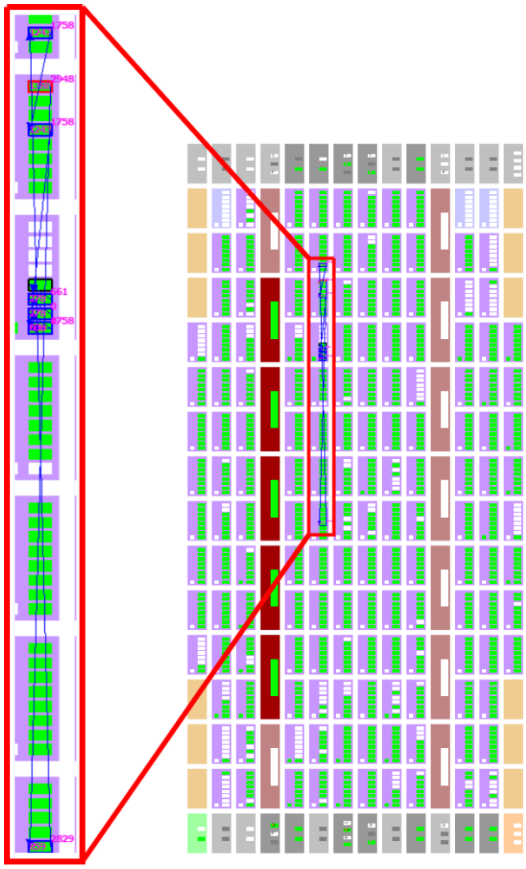

Figure 4.3.8: iCE40UL1K Critical path in Floorplan

4.3.5 Power Estimation

The Power Estimation requires an input voltage and the operation temperature. The input Voltage will be 3.3V as most designs operate in this rank, while a range of -20 to 80 Celsius feeds the Power estimation algorithm. The estimation is requested for both typical and worst case. The results are presented in the diagram of Figure 4.3.9

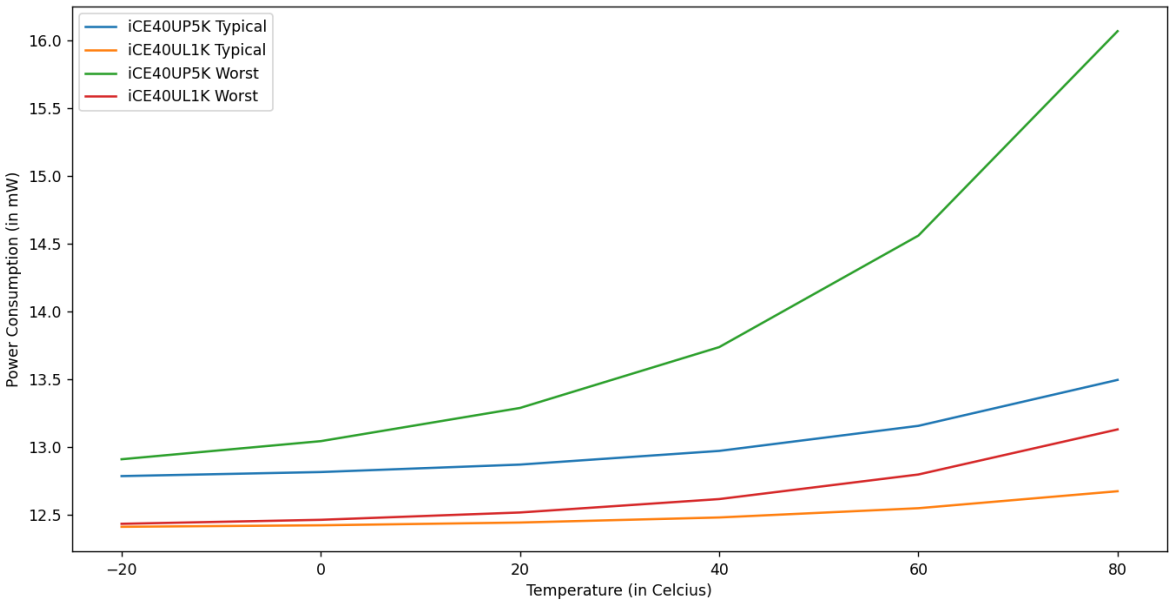

Figure 4.3.9: Power Estimation per Temperature value

Chapter 5 Software Driver

5.1 Driver Structure

The I2C Bridge module has been designed with ease of integration in mind. To simplify its implementation in end-user systems, a software driver is provided, which encapsulates the operational details of the module. The driver, written in C language and structured using CMake, is ready to be utilized by software developers.

The driver comprises three main components: the header file with the Application Programming Interface (API), the core functionality file, and the Hardware Application Level (HAL) layer. The API is defined in the i2c_bridge.h header file, the core functionality is implemented in the i2c_bridge.c source file, with the assistance of utility functions in the utils directory. The HAL layer is defined in the hal directory and serves to instantiate the I2C commands based on the platform.

If no platform is specified, the top-level source file, i2c_bridge_hal.c, will be compiled and its functions will return with a "not implemented" error. The supported platforms are defined in the platform directory, with currently only the Zephyr platform supported. If the Zephyr platform is defined, the HAL commands will result in Zephyr I2C read and write commands being called by the core functionality. To add a new platform, a directory representing it should be added under the platform directory.

.

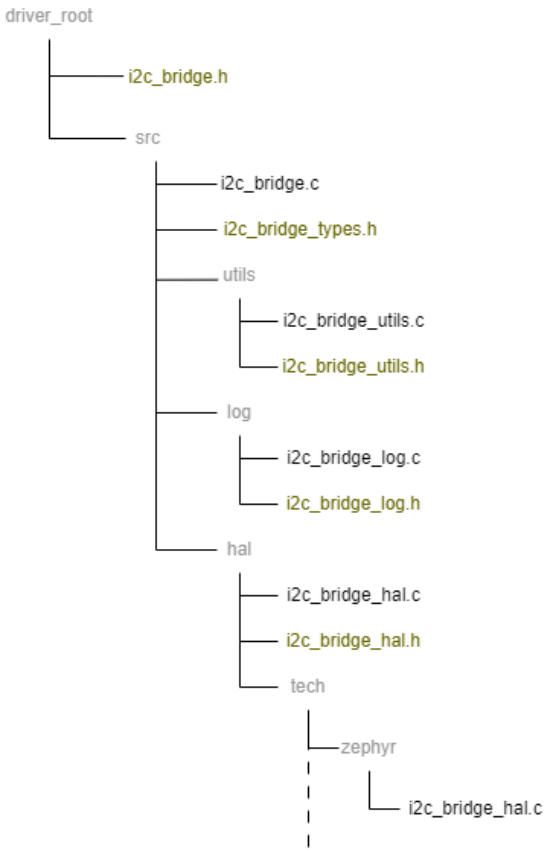

Figure 5.1.1: Driver Filesystem structure

5.2 Driver API

The API of the driver features a user-friendly interface consisting of straightforward read/write procedures and interface configuration functions. Specifically, for the UART interface, an additional function to specify the expected size of the read data is included, as the peripheral device is expected to have already transmitted the data by the time the MCU requests it.

5.2.1 i2c_bridge_set_interface_do

Sets the interface configuration register with the value of Digital Output interface. In case of successful operation, it will update the internal variable “interface”.

5.2.2 i2c_bridge_write_do

The function begins by evaluating the local variable to determine if the interface is set to Digital Output. If it does not match, the function calls the i2c_bridge_set_interface_do internally to set it up. Next, the function checks if the

transmit data size is 1, using the local variable. If it is not, the function updates it. Finally, the function sends a 1-byte bitstream to the transmit data register.

5.2.3 i2c_bridge_read_do

Reads the provided output bitstream. In case the interface or data size are not correctly set, it will return an error value. After this evaluation, it will read 1 byte from the transmit data register.

5.2.4 i2c_set_interface_di

Like the i2c_set_interface_do but setting the register to Digital Input.

5.2.5 i2c_bridge_read_di

In the event that the interface is not configured as Digital Input or the receive data size is not set to 1, the necessary operations will be initialized. Subsequently, a 1-byte input bitstream will be read from the receive data register.

5.2.6 i2c_set_interface_uart

Receives the desired baud rate as an argument. The function sets the configuration interface to UART with the specified baud rate. If the operation is successful, the relevant variables will be updated accordingly.

5.2.7 i2c_bridge_write_uart

The function receives the desired bytes to be sent and the number of them as input arguments. It evaluates whether the currently set interface corresponds to UART. If so, it encodes the number of bytes into a value for the expected transmit data size. Then, it checks if the transmit data size stored in the corresponding variable matches the encoded value. If there is a discrepancy, it updates the variable. Finally, it transfers the data to the module.

5.2.8 i2c_bridge_expect_uart_read_size

This function sets the receive Data Size as usual.

5.2.9 i2c_bridge_read_uart

Evaluates if the configured interface is UART and reads from the receive data register as the receive data size is set.

5.2.10 i2c_bridge_set_interface_i2c

The function takes the desired speed as an input argument. It sets the configuration interface to I^2C with the specified speed code. Upon successful completion, the corresponding variables will be updated to reflect the change.

5.2.11 i2c_bridge_write_i2c

The function takes the peripheral device address, the address of the peripheral device's register and its length, the data to be written to the desired register, and the length as input parameters. To begin with, after checking the set I^2C in the interface configuration register, it will set the encoded values of the transmit and peripheral register address size if they have not already been set. Then, the register address will be transmitted. Finally, the address of the peripheral device will be set if it has not already been set, followed by the data to be transmitted.

5.2.12 i2c_bridge_read_i2c

The function takes in the peripheral device address, the address of the peripheral device register and its length, the length of the expected read data, and a buffer to store the read data as input arguments. Initially, it evaluates the I^2C interface configuration setting and sets the receive and peripheral register address size, if they have not already been set to the encoded values. Then, it sends the register address to the peripheral device. The address of the peripheral device is set, and the read operation is enabled. This is done to initiate the read operation. After a 100 ms delay, the driver retrieves the read data from the receive data register.

5.2.13 i2c_bridge_init

The internal variables are initialized with default values that do not reflect their operational state, and the HAL is also initiated.

5.3 Sample on the Zephyr platform

5.3.1 Structure

Four distinct sample applications have been developed to evaluate the functionality of the I2C Bridge Module, with each application being specifically designed to test a different supported interface. These applications are based on the Zephyr RTOS and the specific hardware platform is specified during the build process. Each application comprises a Kconfig file, a prj.conf file, and a CMakeLists.txt file, which contain the necessary source code and configurations. The Kconfig file initializes a LOOP_DELAY configuration to 10 seconds. The prj.conf file sets the global configurations for I2C and Logging. The CMakeLists.txt file contains the necessary CMake configurations for Zephyr to properly compile the source code. The main source code is located in a main.c file within the src directory. To maintain the cleanliness of the Zephyr repository, the I2C bridge driver is utilized as an external source code and is not included within the internal drivers.

5.3.2 Digital Output Sample

To evaluate the Digital Output a simple sample is implemented that provides the value of an 8-bit counter which increments per 1 value each 10 seconds. The binary representation of the 8 I/O should be updated respectively.

5.3.3 Digital Input Sample

An application has been developed to monitor the digital input bitstream. This application performs a scan of the digital input bitstream every 10 seconds and logs the resulting value obtained after each successful scan.

5.3.4 UART Sample

An application has been developed to use the UART interface at the software level for the purpose of receiving environmental variables from a UART interface sensor. The sensor that will be utilized is the CozIR-A manufactured by Gas Sensing Solutions Ltd. (GSS). To collect measurements, specific strings must be sent to the sensor. To retrieve the CO2 measurement, the string "Z\r\n" should be sent, to retrieve the

temperature measurement the string "T\r\n" should be sent, and to retrieve the humidity measurement the string "H\r\n" should be sent. After each sent command, a response will be received in the format of "Z ####\r\n" for CO2, "T ####\r\n" for temperature, and "H ####\r\n" for humidity [11]. The I^2C Bridge is capable of obtaining the first 8 characters of the response, and therefore, should receive the entire payload. The CO2 measurement value represents the parts per million as is, while to get the temperature number in Celsius, the provided value should be subtracted by 1000 and divided by 10. The humidity value should only be divided by 10.

5.3.5 I2C Sample

For the I^2C to peripheral devices interface evaluation of the I2C Bridge module, also an environmental sensor is used, that can measure CO2, Temperature and relative humidity. This time, SCD4x sensor is used developed by Sensirion. First of all, we need to set the sensor in continues measurement mode, by pointing the 0x21B1 register address of the sensor. Even though pointing to register of Peripheral device is not supported, we can do this by sending the first byte as register and the second as data, with register address and transmit data size set to 1. This is allowed as the register address size of the SCD4x sensor is 2 bytes. Then periodically the measurements are collected in a 10 seconds interval. Before requesting them, a get data ready status command is sent to validate the existence of data. After with a single read command, CO2, Temperature and RH are provided. The CO2 ppm comes as is, the Temperature is converted to Celsius by multiplying the received value with 175 and dividing with 2^{16} which result must be subtracted by 45. To get RH in percentage, the input value should be divided by 2^{16} and multiplied by 100.

Chapter 6 Physical Experiments

6.1 Setup

After developing and validating the RTL of the I^2C Bridge module and generating bitstreams, as well as the software running on a MCU, the next step is to test the

whole system with a physical hardware setup. The main parts of setup includes the below parts:

- A host PC
- An STM Nucleo LR152RE as the MCU/Main Board
- A Lattice iCE40 UltraPlus Breakout board whose FPGA operates as the I^2C Bridge Module.
- LEDs to evaluate Digital Output Operation
- Jumper Wires to toggle the Digital Inputs
- Sensirion SCD4x sensor, as a Peripheral with I2C interface
- GSS CozIR sensors, as a Peripheral with UART interface
- DIGILENT Digital Discovery to visualize the signals 1 or 0 state.

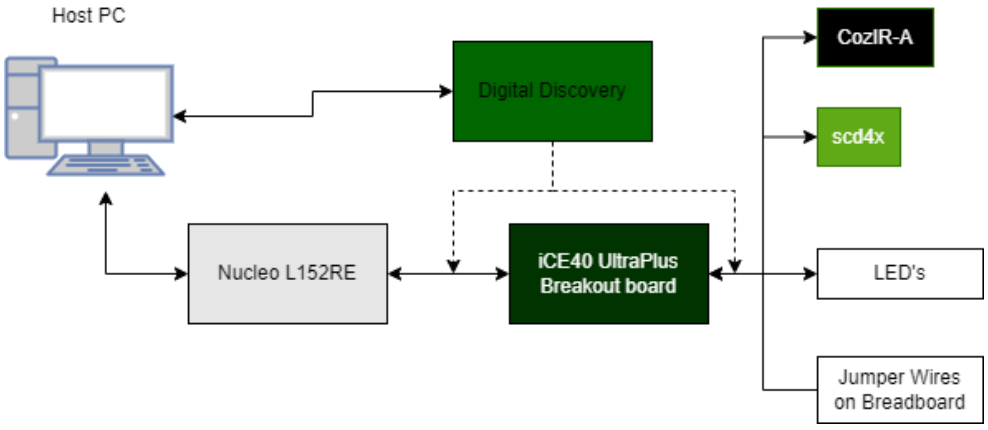

Figure 6.1.1: Schematic of setup for Physical Tests

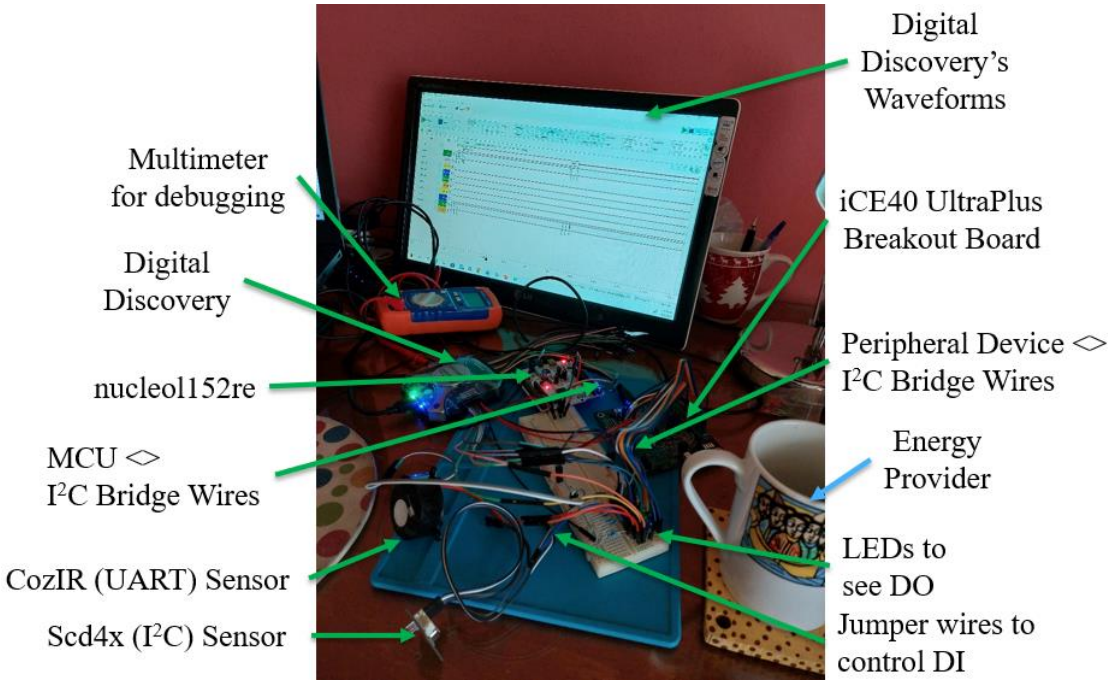

Figure 6.1.2: Setup photo

6.2 Digital Output Interface Test

To evaluate the Digital Output functionality, the Digital Output sample was flashed on the Nucleo LR152RE board. In Figure 6.2.1, we can see from the right side firstly that the I2C bus (2nd and 3rd waveform) was activated and provided the bitstream value 0x1 to the output (next waveforms). After a while it is triggered again and provides bitstream 0x2. The MCU provides its logs to the host PC which are printed in the middle screen. Left we can see the LEDs status after bitstream 0x2 was written.

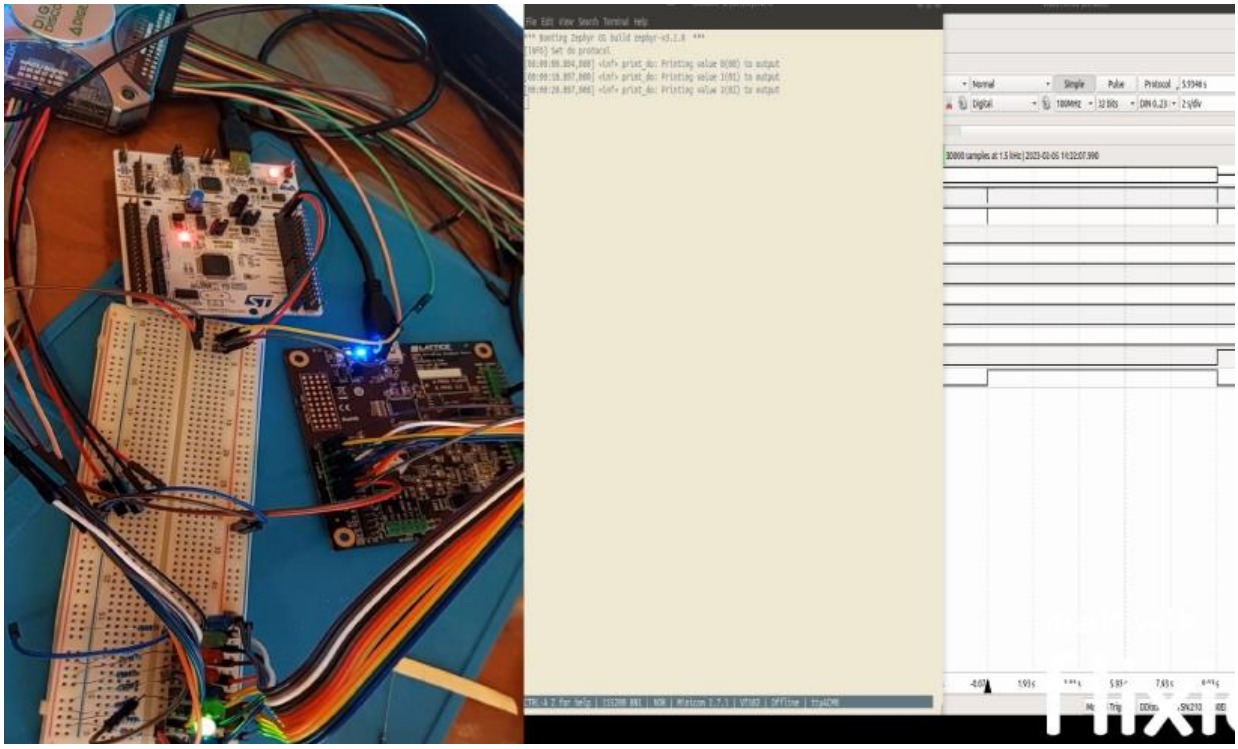

Figure 6.2.1: Digital Output Test Video Print screen

6.3 Digital Input Interface Test

In order to evaluate the functionality of the Digital Input, the Digital Input sample was programmed onto the MCU. As shown in Figure 6.3.1, it can be observed that upon requesting data from the I2C bus (as evidenced by the brief fluctuation of the 2nd and 3rd waveforms), the MCU successfully scans the updated value 0x7 (binary representation 2'b111), as demonstrated by the subsequent 8 waveforms.

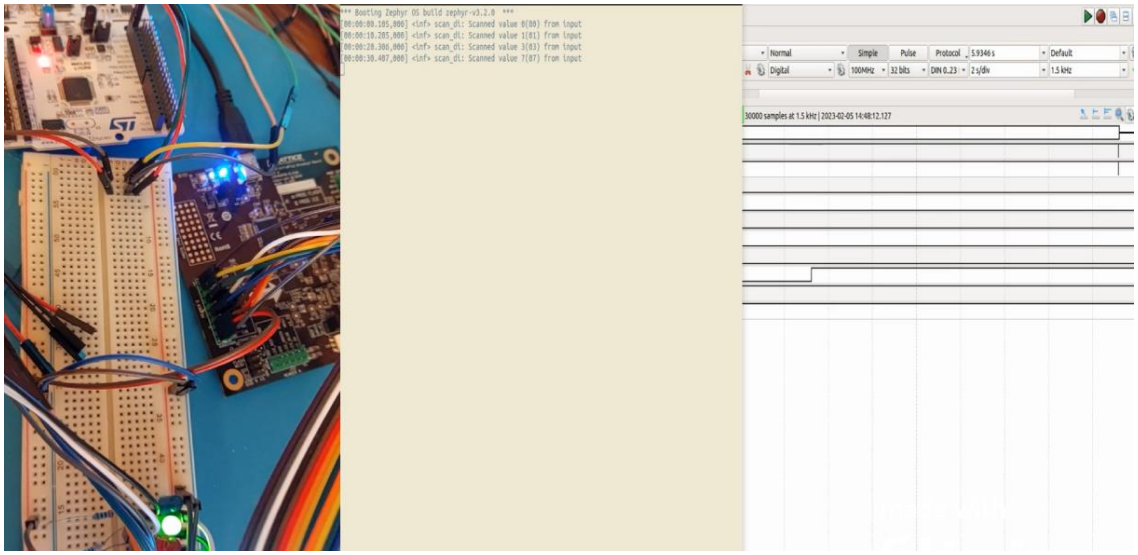

Figure 6.3.1: Digital Input Test Video Print screen

6.4 UART Interface Test

The functionality of the UART interface was evaluated by flashing the CozIR-A sensor sample on the MCU. Figure 6.4.1 demonstrates the UART waveforms toggling three times per measurement. This is because the CO2 measurement is requested first, followed by the temperature, and finally the relative humidity. Although not all toggles are captured in the I^2C lines due to limitations in the measuring equipment, it can be observed that with each measurement, a command is sent to the sensor and the received data is then requested. Following the first measurement, it can be seen that the readings increased after blowing air into the sensor.

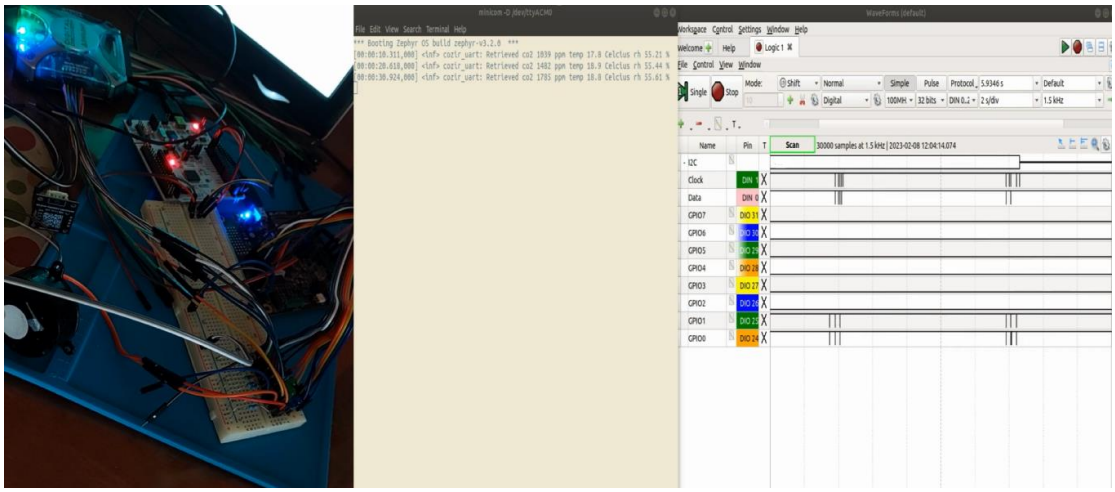

Figure 6.4.1: UART Test Video Print screen

6.5 I²C Interface Test

The performance of the I²C interface was assessed by flashing the SCD4x sensor sample onto the MCU. As illustrated in Figure 6.5.1, the I²C waveforms were observed to toggle twice during communication with the sensor, with one toggle on the SDA line and one toggle on the SCL line, due to limitations with the measurement equipment. The first toggle was to obtain the data ready status, and the second toggle was to retrieve all the environmental variables provided by the sensor. After the second measurement, a breath was directed onto the sensor, leading to an increase in the recorded values, as observed in the figure.

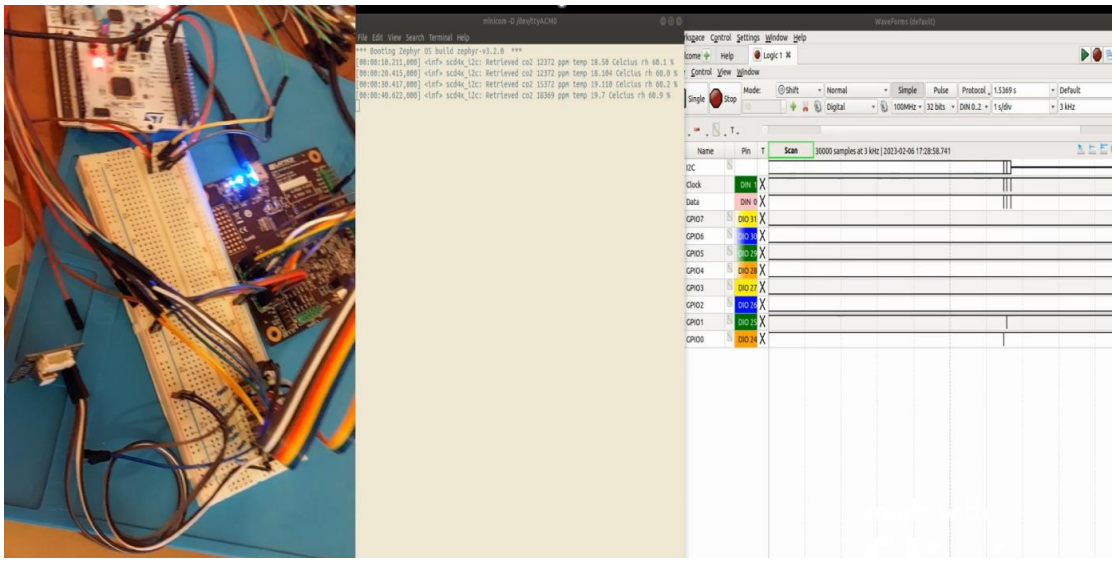

Figure 6.5.1: I²C Test Video Print screen

Chapter 7 Conclusions

7.1 Summary

The objective of this thesis was to develop a solution for streamlining the logistics management of PCBs attached to a Main Board. The aim was to reduce design and certification costs by allowing the use of a single Main Board for multiple applications without any modifications.

The project approach emphasized the importance of creating and running an automated verification process to ensure reliable performance during each step of the implementation. The digital hardware design was kept as simple as possible to facilitate integration in other hardware platforms and maintenance, while ensuring 100% functionality. This design is intended to be integrated into IoT architecture at after the MCU level, where Over-The-Air (OTA) updates can be challenging to take place if a bug is discovered. The author's hope is for this design to eventually be integrated into an ASIC, where OTA updates are not an option.

7.2 Future Work

The project undertaken in this thesis has the potential to serve as a foundation for multiple future projects. These projects can be divided into two main categories: updates and optimizations to the digital hardware design, and integration of the I^2C Bridge Module into existing or new software applications.

In regards to digital hardware development, a possible avenue for further research is the utilization of a low frequency oscillator to reduce energy consumption and increase the application of the system in low power environments. However, this may also have limitations in terms of compatibility with high-speed buses and potential issues with internal delay. Another potential project is the integration of additional communication protocols such as SPI or even analog signals through the use of a Digital to Analog Converter (DAC). Furthermore, the ability for the I^2C Bridge to support multiple protocols simultaneously would also be an area for exploration. For example, it would be desirable for the I^2C Bridge to have the capability to simultaneously utilize UART on I/O 0 and 1, Digital Inputs on I/O 2 and 3, I^2C on I/O 4 and 5, and Digital Outputs on I/O 6 and 7.

At the software level, the integration of the I^2C Bridge into new Internet of Things (IoT) projects is a possibility. As the MCU can communicate with different types of peripherals, just from a simple 2-wire interface, the sky is the limit.

Chapter 8 Bibliography

[1] M. G. L. E. Pena, "UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter," *AnalogDialogue,* vol. 54, no. 4, December 2020.

[2] NXP Semiconductors, *I2C-bus specification and user manual,* 7.0 ed., 1 October 2021.

[3] "Wikipedia," 3 January 2023. [Online]. Available: https://en.wikipedia.org/wiki/Field-programmable_gate_array. [Accessed 23 January 2023].

[4] S. H. H. A. M. Pedram, *Statistical timing analysis of flip-flops considering codependent setup and hold times,* Orlando, Florida, 2008.

[5] LATTICE Semiconductor, *iCE40 UltraPlus Family Data Sheet,* 1.4 ed., 2017.

[6] Wikipedia, "Wikipedia," 20 January 2023. [Online]. Available: https://en.wikipedia.org/wiki/Device_driver. [Accessed 24 January 2023].

[7] S. Williams, "Icarus Verilog," [Online]. Available: http://iverilog.icarus.com/. [Accessed 24 January 2023].

[8] Docker Inc., "Docker Get started," [Online]. Available: https://docs.docker.com/get-started/. [Accessed 24 January 2023].

[9] R. Ginosar, "Metastability and Synchronizers: A Tutorial," *IEEE Design & Test of Computers,* vol. 28, no. 5, pp. 23-35, September/October 2011.

[10] LATTICE Semiconductor, *iCE40 Oscillator Usage Guide,* 1.4 ed., 2017.

[11] Gas Sensing Solutions Ltd., *CozIR®-A Production Data,* 4.2 ed., 2020.

[12] BOSCH, "BME280 – Data sheet," September 2018. [Online]. Available: https://www.mouser.com/datasheet/2/783/BST-BME280-DS002-1509607.pdf. [Accessed 18 01 2023].