Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CENTER FOR
PROJECT-BASED
LEARNING

Bachelor Thesis

Spring Semester 2023

Department of Information Technology and Electrical Engineering

Development of Communication Library between FPGA and MCU for a Hyperloop Prototype at Swissloop

Tim Wyss

tiwyss@student.ethz.ch

Supervisors:	Dr. Christian Vogt, christian.vogt@pbl.ee.ethz.ch
	Dr. Michele Magno, michele.magno@pbl.ee.ethz.ch
Professor:	Prof. Dr. Luca Benini, lbenini@iis.ee.ethz.ch

Abstract

This thesis aims to develop a fast, reliable, and scalable interface library to be used in the motor and levitation control systems by Swissloop. Swissloop is a student initiative associated with ETH Zurich aiming to develop a Hyperloop platform. The Hyperloop concept, popularized in 2013 by Elon Musk, describes a fast and efficient mode of transportation using levitating vehicles in a near vacuum tube.

The interface library developed for this thesis provides a high-speed serial link between MCU- and FPGA-based components of Swissloop's control systems. The interface implementation is accompanied by a software tool that generates code based on user-specified parameters.

The library provides access from the MCU to configurable memory on the FPGA over SPI, which is supported in various modes and configurations with a clock frequency of up to 24.6 MHz. The memory is accessible to the FPGA through a simple I/O-interface with a latency of 14.013 ns.

After implementation, the interface was tested, analyzed, and benchmarked using various methods. This process allowed for the identification and analysis of performance-related specifications and other characteristics of the interface, such as power consumption or FPGA usage. It was shown that running the SCLK at 49.2 MHz does not significantly improve the latency of the interface compared to 24.6 MHz for larger transactions and that it cannot be used reliably for this application. Nevertheless, SPI was used successfully at 49.2 MHz during benchmark tests, achieving a latency of just 3.28 µs.

This work provides a platform for future projects to add support for more interfaces and configurations. To make integration in other projects as straightforward as possible, special emphasis was put on documenting and formatting the code. Additionally, a user guide was written to make it easy to use.

Ackowledgements

I would like to thank the staff of the PBL and in particular my co-supervisor Dr. Christian Vogt for his valuable assistance and expertise. I am also grateful to Dr. Michele Magno and Prof. Dr. Luca Benini for making this thesis possible.

Additionally, I would like to thank Swissloop and everybody involved for the opportunity to work on this project and for their assistance with ideas and resources throughout the thesis. Special thanks are due to the focus coaches at Swissloop, Philip Wiese and Luca Rufer, for the helpful conversations we had along the way.

Declaration of Originality

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisors. For a detailed version of the declaration of originality, please refer to Appendix A.

Tim Wyss,
Zurich, June 2023

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

ASIC	Application-Specific Integrated Circuit
CAN	Controller Area Network
CDC	Clock Domain Crossing
CLB	Configurable Logic Block
CLK	Clock
CNN	Convolutional Neural Network
CPHA	Clock Phase
CPOL	Clock Polarity
CS	Chip Select
DNN	Deep Neural Network
EBR	Embedded Block Random Access Memory (RAM)
EHW	European Hyperloop Week
EMI	Electromagnetic Interference
FIFO	First In – First Out
FPGA	Field-Programmable Gate Array
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
HDL	Hardware Description Language
HLS	High Level Synthesis
I/O	Input/output
I^2C	Inter-Integrated Circuit
IC	Integrated Circuit
ICU	Inverter Control Unit
IDE	Integrated Development Environment
IIS	Integrated Systems Laboratory
LCD	Liquid Crystal Display
LCU	Levitation Control Unit
LSB	Least Significant Bit
LUT	Look-Up Table

MCU	Microcontroller Unit
MISO	Master In Slave Out
MOSI	Master Out Slave In
MSB	Most Significant Bit
OS	Operating System
OSI	Open System Interconnection
PAR	Place And Route
PBL	Center For Project-Based Learning
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PDF	Portable Document Format
PFU	Programmable Functional Unit
PISO	Parallel-in, Serial Out
RAM	Random Access Memory
RTL	Register Transfer Level
SCLK	Serial Clock
SIPO	Serial-in, Parallel-out
SPI	Serial Peripheral Interface
TCL	Tool Command Languages
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus

1. Introduction

1.1. Swissloop

Swissloop is a student-led organization associated with the ETH Zurich aiming to contribute to the development of the Hyperloop technology and its application in the real world. The Hyperloop concept usually refers to a white paper called *Hyperloop Alpha* [1] published in 2013 by Elon Musk. It describes a futuristic mode of high-speed transportation, with pressurized vehicles, called *pods*, traveling at up to 1220 km/h in a near vacuum tube. To this end, the Hyperloop concept envisions the pods using contactless propulsion while levitating to minimize friction. Compared to current transportation systems such as roads, rails, and air, the Hyperloop concept is significantly faster, more efficient, and less expensive to operate.

Swissloop designs and builds operational prototype Hyperloop pods, with which they compete in the annual European Hyperloop Week (EHW) [2], an international competition where teams from across the world demonstrate their designs. Swissloop plans to participate in the EHW in July 2023, where a new motor, improved levitation systems, and more will be demonstrated.

1.2. Objective

For the controls of the levitation system and motor, several high-speed controllers were implemented on Field-Programmable Gate Arrays (FPGAs) and microcontroller units (MCUs). The objective of this thesis is to create an interface library for communication between the MCU and FPGA, which can be used for several years to come. The interface should reach the following performance metrics:

- The maximum latency should be lower than 100 µs.
- Throughput should be at least 500 kbits/s.

Additionally, the thesis aims to demonstrate the viability and benefit of code generation for communication interfaces, e.g., between microcontrollers and FPGAs, to ensure compatibility between both devices.

The full outline can be found in Appendix B.

1.3. Outline

After the introduction to this thesis in this chapter, the necessary theoretical background is explained in Chapter 2. A presentation of similar or related works follows in Chapter 3, before design and implementation are described in Chapter 4. Chapter 5 explains and analyses the various measurements conducted, before discussing their results. Finally, in Chapter 6 conclusions are drawn and possibilities for future work are presented.

2. Background

This chapter provides a detailed explanation of the fundamental concept to better comprehend this work. First, an introduction to serial communication interfaces is provided, before focussing the discussion on Serial Peripheral Interface (SPI). Afterward, various aspects of FPGAs and its workflow are explained. The hardware used for this thesis is presented at the end of this chapter.

2.1. Serial Communication

Serial communication is a method of transmitting data one bit at a time, either over a single channel or a bus. This is in contrast to parallel communication interfaces, such as the *Peripheral Component Interconnect (PCI)*, where data is transmitted over multiple parallel channels. Communication interfaces often use a master/slave model, in which one device known as *master* or *master device* controls the operation of one or more other devices referred to as *slaves* or *slave devices* [3, 4]. A transaction is always initiated by the master, as the slave can only respond to a request by the master.

Serial data transmission can be either synchronous or asynchronous. Synchronous interfaces use a clock (CLK) signal to coordinate transmission, while asynchronous interfaces use other methods for synchronization [4]. The clock signal used in synchronous interfaces is also referred to as serial clock (SCLK) to avoid confusion with system clock signals on other integrated circuits. The clock signal in synchronous interfaces is driven by the master device. Transmission is achieved by setting the state of the data lines such that it can be sampled by the receiving device at the active edge of the SCLK.

One of the most widely used serial interfaces is the relatively new *Universal Serial Bus (USB)* from 1996, which is commonly used to exchange data with peripheral devices in consumer computer systems [5]. However, in some applications in embedded systems, other serial interfaces may be more advantageous. These include the *Universal Asynchronous Receiver-Transmitter (UART)*, *Inter-Integrated Circuit (I^2C)* (1982), *Controller Area Network (CAN)* (1989), and *SPI* (1979) [4, 6, 7, 8, 9].

For this thesis, SPI was chosen for implementation because of the design of the hardware provided by Swissloop (Section 2.6). SPI additionally allows for high data rates, making it a great choice for a low-latency interface. As such, theoretical research was focused on this particular interface.

2.2. SPI

SPI was introduced by Motorola in the 1980s and has since become a widely used synchronous interface specification in embedded systems [3, 4]. Unlike other synchronous interfaces that use only two lines (clock and data) to connect the master to all slaves, SPI employs additional lines for faster communication and more efficient slave device selection. Three lines lead from

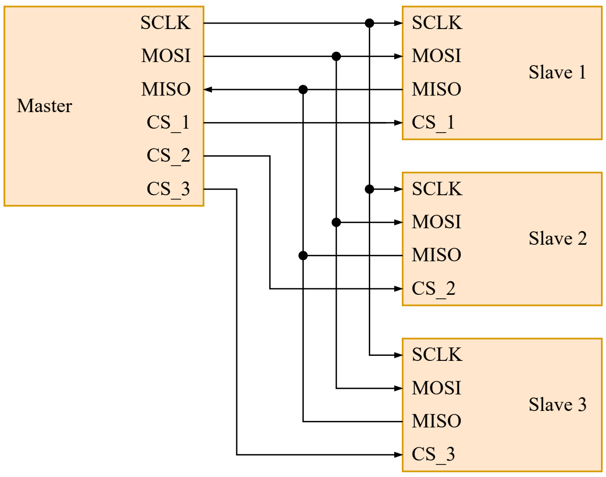


Figure 2.1.: Schematic view of a SPI interface with three slave-devices.

the master to all slaves: One line for the SCLK signal, and two lines for bidirectional ("full-duplex") communication - *Master Out Slave In (MOSI)* and *Master In Slave Out (MISO)* [3, 4, 9]. Additionally, there is a *chip select (CS)* line connecting the master to every slave device individually, which is implemented in a active low configuration [6]. This signal enables the master to select the slave device for communication. A layout with multiple slaves is shown in Figure 2.1. Interfaces without a CS signal must begin transactions with a special sequence of bits to indicate the start, resulting in additional overhead.

After asserting the CS, the master can begin transmitting data packets. Typically, a packet contains 8 bits (1 byte), but this can vary depending on the protocol (Section 2.3) [6].

Although it was shown that SPI can operate at clock frequencies above 900 MHz, typical implementations run their SCLKs in the range of 500 kHz to 30 MHz [10]. Using frequencies in this range limits standard SPI configurations to distances shorter than 10 meters to maintain synchronicity between the master and the slave [11].

SPI peripherals are either implemented using two-state or tri-state logic. However, if multiple slaves are connected to the same MISO line, they must be implemented in tri-state logic and their MISO outputs must be set to 'Z' (high impedance) when inactive [12]. The logic levels are typically 3.3V or 5V.

To improve latency and throughput, the MOSI and MISO lines can be expanded to a bus to enable parallel transfer of bits. The resulting configurations are referred to as quad- or octo-SPI, depending on the number of parallel lines in the bus.

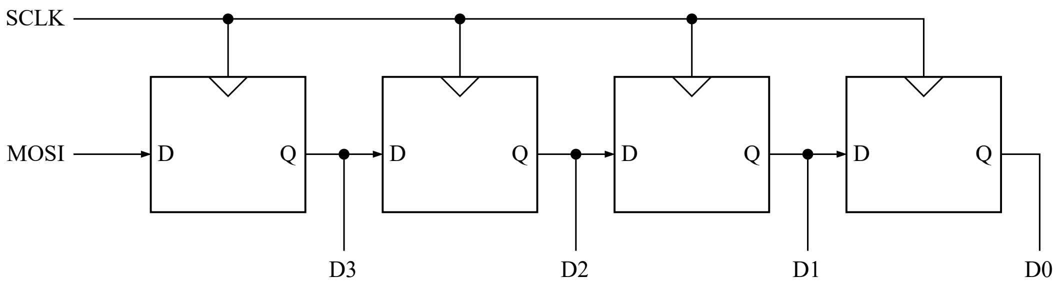


Figure 2.2.: 4 bit serial-in, parallel-out (SIPO) shift register.

Mode	CPOL	CPHA	Active Edge
0	0	0	Rising
1	0	1	Falling
2	1	0	Falling
3	1	1	Rising

Table 2.1.: The four Motorola SPI modes.

Two key components to a SPI peripheral are the SIPO (illustrated in Figure 2.2) and parallel-in, serial out (PISO) shift registers. The data line is connected to one of the start (SIPO) or to the end (PISO) of the shift register. At the active clock edge, the SIPO shift register appends the state of the data line to the currently held value, while the PISO shift register updates the state on the data line by changing the value in the final flip-flop.

SPI introduces minimal overhead compared to other interfaces by reducing the time required to set up a transaction and supports full-duplex data exchange. However, it can be relatively expensive from a hardware perspective, due to the additionally required lines between the master and slaves.

2.2.1. SPI Configurations

Because there is no official specification of how SPI should be implemented, several common configurations are used. These include four Motorola modes and a less common TI mode. The Motorola modes depend on the clock polarity (CPOL) and clock phase (CPHA) parameters, while the TI mode employs a distinct interpretation of the CS signal [3, 6].

CPOL describes the clock's state when the interface is idle; if CPOL is set, the clock is high when idle. CPHA specifies the clock transition at which data is sampled; if CPHA is unset, the data is sampled on the leading edge, otherwise, it is sampled on the trailing edge [4, 6]. The four possible Motorola modes are listed in Table 2.1. Figure 2.3 visualizes the active edge of the different modes.

In Motorola mode, the CS signal is expected to remain active (i.e., low) for the entire duration of the transaction. In contrast, in TI mode, the CS is expected to deactivate during the transaction of the final bit of every packet. [13].

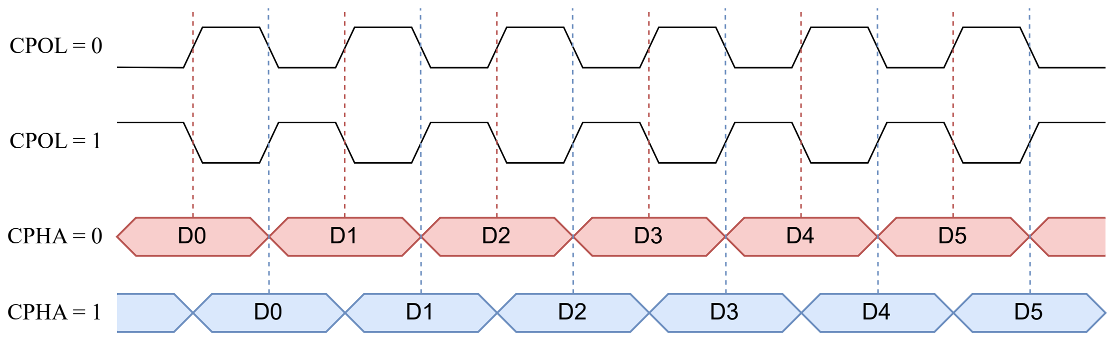


Figure 2.3.: Timing diagram of the Motorola modes.

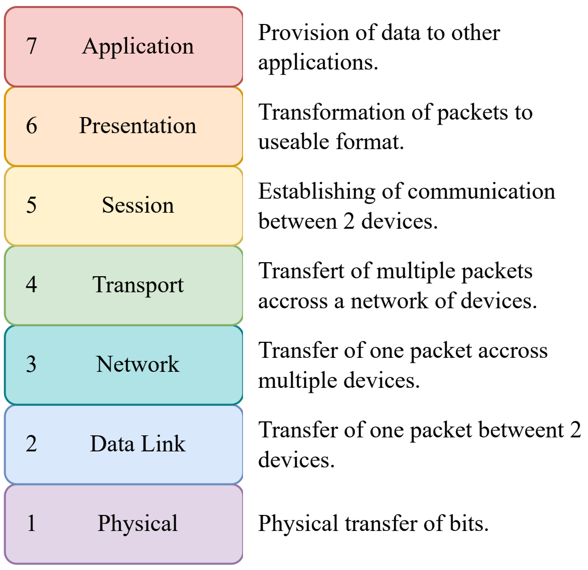

7 Application
Provision of data to other applications.
6 Presentation
Transformation of packets to useable format.
5 Session
Establishing of communication between 2 devices.
4 Transport
Transfert of multiple packets accross a network of devices.
3 Network
Transfer of one packet accross multiple devices.
2 Data Link
Transfer of one packet betweent 2 devices.
1 Physical
Physical transfer of bits.

Figure 2.4.: The 7 OSI layers.

2.3. OSI Model

The Open System Interconnection (OSI) model is a model commonly used to standardize communication between different computing systems over a network. It provides an abstraction of the communication process by dividing it into 7 layers as shown in Figure 2.4. Each layer provides functionality to the layer above while relying on the functionality of the layer below [14].

The OSI model is commonly used to describe complex communication networks. Similar abstractions can be made for other, simpler communication interfaces. The interface specification generally resides in the physical layer, as it describes how single pieces of information (i.e., bits) are transmitted. The reassembly and interpretation of data occur in the higher layers of the model.

2.4. FPGA

Field-Programmable Gate Arrays (FPGAs) are integrated circuits that can be programmed to implement a wide range of logic circuits. At their core are *logic blocks*, which can be configured to implement combinational logic functions like AND or NOT, or sequential logic with flip-flops. They can also be used as Random Access Memory (RAM) in some cases [15]. A logic block consists of so-called *slices*, containing the Look-Up Tables (LUTs) and flip-flops to implement its functionality. Additionally, most FPGAs include Embedded Block RAM (EBR), which provides a more area-efficient way to store data compared to using flip-flops.

Unlike microcontrollers, the design on a FPGA is programmed using a Hardware Description Language (HDL), which describes the desired behavior of the FPGA in terms of its inputs, outputs, and internal logic. This allows FPGAs to perform many different operations in parallel, making them more powerful than microcontrollers in some applications [16]. Some popular HDLs are VHDL, Verilog, and SystemVerilog.

The Register Transfer Level (RTL) is an abstraction in the digital circuit design, where a circuit is described by the flow of signals between hardware registers. Those registers are usually implemented with flip-flops. Between the registers, logical operations are performed on the signals. An RTL design is structured in so-called modules. A module describes a complete logic circuit, e.g. a counter or a shift register, that can be used multiple times in other modules. The module where the functionality of all modules is combined is referred to as the `top` module.

After the HDL-code is completed, the design is transformed from an RTL description to a *netlist* that describes the design in terms of gate-level logic functions. This process is known as *synthesis*. During synthesis, the synthesis tool optimizes the design by removing unused components and simplifying the logic where possible. The netlist is then adapted for the components of a specific FPGA model during the mapping process; before the physical layout on the FPGA is computed during the Place and Route (PAR) process. To optimize placement and routing, constraints such as timing and temperature can be specified before starting the PAR process. As the final step of the FPGA design flow, the bitstream used to program the FPGA is generated from the netlist.

Before using an RTL design on an FPGA, it should be thoroughly tested and verified in software. This can be done using simulation software, where the internal signals of the design can be observed at any point during the simulation. The input stimuli for the simulation are applied from a testbench program, which is also written in an HDL. The outputs can then be verified by comparing them to values computed beforehand or simultaneously by a reference model, known as a *golden model*.

An RTL design can be tested and verified at various stages of the design flow. Pre-synthesis simulation assumes ideal hardware without internal delays. At this stage, all signals of the design can be observed because the modules have not been optimized or restructured by the synthesis tool. *Post-synthesis* simulation shows the behavior of a design after it was optimized by the synthesis tool. Although the simulation still assumes ideal hardware conditions, the design is now in the form that will be used on the FPGA. For a more complete analysis, a *post-implementation* simulation based on the results from the PAR process can be used. This simulation includes the timings within the FPGA and thus allows for accurate timing analysis.

2.4.1. Clock Domain Crossing

A clock domain refers to a section of a circuit that is driven by a single clock signal. In this thesis, there are two clock domains: one for the SPI clock (SCLK), and one for the FPGA clock. When a circuit has multiple clock domains, special care must be taken when interactions occur between them. Clock domains can differ in their clock frequencies and phases, making the relationship between the clock edges in the two clock domains unpredictable. As a result, if a signal changes near the active edge of the clock signal in the destination clock domain, it may be registered incorrectly by a flip-flop This issue is also known as *metastability*. To avoid this issue, special logic is used to handle a transition across a clock domain crossing (CDC).

There are several ways to handle this problem, with some presented here:

Handshakes: With handshake signals, two clock domains can communicate to ensure that the data is stable when registered by a flip-flop and therefore avoid metastability.

FIFO buffer: A dual port first in – first out (FIFO) buffer can be used for clock domain crossings by writing from the source clock domain and reading from the destination clock domain. The `empty`-flag is synchronized to the destination domain to make sure the data arrives correctly.

Dual port ram: Dual port RAM allows simultaneous `read` and `write` operations by both clock domains. However, the data is metastable during a `write` operation [17]. To prevent this, access from the other clock should be blocked when one clock writes to a register.

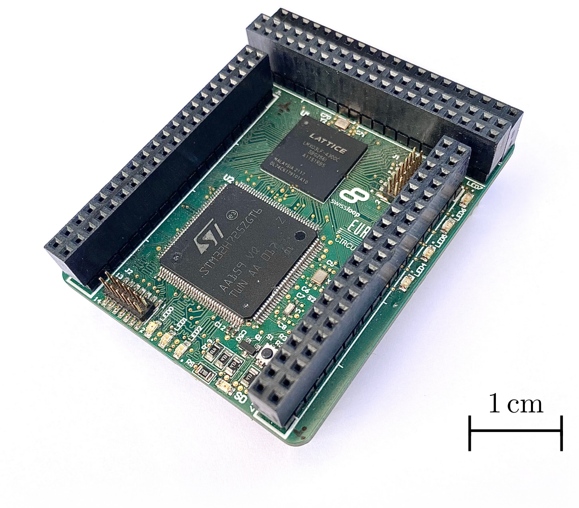


Figure 2.5.: Inverter Control Unit (ICU) control board.

2.5. HLS

High Level Synthesis (HLS), also referred to as *C synthesis*, is a design process that automatically converts a high-level specification of a digital circuit to an RTL implementation. The high-level description is provided on an algorithmic level and is most commonly implemented in C. The HLS compiler compiles the formal, high-level language to an internal, often graph-based representation of the implementation, which is then optimized and implemented in an HDL [18]. HLS aims to provide better and more efficient access to the computational power of FPGAs for developers by allowing them to implement a design on the algorithmic layer of abstraction, while the HLS compiler handles the RTL implementation. The algorithms used by the HLS compiler to optimize the design are an essential component to this process, and as such is an active field of research [19].

2.6. Demonstration Hardware

This section aims to provide an overview of the hardware used for this thesis, in particular the hardware developed and provided by Swissloop. The interface was designed for implementation on the ICU control board integrated on the Levitation Control Unit (LCU) used in the pod developed from 2022 to 2023. The ICU control board is used to control the electromagnets in the levitation system and the motor.

The ICU control board is equipped with a microcontroller from STMicroelectronics and an FPGA from Lattice. The FPGA controls the current in the electromagnets, while the microcontroller collects the data from laser sensors measuring the lateral deviation of the pod from the track. Based on those readings, the microcontroller computes the necessary current in the electromagnets to stabilize the pod. The current targets are transferred to the FPGA on the control board over the interface developed in this thesis, allowing for adjustment of the current in the electromagnets. The ICU control board is additionally used in the motor system, where it performs a similar function.

The chips are programmed over a JTAG interface with the help of programming devices from STM and Lattice. The layout of the ICU control board provides access to various signals, including all signals of the SPI bus between the microcontroller and the FPGA, over sockets at the edges of the printed circuit board (PCB). A USB to UART adapter is used to supply 3.3 V to the ICU control board.

2.6.1. Microcontroller – STM32H725

The microcontroller integrated on the ICU control board is the `STM32H725ZGT6` from STMicroelectronics' STM32H725 line. The microcontroller features a 32-bit processor architecture and can operate at frequencies of up to 500 MHz. The microcontroller comes with 97 configurable I/O pins, that can support up to 6 integrated SPI-peripherals [20]. The instance connected to the FPGA is SPI5, which supports clock frequencies of up to 49.2 MHz. The applied voltage level is 3.3 V on all lines. The SPI peripheral is configured in mode 0 (Table 2.1), and sends the data in packets of 8 bits.

The microcontroller can be set up through the CubeMX software released by STMicroelectronics. CubeMX provides a Graphical User Interface (GUI) allowing developers to configure ports, peripherals, clocks, and more without having to write low-level code. When the configuration in CubeMX is complete, the corresponding initialization code is automatically generated [21].

STMicroelectronics provides a Hardware Abstraction Layer (HAL) library containing drivers for its STM32 microcontrollers. The library provides functions to simplify the interaction between the software and hardware layers by providing a high-level interface to the microcontroller's peripherals. The library is automatically configured by the CubeMX software based on the user's configurations [22].

2.6.2. FPGA – Lattice MachXO3

Integrated on the ICU control board is the `LCMXO3LF-9400C-5BG256i` FPGA from Lattice's MachXO3 family. It features 9400 LUTs, 48 blocks of 9 kB (432 kB total) of EBR, and has an internal clock capable of running at up to 400 MHz [15]. The FPGA on the ICU control board is configured with a 66.5 MHz clock. The I/O pins for the SPI peripheral are set up with a voltage level of 3.3 V.

Lattice provides its own Integrated Development Environment (IDE) called Lattice Diamond for programming their FPGAs. This design software includes the Synplify Pro synthesis tool by Synopsys, and the Modelsim simulation software by Siemens and offers several tools like IPexpress or the power calculator. IPexpress is a tool that simplifies the generation of IP modules from Lattice, such as buffers or RAM units based on the EBR [23]. The Power Calculator can be used to calculate the power consumption of an RTL design on a Lattice device after running PAR.

3. Related Work

This chapter first presents two examples of SPI implementations from Texas Instruments and NXP Semiconductors. Afterward, some projects implementing code generation are introduced. This collection is focussed on code generation specifically for FPGA, without considering HLS.

3.1. SPI

The BQ76942 by Texas Instruments [24] is a high-accuracy battery monitor and protector for various lithium-ion battery packs. It features multiple serial interface peripherals, such as I^2C and SPI, for integration in other systems. The SPI interface is used in Motorola mode 0 (Table 2.1). A transaction consists of two packets of 8 bits, with an optional third packet for CRC error correction. The first packet consists of one bit indicating the operation (read or write), followed by 7 bits defining the address. The minimum clock period is 500 ns, which corresponds to a maximum frequency of 2 MHz.

The PCA8561 by NXP Semiconductors [25] is an ultra-low power Liquid Crystal Display (LCD) segment driver, which is connected over either I^2C or SPI. For the SPI interface, the MOSI and MISO lines are combined into one bidirectional data line, where data is sent in packets of 8 bits. In this half-duplex implementation, the SCLK runs at up to 5 MHz. The protocol does not specify the number of packets in a transaction. Instead, the first packet indicates the operation and the address, before data is transmitted until the CS is deactivated. The start address is specified in the lower 4 bits of the first packet. This address is incremented automatically during a transaction, allowing for the registers to be streamed over the interface without having to specify a new transaction. The device has 10 registers which are 1 byte wide.

3.2. Code Generation

While generating code based on predefined configurations is not a new concept, its application in generating HDL code is not widely adopted. In the following, a selection of projects utilizing HDL code generation is presented.

3.2.1. Protocol Buffers

The Protocol Buffer (protobuf) data format is Google's solution for serialization of structured data. The data is compiled by the `protoc` compiler, which produces output in various programming languages. However, there are no officially supported compilers for HDLs like SystemVerilog available [26]. Despite this, several projects have successfully utilized protocol buffers to automatically generate HDL code from protobuf definitions, as presented in the next section.

3.2.2. Machine Learning Projects

Artificial neural networks have become increasingly relevant in recent years for their ability to perform various classification tasks. Despite their computational complexity, their structure allows for parallel computations, making them ideal for accelerators on FPGAs.

Yijin Guan et al. [27] proposed FP-DNN, a framework for automatically creating FPGA-based hardware accelerators for Deep Neural Networks (DNNs) to take advantage of the computational power of FPGAs. DNNs are both computationally and memory-intensive due to their complex topological structures and the large amount of data they process. In the FP-DNN framework, a model description is generated in the protobuf format using Tensorflow [28]. Based on this description, the framework generates all C++ and Verilog code for the accelerator by using HLS. The entire process is completed without any human intervention.

Emanuele Del Sozzo [29] proposed a similar CNN framework for creating an FPGA-based accelerator for Convolutional Neural Networks (CNNs). In this framework, the user provides a high-level description of the CNN, which is then converted into an internal representation based on protocol buffers. This representation is used to produce a C++ implementation of the CNN and all necessary scripts to automate the bitstream generation utilizing HLS.

In 2017, Chang et al. [30, 31] introduced Snowball, a compiler for their Snowflake accelerator designed for CNN architectures. Snowflake was implemented on a XILINX Zynq XC7Z045 FPGA and uses its own instruction set. Snowball loads the parameters of a PyTorch [32] model description, and creates the instruction set for the Snowflake accelerator, similar to HLS.

A more detailed presentation and discussion of FPGA-based accelerators for CNNs can be found in the paper by Venieris et al. [33].

3.2.3. OpenTitan

OpenTitan [34] is an open-source project aiming to provide a reference design and integration guidelines for silicon design. It is developed in Python by engineers and researchers from ETH Zurich, Google, lowRISC, and more. OpenTitan provides a tool, referred to as *regtool*, to construct register RTL designs, C header files, and register documentation. It can output Verilog code, various C header files, HTML documentation, and more. The input is a Hjson, which is a variant of JSON, containing a comprehensive description of the registers, which allows for extensive documentation to be generated [35]. The code generation is done individually for every output, requiring the user to select one output file to be created.

3.2.4. Reginald

Reginald [36] is an open-source project developed by Philip Schilk at the Center for Project-Based Learning (PBL) at ETH Zurich. It is a tool that automatically generates macros and helper functions in C for accessing registers of sensors or microcontrollers. The tool retrieves register parameters from a YAML configuration file and generates a library of enums, macros, and functions. Additionally, it includes a generator for creating documentation in Markdown format. Reginald parses the YAML file and generates the desired code as a list of formatted strings. Once all strings are generated and added to the list, the text is saved to a file with the corresponding filename extension.

4. Design and Implementation

This chapter discusses the design decisions made for this thesis. Section 4.1 provides an overview of the implementation with a focus on its features and application, before the different components are explained in more depth in the subsequent sections. The code for this project is linked in Appendix C.1, along with an overview of the project structure. The project is documented in code with Doxygen-style comments that explain every function. Additionally, a user manual in the form of a Markdown document is made available alongside the project, providing detailed instructions on how to use the interface. This style of documentation was chosen to create a clean project structure, enabling the users to quickly find the necessary information, while still having access to complete documentation of the project.

4.1. Design

Figure 4.1 presents an overview of the implemented interface. At the center of the interface is a register file, an array of registers implemented on the FPGA, which enables the exchange of data between the microcontroller and FPGA. The register file can be accessed by the microcontroller over SPI or directly by other modules on the FPGA. A more detailed description of the register file can be found in Section 4.3.2. For the remaining chapters, the term *register* refers to an address in the register file. Other types of registers implemented on the FPGA are designated accordingly.

A key feature of this thesis is a Python program, referred to as *Interface Manager*, which can generate C++ and SystemVerilog code alike. The Interface Manager enables the user to choose the configuration of the interface by selecting the register width, register names, and various interface parameters. In addition to that, a prefix to differentiate the generated modules and functions from others can be chosen. However, for the scope of this thesis, all module and function names are used without prefixes to simplify reading.

All parameters are set in a YAML file, from which the Interface Manager retrieves the required information. The YAML format was chosen for its simple syntax and ease of parsing in Python. This makes it more favorable than other formats like JSON or XML, which have more complex syntax. An exemplary configuration file is presented in Listing 1.

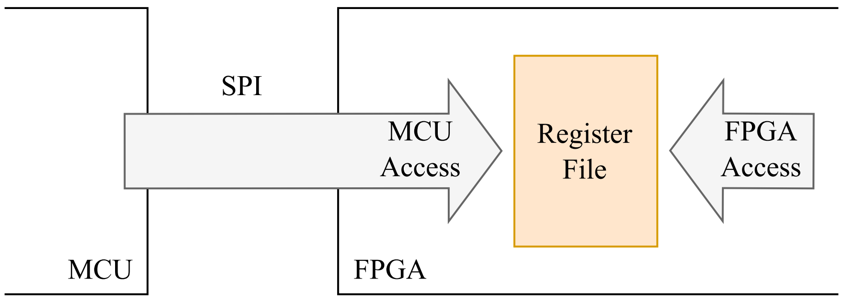


Figure 4.1.: Overview of the interface structure.

The incorporation of the interface into a project was kept simple and can be done in a few steps:

- In the SystemVerilog project, the necessary files must be included, and the top module of the interface must be instantiated within the project. The Interface Manager provides a template of the module declaration to simplify this process.
- To use the interface on a microcontroller, the hardware has to be configured by the user, e.g., with CubeMX (Section 2.6.1). Since CubeMX already offers this functionality and is commonly used, no effort was made to recreate it for this thesis. Afterward, the `spi_interface.hpp` header file must be included in the project. To synchronize the library for the interface with the hardware configurations, i.e., the pinmapping and information about the SPI peripheral, it must be initialized before it can be used. This is done using the function `spi_init()`.

To access the register file from a microcontroller, only two functions are needed: `spi_read()` and `spi_write()`. These functions handle the SPI transaction and take as arguments the target register and a pointer to the memory location on the microcontroller where the data is either sent from or written to.

To access the register file from an FPGA, direct access to the input and output data signals of the register file is provided. To select a specific register, there are `read` and `write` input signals for each register, named according to the names specified in the configuration file. If a `read` signal is selected, the corresponding register is read and the data becomes available at the output after the rising clock edge with a clock-output delay of $t_{co} = 14.013\,\text{ns}$. This delay has been determined experimentally, as explained in Section 4.3.4. If a `write` signal is selected, the data on the input data bus is written to the corresponding address at the next rising edge.

4.2. Protocol

The protocol used in this thesis is optimized for latency and throughput at the expense of address depth. A valid transaction consists of an arbitrary number of packets ranging in size from 4 to 8 bits. The structure of a transaction in a set of packets is a result of the behavior of the SPI peripheral on the STM microcontroller and its HAL library.

For the implementation at Swissloop, a packet size of 8 bits and a register width of 128 bits were chosen. The choice of register width was influenced by the performance of the rest of the implementation on the microcontroller.

The first packet of a transaction contains the opcode, which carries information about the operation type (read or write) and the register address. The most significant bit (MSB) of the opcode indicates the operation: a value of 1 means that the specified register should be written to, while a value of 0 means that it should be read. The remaining bits of the opcode decode the address. While this limits the address depth to 2^7 registers, it reduces the overhead of a transaction to only one packet. This number of available addresses was considered sufficient for Swissloop's application.

In a `write` transaction, the subsequent packets contain the data that should be written to the register. During a `read` transaction, the data on the MOSI line after the opcode is ignored. Instead, the data from the register is returned on the MISO line. Timing diagrams of both transaction types are shown in Figure 4.2.

This dense packing of the necessary information in the opcode has the consequence that the number of available address bits is strictly smaller than the number of bits in one packet.

```
# Output paths for MCU and FPGA
path_device_A: ..\MCU\source
path_device_B: ..\FPGA\source

# General project settings:
project_setup:
  project_prefix: example           # str
  interface: SPI                    # str
  generate_memory: True             # bool
  register_width: 32                # int

interfaces:
  # General SPI settings:
  SPI:
    cpol: 0
    cpha: 0
    packet_size: 8
    address_depth: 7

# Register map
registers:
  EXAMPLE_REG_0:
    adr: 0x1
  EXAMPLE_REG_1:
    adr: 0x2
```

Listing 1: Basic configuration file example with minimal documentation.

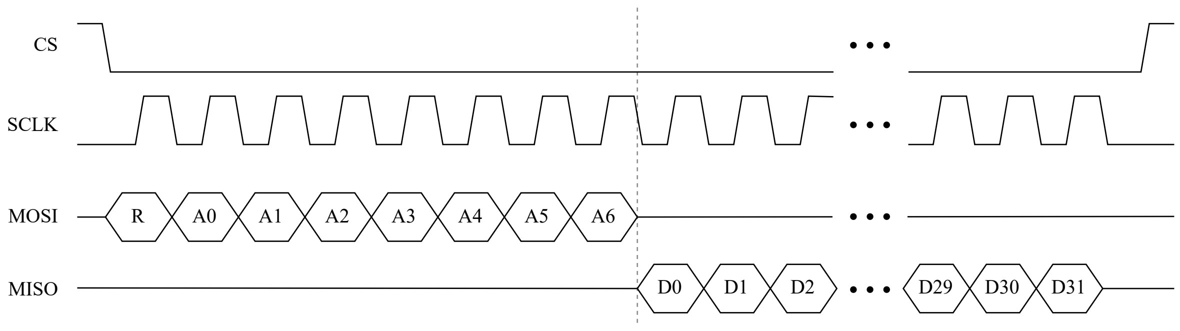
However, it reduces the overhead by requiring only one packet for the opcode instead of using one packet for the operation and one or more packets for the address. This is especially noticeable during smaller transactions.

To correctly exchange data over the interface, both sides of the interface – the SPI master on the microcontroller and the SPI slave on the FPGA – must follow this protocol. This is ensured by generating code for both the library on the microcontroller and the SPI-module on the FPGA.

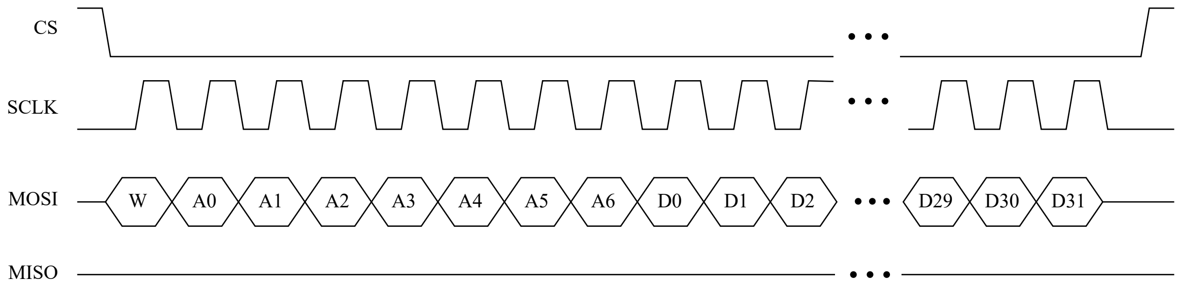
4.3. FPGA Implementation

All code for the FPGA implementation is written in SystemVerilog and complies with the lowRISC Verilog coding style [37]. During development, an older version of the ICU control board equipped with the slightly less powerful 4300C version of the FPGA was used. The `top`-module of this thesis is called `spi_peripheral`. The `spi_peripheral` instantiates the `spi_slave` module, the `spi_register_file` memory module, and provides simple access to the registers in the `spi_register_file`. These components are presented in Section 4.3.1, Section 4.3.2 and Section 4.3.3. A block diagram of the `spi_peripheral` is shown in Figure 4.3.

The `spi_peripheral` supports an asynchronous reset (`rst_ni`), implemented in the active low configuration. In practice, the reset should only be used at startup to initialize all internal



(a) Read transaction



(b) Write transaction

Figure 4.2.: Timing diagrams of read and write transactions.

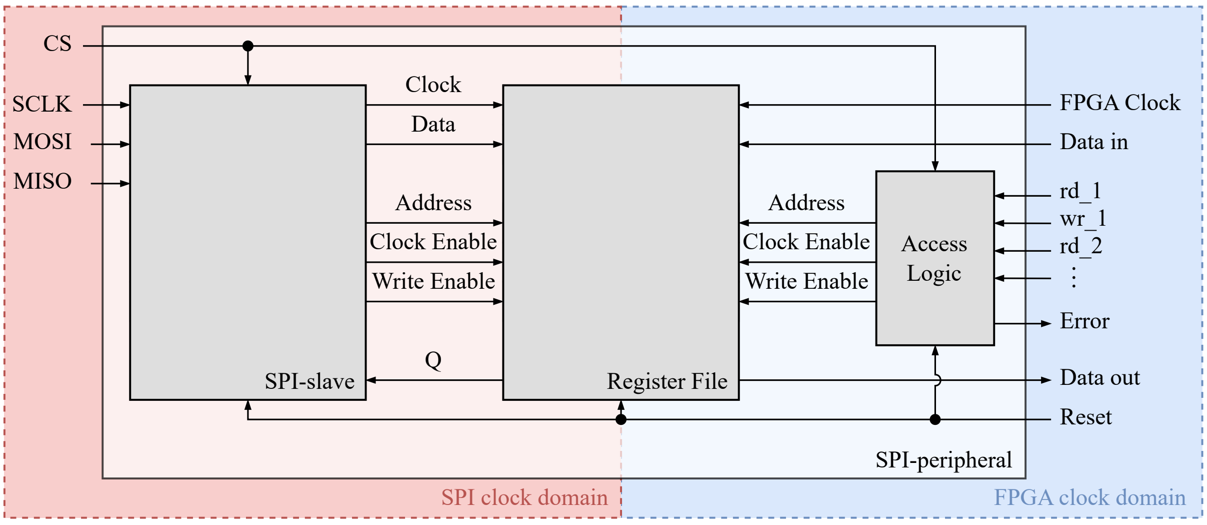


Figure 4.3.: Block diagram of the `spi_peripheral` module.

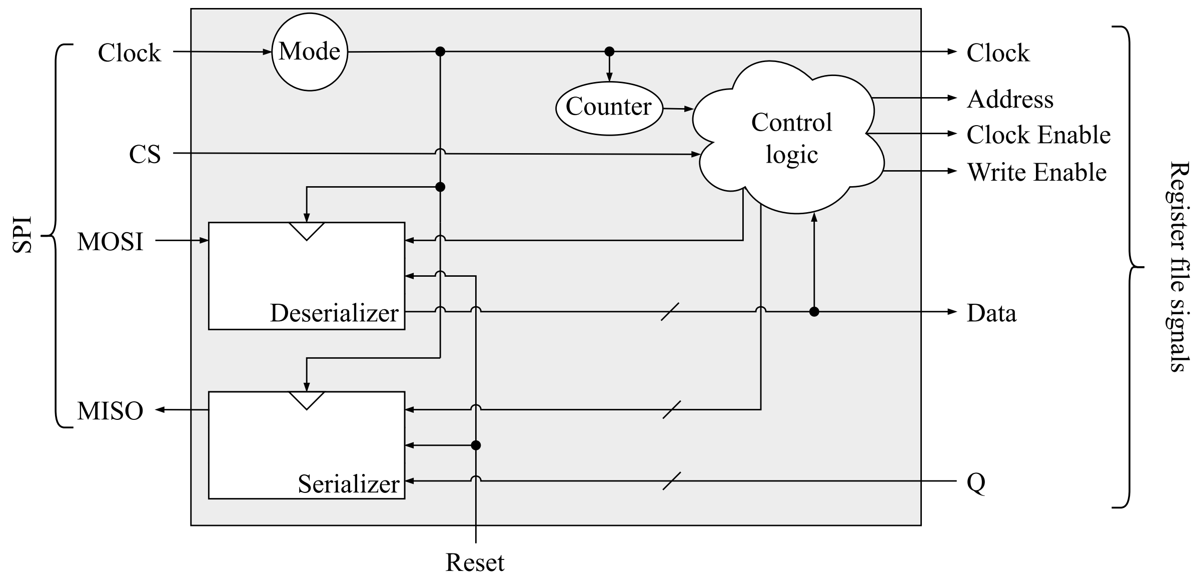


Figure 4.4.: Block diagram of the spi_slave module.

signals to known states. If the reset is used during operations, the contents of the memory would be preserved, but no read or write operations would be possible during the active period of the reset signal.

4.3.1. SPI slave

The spi_slave module is responsible for handling the SPI transaction, as described in Section 2.2. It receives the SPI signals (SCLK, CS, MOSI, MISO) as inputs and outputs control signals to access the spi_register_file. A block diagram is provided in Figure 4.4.

The clock is adapted based on the SPI mode to ensure that the data signal is always sampled on the rising edge. This is accomplished by inverting the SCLK signal in modes 1 and 2 (Table 2.1).

To convert the serial data stream from the MOSI line to a parallel representation for the register file and vice versa for the MISO line, two shift register modules, named deserializer and serializer, are instantiated. Unlike the deserializer, which shifts its contents on the rising edge, the serializer is activated on the falling edge to ensure a valid signal is present on the MISO line when it is sampled by the master. The shift registers are the same size as the registers in the register file, eliminating the need for additional logic to reassemble the transmitted data from the packets.

The spi_slave uses two counters to track the rising and falling edges. During the implementation, it was noticed that these counters were occasionally optimized by the synthesis tool in a way that caused the PAR process to fail with no clear pattern as to when this issue arose. This was circumvented by using the synkeep flag on the counters to tell the optimizer to preserve them. Based on the state of the two counters, the shift registers are controlled and the output signals are set. At the final rising edge of the first packet, the value on the MISO line is combined with the previously sent bits stored in the deserializer and written into the opcode register. The deserializer is deactivated during this time to keep its content stable. If the opcode indicates a read operation, the data stored in the corresponding register is read simultaneously and is transferred to the serializer at the ensuing falling edge. If a write transaction is indicated by the opcode, the deserializer is deactivated during the final rising edge of the transaction

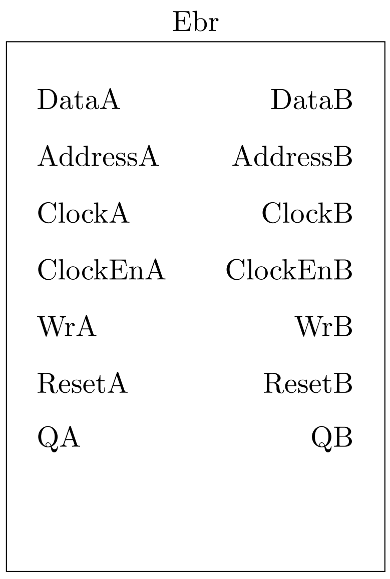


Figure 4.5.: I/O of the register file. Q denotes the output data bus.

to stabilize its content. The data is applied to the `spi_register_file` as a combination of the MOSI line and the contents of the `deserializer`, along with the necessary flags to write to the register file. At the end of the transaction, the counters are reset to their initial states.

The `spi_slave` also contains logic to handle transactions if the CPHA parameter is set to 1. In this case, the active clock edge arrives after the inactive edge, and the values in the counters are treated differently than when the active clock edge arrives before the inactive edge.

4.3.2. Register File

The register file was implemented on the FPGA using volatile RAM, which is a simple and inexpensive way to store data on an integrated circuit. It can be configured through the Interface Manager but can be replaced by any other dual-port RAM module that supports the same input/output (I/O) interface.

As shown in Figure 4.5, the register file can be accessed from two ports, i.e., from two different clock domains. If `ClockEn` is active, data is read from or written to the address specified on the `addr` bus at the rising edge of the clock, depending on the `Wr` signal.

Dual-Port RAM

To implement the register file, *true dual-port RAM* was used. This means that the RAM can be accessed from two ports independently using different clock, data, and control signals. The implementation is based on the `DP8KC` EBR module from Lattice's Primitive Library [38], which contains low-level modules that directly describe hardware components on the FPGA.

The register file module is generated by the Interface Manager to adjust the register and address size. The code for the `spi_register_file` is written in SystemVerilog and employs a code style consistent with the rest of the project. The `DP8KC` is used in a 1024×9 bit configuration, providing 9 kB of storage per module. To achieve greater register widths, multiple instances of the `DP8KC` are combined. However, this configuration limits the number of available registers to 1024. If more registers are required, a different dual-port RAM module can be created with IPexpress or another tool and instantiated in the `spi_peripheral`.

Clock Domain Crossing

The use of dual-port RAM simplifies the CDC. However, accessing a register that is being written from the other clock may still lead to metastability (Section 2.4.1). The logic for properly handling this issue was not addressed in this thesis and is left for future work. Instead of employing handshake signals to prevent simultaneous access to a register, the `ClockEn` signal on the FPGA side is blocked when the CS signal is active. To synchronize the CS signal with the FPGA clock, it is first registered twice in flip-flops, before taking effect on the `ClockEn` signal. This means that at most two active edges later, access to the register file is blocked for the FPGA clock domain. An error flag signal is used for indication if there is an attempt to access a register when the CS is active. This method works as long as the FPGA clock is faster than the SPI clock. Although not elegant, this pragmatic solution proved to be sufficient for the application at Swissloop.

FIFO Buffer

During the thesis, an alternative implementation using a dual-port FIFO buffer for the CDC was considered. This design aimed to transfer packets to the much faster FPGA clock domain as soon as possible to process the transactions more quickly. In this alternative design, every packet read by the `deserializer` was loaded into the FIFO and read from the other clock domain.

The dual-port FIFO controls an `empty` flag signal to indicate that valid data can be read. This flag is synchronized with the destination clock domain and thus guarantees the integrity of the data across the CDC. However, synchronizing the `empty`-signal with the destination clock requires an additional active edge in the destination clock domain. This led to issues when transferring data to the `serializer`, as the first edge of the data packets was used to synchronize the `empty` signal instead of presenting the data bit on the MISO line.

This ultimately led to the refusal of this design.

4.3.3. Memory Access

The final component of the `spi_peripheral` is the memory access logic from the FPGA, as described in Section 4.1. The input and output data buses are directly connected to the `spi_register_file` module. The input data is not registered, while the output data is registered by the `DP8KC` (Section 4.3.2).

The `read` and `write` input signals for each register are evaluated using `if-else` logic, as shown in Listing 2. This logic automatically sets the address and control signals, while ensuring that only one memory access takes place at any time. The logic prioritizes the topmost signal, starting from the lowest address, with `read` signals being preferred over the `write` signals. However, there is no logic to detect if a request is lost. Therefore, the top module is responsible for applying the correct signals.

The inputs take effect in the register file at the rising edge of the FPGA clock if `ClockEn` was active. If a `write` signal is asserted, the data on the input data bus is written to the register file, or, if a `read` signal is asserted, the data is read from the corresponding register address and is made available after a clock-output delay of $t_{co} = 14.013\,\text{ns}$. This delay was determined during post-implementation simulation, which is explained in the next section. A timing diagram is shown in Figure 4.6.

```
always_comb begin
  regs_addr_b = 0;
  access_regs_b = 0;
  regs_wr_b = 0;
  if (rst_ni) begin
    if (rd_example_reg_0_i) begin
      regs_addr_b = 7'h01;
      access_regs_b = 1;
    end else if (wr_example_reg_0_i) begin
      regs_addr_b = 7'h01;
      access_regs_b = 1;
      regs_wr_b = 1;
    end else if (rd_example_reg_1_i) begin
      regs_addr_b = 7'h02;
      access_regs_b = 1;
    end else if (wr_example_reg_1_i) begin
      regs_addr_b = 7'h02;
      access_regs_b = 1;
      regs_wr_b = 1;
    end
  end
end
```

Listing 2: Memory access logic from the FPGA

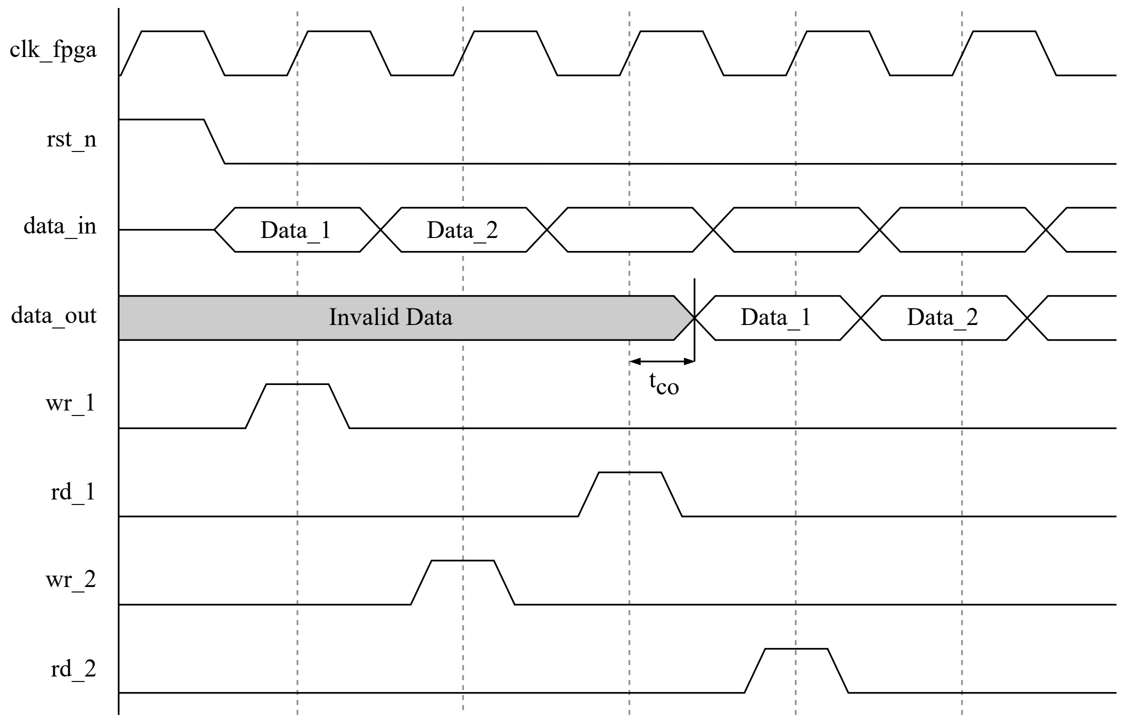


Figure 4.6.: Timing diagram of the access from the FPGA.

4.3.4. Verification

During development, all modules were tested individually prior to synthesis to progress more quickly. A Testbench (Section 2.4) applied stimuli, such as the glsspi signals and FPGA input signals, and compared the generated output on the output data bus and the MISO line to the expected values. Additionally, the functionality of the modules was verified by analyzing the waveform in Modelsim (Section 2.6.1). The testbench automatically generated a report providing information about the test cases and discovered errors.

After completing the implementation, a post-implementation simulation was used to verify the `spi_peripheral`. The simulation was based on the 9400C version of the FPGA to more accurately test Swissloop's setup. Similar to the pre-synthesis simulations, a testbench was used to analyze the performance of the implementation and to summarize it in a report.

To verify the SPI interface, the testbench first applied stimuli to the SPI input signals, sending patterns corresponding to `write` transactions to 64 registers. Afterward, it sent transactions to read from the same registers and compared the output on the MISO line. The memory access logic from the FPGA was tested next: The testbench read the first 4 registers that were written to over SPI by asserting the corresponding input signals one by one. The output on the output data bus was again compared to the expected value. Finally, the testbench wrote to 4 registers by asserting 4 `write` signals one by one and read from them again in the same way as described above, before evaluating the results. The results were summarized in a report file.

During post-implementation simulation, t_{co} was measured to be 14.013 ns. This was done using cursors on the `read` operation for the first register. The delay remained constant across a range of various temperature settings. More discussion about the delay can be found in Section 5.2.3.

For post-implementation verification, the FPGA clock frequency was configured to 33.25 MHz with the register width set to 32 bits. This clock frequency allowed the testbench to more easily verify the data output of the register file because the clock period was greater than t_{co}. The SPI clock was constrained to 25 MHz during netlist generation and was simulated at this frequency during verification. The design was tested across a range of operating temperatures from 0 °C to 80 °C, which corresponds to the *consumer grade* level typically used by large electronics manufacturers to classify their products [39, 40].

4.4. Implementation of the Microcontroller Library

The library developed in this thesis for the STM microcontroller was implemented to work in conjunction with the HAL library provided by STM. This was achieved by utilizing functions provided by the HAL library to address the CS pin, implemented as a GPIO pin, and SPI peripheral integrated into the microcontroller. The library additionally handles the protocol by sending the opcode combined with the data. To specify the target address, an `enum` providing more meaningful names to the addresses is included. Since C++ is a strongly-typed language, using an enum instead of macros helps to identify errors more quickly, because unknown register addresses are not accepted by the compiler.

4.4.1. Initialization

As described in Section 4.1, the microcontroller must be configured before the library can be used. For this task, CubeMX (Section 2.6.1) was used during this thesis. To set up the library to work with these configurations, the function `spi_init()` was implemented. This function is

```c
/**
 * @brief This (private) struct contains the hardware settings for
 * this interface.
 */
struct __example_spi_config {
  GPIO_TypeDef * cs_port;
  uint16_t cs_pin;
  SPI_HandleTypeDef spi_instance;
  bool initialized;
} example_spi_config;
```

Listing 3: Struct carrying the interface information.

used to pass information about the hardware, such as the CS pin and the instance of the SPI peripheral, to the library.

The provided parameters are stored in a struct describing the interface, shown in Listing 3. After setting these values, the configuration of the integrated SPI peripheral by CubeMX is verified. Specifically, the packet size, CPOL, and CPHA are compared to the values set in the configuration file. If the peripheral is set up correctly, the `initialized` flag is set to `True`, indicating that the interface can be used.

4.4.2. Application

As described in Section 4.1, the library provides two functions for using the interface: One for reading and one for writing. First, the opcode is created by setting the first bit according to the desired operation. Then, the opcode is copied to a `struct` (depicted in Listing 4) with the `packet` attribute, together with the data in case of a `write` transaction. This struct stores its contents in contiguous memory. Afterward, the CS signal is activated and a transaction is initiated using the `HAL_SPI_TransmitReceive()` function from the HAL library, which processes the input on the MISO line at the same time as controlling the output on the MOSI line. The data is sent from the base address of the packed struct. Thus, the opcode will be sent first, followed by the data in case of a `write` transaction. After the transaction, the CS signal is deactivated again. If a `read` transaction is executed, the data is copied from the struct to the address specified by the caller.

```c
/**
 * @brief This (private) struct is used to move the address ahead of the
 * data in memory.
 */
struct __attribute__ ((packed)) __example_spi_transaction {
  uint8_t address;
  uint8_t message[EXAMPLE_SPI_TRANSACTION_SIZE-1];
} example_spi_transaction;
```

Listing 4: Struct used to assemble the transaction.

4.5. Interface Manager

The Interface Manager is implemented in Python and follows a similar approach to the Reginald project (Section 3.2.4). Python makes it easy to work with formatted strings, which form the basis of the code generation in this thesis.

The desired parameters for the interface are specified in a YAML configuration file, as demonstrated in Listing 1. This file is parsed and the retrieved parameters are stored as attributes of a `Configuration` object, shown in Listing 5. This object is used to pass the parameters to the code-generating functions. The structure was designed to be modular and can potentially be expanded in the future to support more interfaces and protocols using the same register mapping. Before generating the code, the parameters are validated to ensure they are within the expected and supported ranges.

The code is generated by appending formatted strings to a list, one line at a time. These formatted strings are generated based on the parameters parsed from the configuration file. In the end, the strings are combined and written to a file with the corresponding file extension.

To allow for user customization in the implementation, there are sections marked as `user code`, where the contents are retained when the code is regenerated. The program searches for a file with the same path and name as the currently generated file. If it finds one, it is scanned for `user code` sections using `regex`. The content of these sections is copied to the currently generated file. If a section is not found, an empty section is created.

```python
@dataclass
class Configuration:
    """!
    @brief This object is used to hold the parameters for the generation
           of the interface
    """
    output_directory_A: str
    output_directory_B: str
    project_prefix: str
    register_width: int
    generate_memory: bool
    interface: str

    interface_param: dict
    registers: dict
```

Listing 5: Configuration object storing code generation parameters.

The generated code complies with the style conventions used throughout this thesis. For example, signal declarations in SystemVerilog files are automatically formatted and aligned. To achieve this, the length of the generated strings is compared and whitespaces are inserted where necessary. The functionality of the Interface Manager was tested and verified extensively during application for in-field testing, explained in Section 5.2.

5. Evaluation

To put the work of this thesis into context, several experiments and measurements were conducted. This chapter presents the methodology, results, and discussion for each experiment separately. Testing was done in two parts: First, the RTL design was analyzed in software after synthesis and PAR to determine its resource and power efficiency, as well as other performance metrics like the maximum frequency at various conditions.

Afterward, the microcontroller and the FPGA on the ICU control board were programmed to test the performance and reliability of the SPI interface in-field. During this process, the functionality of the Interface Manager was also verified. A link to the code used during testing and the data obtained is provided in Appendix C.2.

5.1. Post-Implementation Analysis

The post-implementation (Section 2.6.2) analysis was conducted after the PAR process was completed for the LCMXO3LF-9400C-5BG256i FPGA used on Swissloop's most recent ICU control board. At this stage, the layout and resources required for the design are known, allowing for the computation of the power consumption and timings during operation.

5.1.1. Methodology

For the post-implementation analysis, reports of the PAR process, timing analysis, and power consumption of the HDL implementation were generated and analyzed. To analyze the clock and timing behavior, the FPGA clock was set to 66.5 MHz by instantiating an oscillator, while the SPI clock was constrained to 50 MHz. The tests were set up for the LCMXO3LF-9400C-5BG256i FPGA, which is described in more detail in Section 2.6.2. For this set of tests, the temperature was constrained to 25 °C, and the `synkeep` (Section 4.3.1) constraint was removed. The strategy and constraint files can be found alongside the code (Appendix C), with the constraint file additionally listed in Appendix D

The variable parameter during this analysis was the register width of the register file. The number of used register addresses was kept constant at 8. Using the Interface Manager developed for this thesis, SPI peripherals with register widths ranging from 1 byte to 51 bytes were generated. A Tool Command Languages (TCL) script was run to synthesize and map the different peripherals before placing and routing them on the FPGA. During the mapping process, a report about the required resources for this design was printed to the console, from where it was copied to a text file. These reports provided information on the number of LUTs, slices, flip-flops, and EBRs used. During the PAR process, a timing analysis report was created by Lattice Diamond, which was used to determine the critical paths and the maximum possible clock frequencies in both clock domains.

After the PAR process was completed, another TCL script automatically started by a macro to create a new project in the Power Calculator tool, described in subsection 2.6.2, based on the newly created netlist. This enabled detailed analysis of the power consumption of numerous

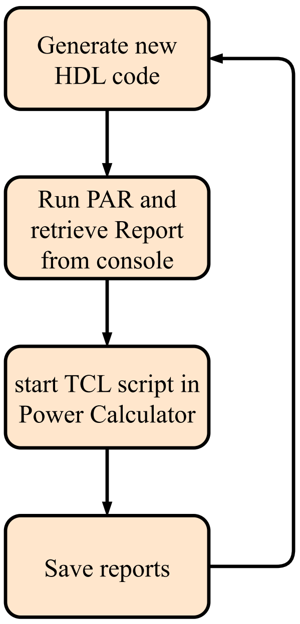

Generate new HDL code
Run PAR and retrieve Report from console
start TCL script in Power Calculator
Save reports

Figure 5.1.: Flowchart of the benchmarking script.

components in the FPGA in different configurations. All calculations by the Power Calculator were made with the ambient temperature set to 25 °C and a report with the results was provided. The workflow of this script is shown in Figure 5.1.

As mentioned in Section 4.3.4, the design was verified to work across a range of operating temperatures from 0 °C to 80 °C with the register width set to 4 bytes. During this process, a timing report was created as described above. The timing reports were copied manually before they were analyzed automatically along with all other reports.

5.1.2. Results

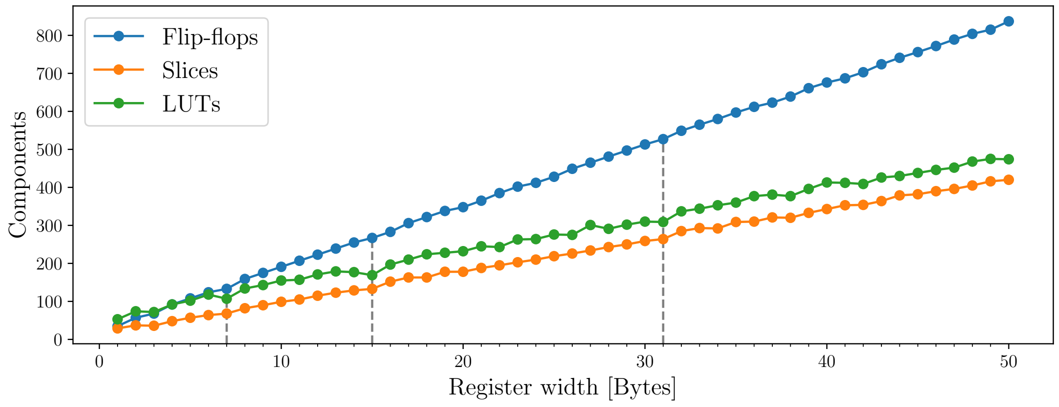
The usage of LUTs, slices and flip-flops of the implementation is shown in Figure 5.2a, with the usage of EBRs plotted in Figure 5.2b to show more details.

The utilization of all 4 components scales linearly with larger registers. There are some notable deviations in the register and slice usage where the register width is 3, 7, and 15 bytes. In the EBR plot in Figure 5.2b, there are steps after every 9 bytes.

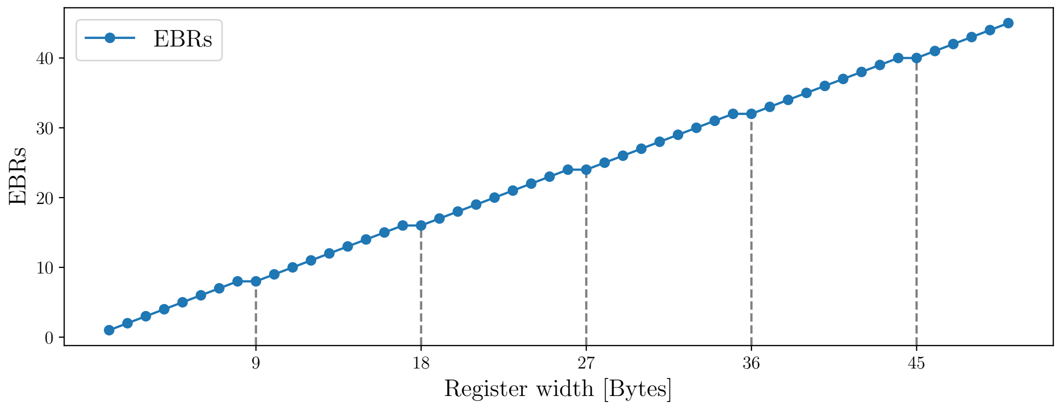
Power consumption is shown in Figure 5.2c and also shows linear scaling with some larger deviations where the register width is 3, 7, and 15 bytes.

The evolution of the maximum possible frequency after two separate runs of PAR of the same implementation is shown in Figure 5.3. The maximum frequency in the FPGA clock domain is calculated to be constant at 162.9 MHz, while it oscillates between 45.7 MHz and 60.4 MHz in the SPI clock domain. The oscillations differ between the two plots. The dashed grey line indicates a frequency of 49.2 MHz, which is the maximum frequency that the microcontroller can provide to the SPI clock. In total, there are 3 configurations where the maximum SCLK frequency is below 49.2 MHz.

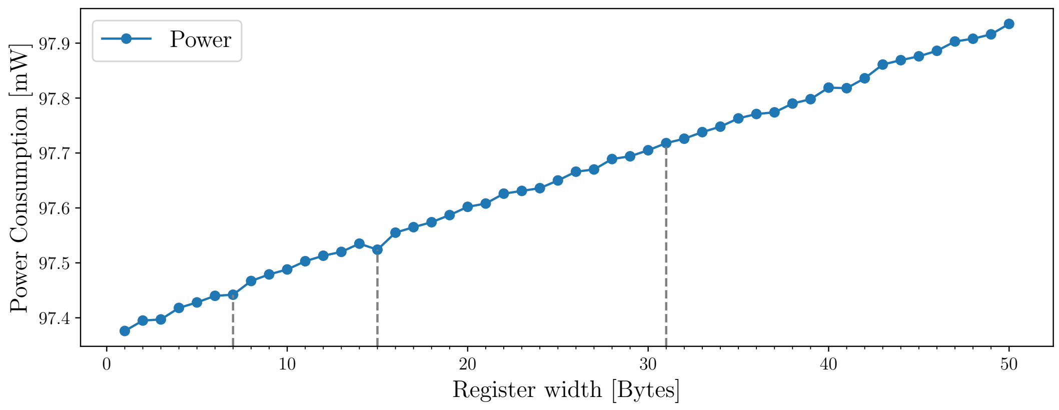
The maximum possible frequency depending on the temperature is shown in Figure 5.5. The register width was set to 4 bytes during the tests. The maximum frequency of the SCLK is decreasing by 6 MHz, from 55.3 MHz to 49.3 MHz. For the FPGA clock, the maximum frequency is decreasing from 150.2 MHz to 140.6 MHz.



(a) Various FPGA components.



(b) Embedded Block RAMs.



(c) Power consumption.

Figure 5.2.: Usage of various resources with different register widths. Interesting data points are highlighted with a vertical line, which also reveal more information about them.

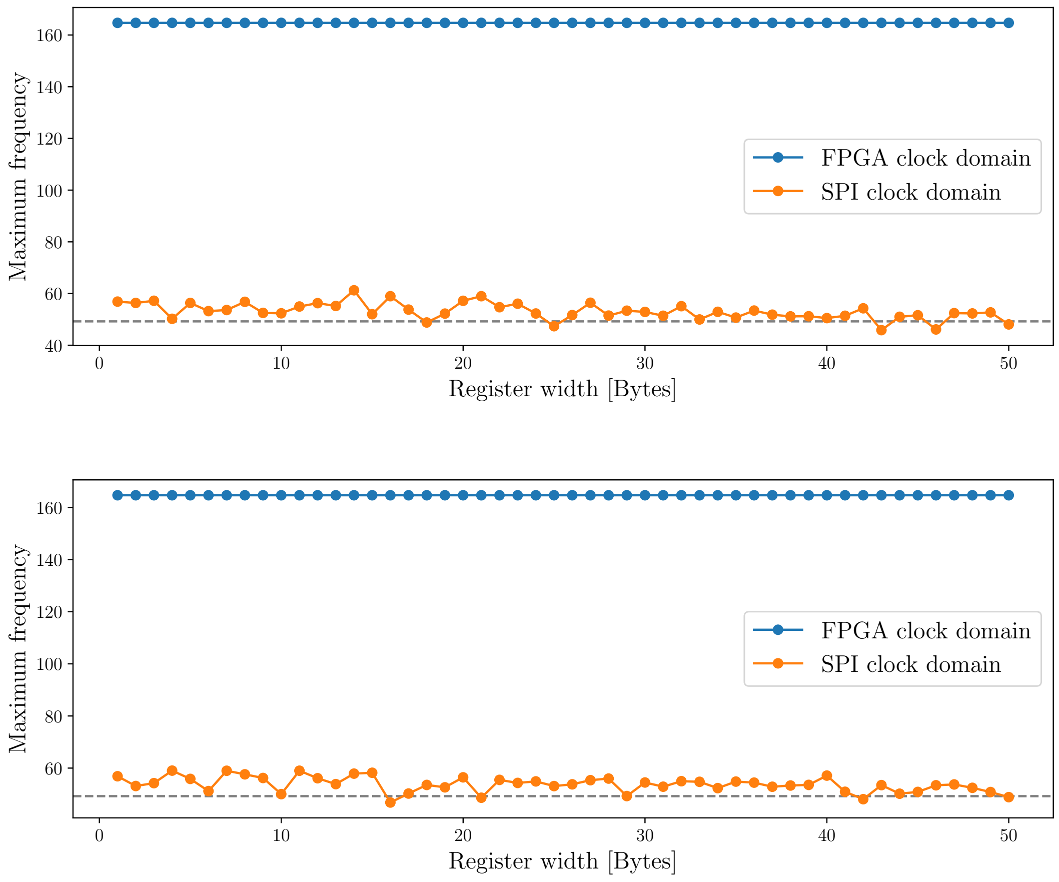


Figure 5.3.: Maximum frequency vs. data register width after different runs of PAR.

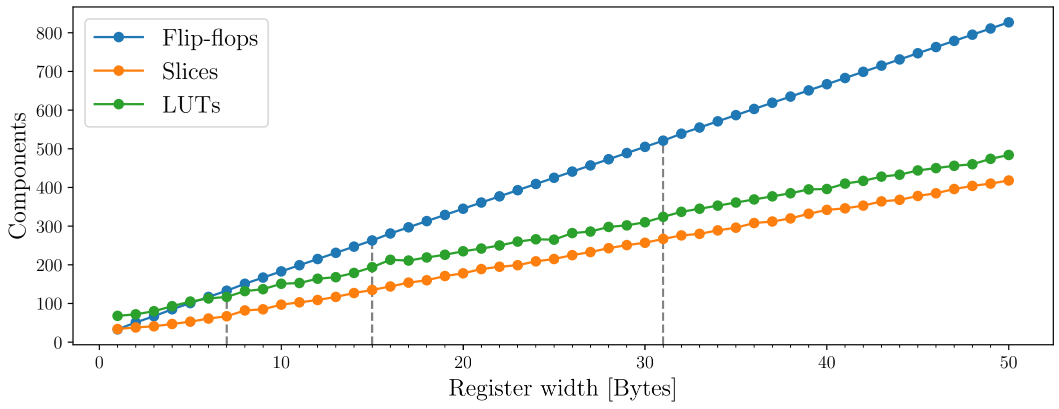


Figure 5.4.: Usage of various components with the synkeep flag used.

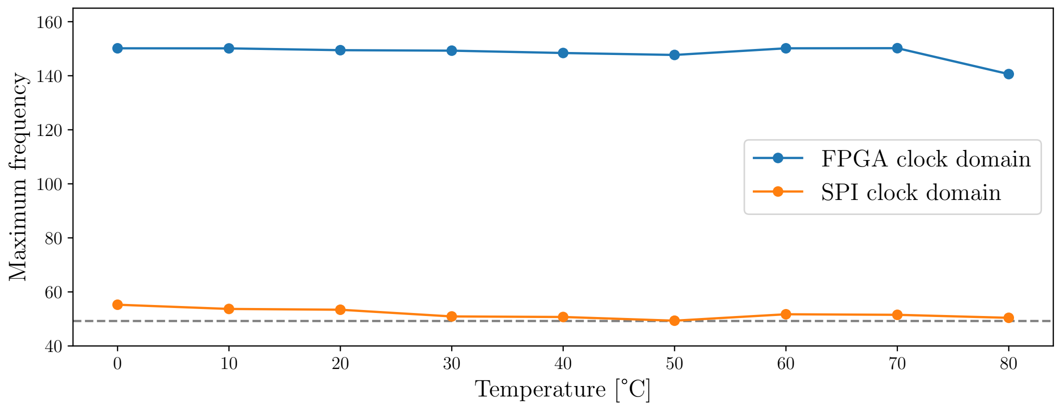


Figure 5.5.: Maximum frequency of a 4-byte register width configuration vs. operating temperature.

5.1.3. Discussion

Resource Usage and Power Consumption

Intuitively, the resource usage should depend linearly on the register size in an implementation where a dual-port RAM block is used. This is because the only changing components in such an implementation are the shift registers and counters. The resource usage in Figure 5.2a generally proves this intuition right. However, the deviations at 3, 7, and 15 bits are less expected. The deviations occur when the number of bytes in the data registers is $2^n - 1$. This pattern suggests a connection to a binary counter, potentially the one used in the `spi_slave` module, being optimized during synthesis. The results shown in Figure 5.4 confirm this explanation, as the deviations disappear when using the `synkeep` flag to suppress the optimization of unused components of the counters.

The usage of EBRs showing steps after every 9 bytes, on the other hand, is expected because the `DP8KC` primitive block is configured with 9-bit wide registers. The Interface Manager can use 8 EBR blocks for 9-byte wide registers.

Power consumption shown in Figure 5.2c scales proportionally to the resource usage depicted in Figure 5.2a. This result is expected since no other changes were made to the FPGA between tests. A constant part of the used power can be attributed to the simulated clocks and other components on the FPGA that are not part of the interface. However, this constant offset does not influence the observed scaling for larger data register widths.

Timing Analysis

The maximum frequency in the FPGA clock domain remains constant for various data register sizes. This result is expected since nothing other than the bus width of the input and output data buses changes. With a maximum frequency of 162.9 MHz, this side of the interface is sufficiently capable of running in Swissloop's application, where the FPGA clock frequency is set to 66.5 MHz.

However, the testing only covers different data register sizes, but not different address depths. This is significant because the number of input signals to the `spi_peripheral` is proportional to the number of data registers used. As a result, the data obtained during testing is insufficient to predict the maximum frequency of the `spi_peripheral` with more than 8 registers, as routing the signals efficiently becomes increasingly difficult.

Since the maximum frequency is almost 2.5 times faster than what Swissloop is using, it can be assumed that the `spi_peripheral` is still functional in their application even if more addresses are used.

The other side of the interface is less consistent, with the maximum frequency ranging from 45.7 MHz to 60.4 MHz. However, the results are inconsistent across different testing iterations. It appears that the PAR process produces different results with varying lengths in every iteration. This makes it impossible to reliably use the maximum 49.2 MHz clock from the microcontroller. On the other hand, the data shows that 24.6 MHz can be used reliably with this interface.

The latency to access the register file from the FPGA clock domain was identified in Section 4.3.4. It was measured at 14.013 ns across a range of temperatures but only for accessing one register, which is not enough to determine the maximum latency from the FPGA. Nevertheless, it provides a sensible point of reference for evaluating the interface. Specifically, the maximum clock frequency for which the data is available within one clock cycle can be derived

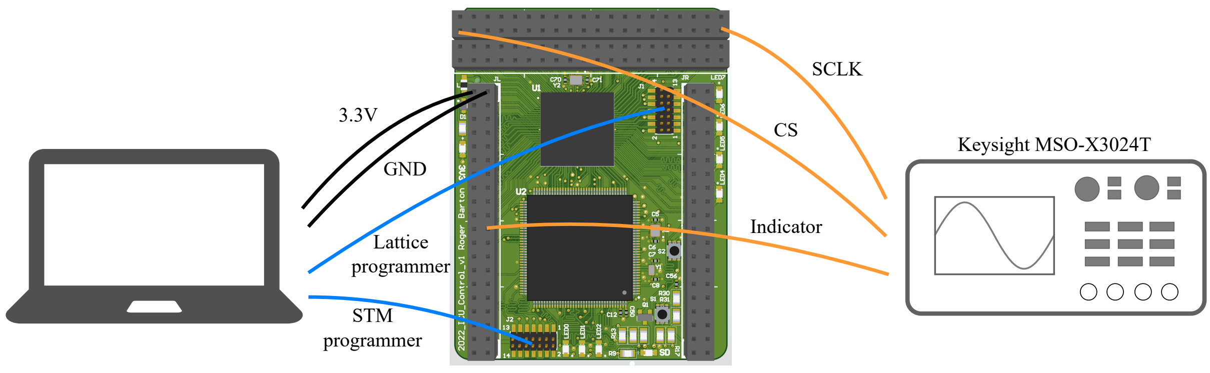


Figure 5.6.: Test setup with a Keysight MSO-X3024T oscilloscope and a ICU control board under test.

as $1/t_{co} = 71.36\,\text{MHz}$. This is faster than the frequency used by Swissloop, which makes accessing the register file in their implementation straightforward. For faster frequencies, additional clock cycles are needed before the data becomes available.

5.2. Physical Characterisation

Post-layout analysis provided insight into the internal FPGA usage and associated cost but did not provide information about the performance of the SPI implementation. To characterize the performance of the interface, the latency of a transaction was measured with an oscilloscope and a logic analyzer.

5.2.1. Methodology

A measurement of the latency was obtained by setting a GPIO pin, in the following referred to as *Indicator* signal, to high before calling `spi_read()` or `spi_write()` and setting it to low when the function returned. The duration of this signal represents the latency to access the register file from the microcontroller. To ensure that no other code was executed during testing, the code responsible for handling the testing was marked as a critical section to instruct the Operating System (OS) not to switch context during its execution.

To identify different timings during a transaction, the pins on the I/O bank on the ICU control board of the SCLK, CS, and the Indicator signals were probed. The signals were measured using a Keysight MSO-X3024T oscilloscope with a sample rate of 5 GS/s. The setup is illustrated in Figure 5.6. The measured data was exported in CSV format and plotted in Python. Additionally, the time during which the Indicator signal was active was measured using cursors on the oscilloscope to verify the accuracy of the exported data.

The measurements were made on the 2021 ICU control board, where the 4300C version of the same FPGA is installed. This version only differs from the 9400C version in a lower number of components, such as LUTs, slices, EBR units. This limited the number of samples that could be measured, as the 4300C version provides only 10 EBR units, supporting a theoretical maximum register width of 90 bits.

Transactions ranging in size from 1 to 10 bytes were measured at three different clock frequencies: 49.23 MHz, 24.62 MHz, and 12.31 MHz. These frequencies are the three fastest available to the SPI peripheral on the microcontroller.

5.2.2. Results

The most relevant plots for the discussion of the physical characterization of the interface are presented and analyzed in this section. The remaining plots can be found in Appendix E.

In Figure 5.7, the transaction with the data register size set to 8 bytes is shown for the three tested frequencies. Due to the additional opcode packet, a total of 9 bytes are sent.

First, the Indicator signal becomes active before the CS activates about 300 ns later. This delay is consistent for all three frequencies and for all transaction sizes. Both signals oscillate when one of them transitions state.

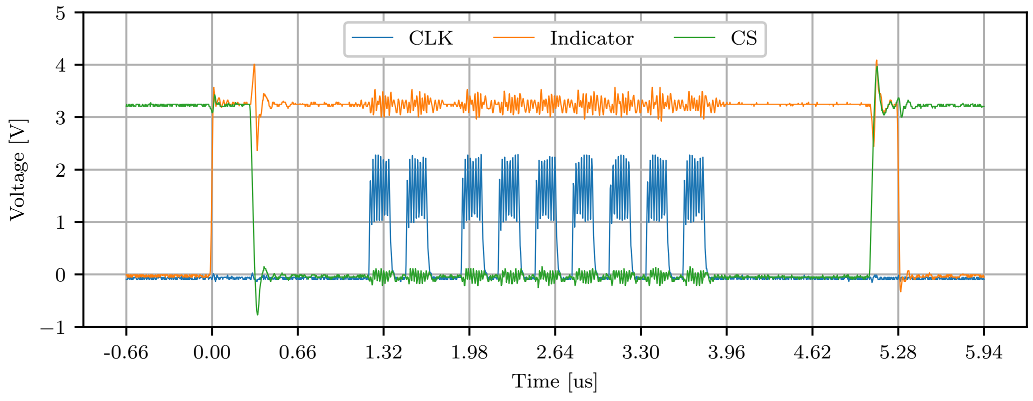
The clock becomes active about 800 ns later, but this value varies between the three clock frequencies. Oscillations can be observed on the Indicator and CS lines when it is active. The clock runs continuously at 12.3 MHz and 24.6 MHz but is active in bursts of 8 cycles with short periods of inactivity in between at 49.2 MHz. The measurements only show the clock oscillating from 0 to 3.3 V at 12.3 MHz, with less concise oscillations at the faster two frequencies. The clock runs for a total of 72 cycles.

There is a delay of about 1.2 µs before the CS becomes inactive after the last packet is sent. This value varies between the three clock frequencies, similar to the delay at the beginning of the transaction. The Indicator signal turns off about 200 ns later at all clock frequencies. Both signals show oscillations when transitioning state, similar to the beginning of the transaction.

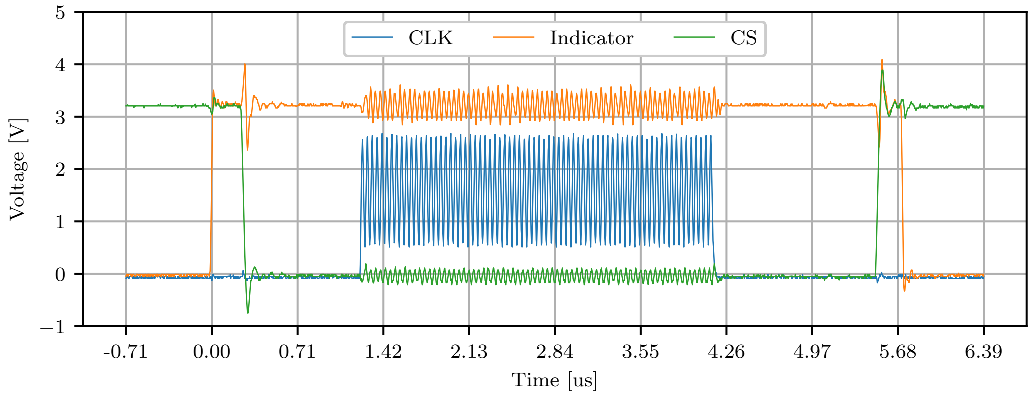
The inactive time of the clock signal during a transaction was observed to be independent of the register width but to vary between the three different frequencies. It includes the delays from the Indicator becoming active to the first clock edge and from the final clock edge to the Indicator signal deactivating. It is 2.72 µs at 49.2 MHz, 2.82 µs at 24.6 MHz, and 3.08 µs at 12.3 MHz.

The smallest transaction at 49.2 MHz is shown in Figure 5.8. It shows two packets being transmitted in two bursts of 8 bits. The Indicator signal is active for a total of 3.28 µs. The shapes of the Indicator and CS signals look similar to those in Figure 5.7, with the signal oscillating in the same locations.

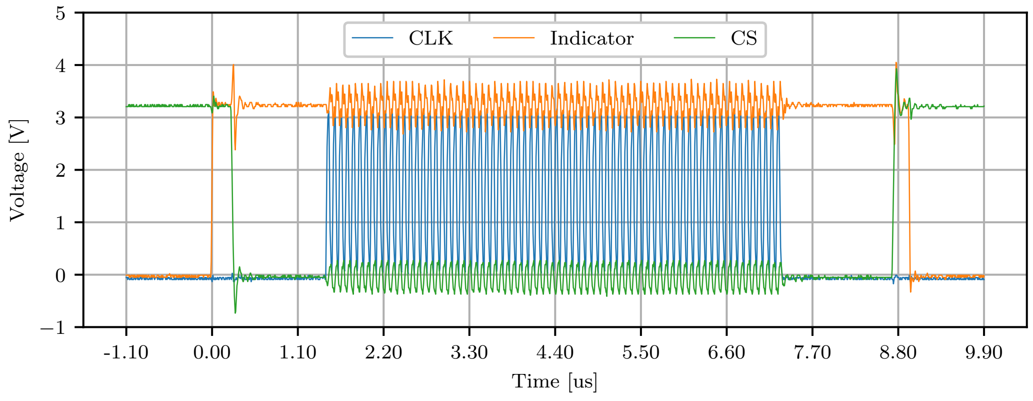
Figure 5.9 shows the latency for various register widths. A line was fitted between the data points to highlight the trend of the latency when the register width is increased. The latency scales linearly for all frequencies, therefore the trend can be expressed in the form of $mx + b$, where m is the gradient of the line and b is the offset at $x = 0$. The gradient, offset, and values at all data points are listed in Table 5.1.



(a) 49.2 MHz



(b) 24.6 MHz



(c) 12.3 MHz

Figure 5.7.: Transactions of 8 bytes at 49.2 MHz, 24.6 MHz and 12.3 MHz.

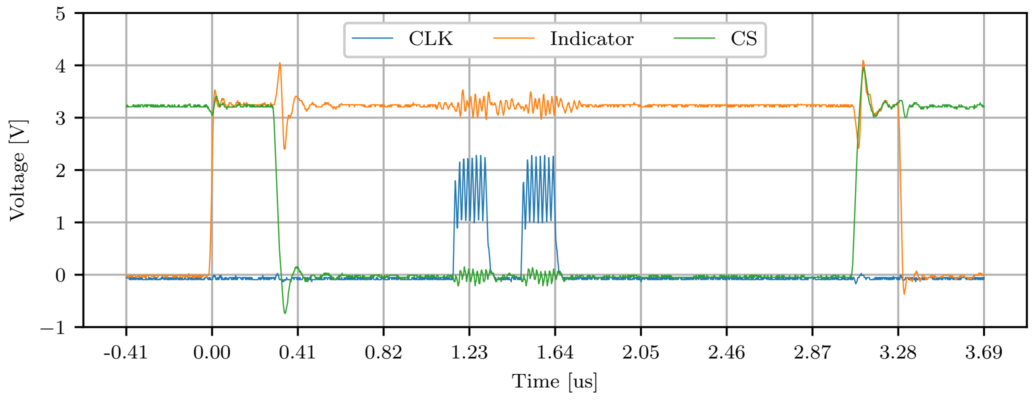


Figure 5.8.: Transaction of 1 byte with 49.2 MHz.

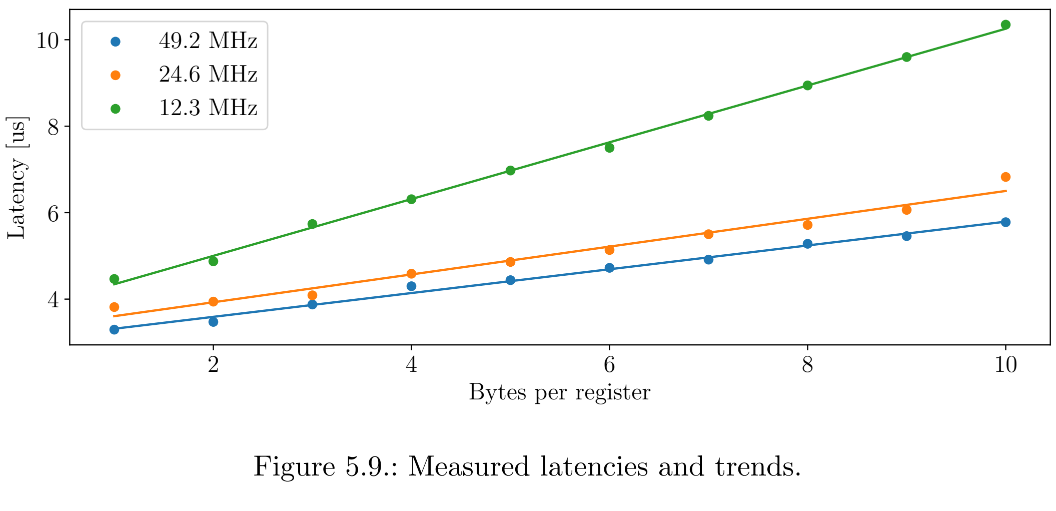


Figure 5.9.: Measured latencies and trends.

Register Width	49.2 MHz	24.6 MHz	12.3 MHz
1	3.285 µs	3.810 µs	4.460 µs
2	3.475 µs	3.930 µs	4.870 µs
3	3.850 µs	4.080 µs	5.730 µs
4	4.290 µs	4.580 µs	6.300 µs
5	4.430 µs	4.860 µs	6.980 µs
6	4.720 µs	5.130 µs	7.500 µs
7	4.900 µs	5.490 µs	8.240 µs
8	5.280 µs	5.710 µs	8.940 µs
9	5.450 µs	6.050 µs	9.580 µs
10	5.780 µs	6.810 µs	10.34 µs
Gradient (m)	0.275	0.322	0.657
Offset (b)	3.040 µs	3.284 µs	3.687 µs

Table 5.1.: Latency in various configurations.

5.2.3. Discussion

The oscillations on the Indicator and CS signals after switching states can be attributed to transient effects, which also affected the other signals over electromagnetic interference (EMI). Those that occur when the clock is active can most likely be attributed to EMI too. It is unclear whether the interference induces the oscillations on the ICU control board or in the cables of the test setup.

At higher frequencies, the clock appears to oscillate less strongly. This could be due to the oscilloscope reaching its limits in resolution, but this observation was not further investigated.

The experimental results show that the clock runs continuously at 24.6 MHz and 12.3 MHz. In these cases, the change in latency stems exclusively from the additionally sent packet:

$$Packetsize \times T_{24.6} = 8 \times \frac{1}{24.6\,\mathrm{MHz}} = 0.325\,\mathrm{us} \approx 0.322\,\mathrm{us} = m_{24.6} \tag{5.1}$$

Similar calculations can be made for the 12.3 MHz transaction.

However, the clock no longer runs continuously at 49.2 MHz; instead, packets are sent in separate bursts. This could be due to internal delays in the integrated SPI-peripheral of the microcontroller or because of delays caused by the HAL library used to access the peripheral. This means that the 49.2 MHz does not transmit data twice as fast as the 24.6 MHz clock. The scaling of the latency is only about 15% faster than for the 24.6 MHz clock.

Considering the findings of the timing analysis (Section 5.1.3) where the 49.2 MHz clock was shown to be unreliable, it can be concluded that 24.6 MHz should be the recommended clock frequency for this interface. This is still significantly faster than the SPI interfaces used on the two devices presented in Section 3.1, which run at 2 MHz and 5 MHz. The type of transaction (read or write) is irrelevant to this discussion, as the integrated peripheral on the microcontroller runs independently from the protocol. However, this result may be more challenging to reproduce due to the unreliability of this clock frequency, as explained above.

Nevertheless, the faster clock proved to be crucial to achieving the fastest transaction in a benchmarking scenario, which had a latency of 3.285 µs. It was the transaction of two packets, one for the opcode and one for data, at 49.2 MHz. The throughput of this configuration is

$$\frac{8\,\text{bit}}{3.285\,\mu\text{s}} = 2.439\,\text{Mbit/s}, \tag{5.2}$$

despite sending one opcode packet for every data packet, resulting in an overhead of 100%. The throughput improves for larger transactions, as the relative overhead decreases.

The protocol used by the BQ76942 battery monitor has 50% overhead if no CRC packet is used. However, throughput is still lower because of the 2 MHz clock. The PCA8561 LCD driver can stream data with no additional opcode packet needed, thus minimizing overhead for this application. However, with only 10 registers to read, the overhead is still comparable to the protocol implemented in this thesis.

6. Conclusion and Future Work

The interface developed for this thesis was verified to exceed the goals set in the beginning in terms of latency and throughput. It provides reliable access to a shared register file, enabling both devices to run independently from each other while still exchanging information. The interface is easily accessible and scalable, making it viable for use in future projects. This chapter aims to summarize and evaluate this thesis, before providing some ideas for future work.

6.1. Conclusion

During this thesis, the maximum reliable clock frequency for the SPI interface was found to be 24.6 MHz, which is faster than some of the applications on the market.

The register file was accessed from the microcontroller with a latency of 3.28 µs in a benchmarking scenario using a 49.2 MHz clock frequency, surpassing the goal of 100 µs more than 30 times. The goal was also comfortably achieved at the recommended frequency of 24.6 MHz. The latency on the FPGA was measured to be two orders of magnitude lower at 14.013 ns, and thus does not influence this discussion significantly.

The implemented protocol combines all necessary information to access the register file into one packet. This reduces the overhead to a single packet per transaction, therefore improving throughput. The goal of 500 kBit/s was exceeded by a factor of almost 5 with a suboptimal configuration (Section 5.2.3). This metric could even be improved by considering larger transactions.

The flexibility of the interface is mainly attributed to the Interface Manager, which was additionally developed during this thesis. It is capable of reliably adapting the interface to the needs of the user and proved to be a powerful tool during this thesis, enabling extensive tests to be automated. The Interface Manager demonstrated the benefits of using code generation for communication interfaces, as it automates the otherwise tedious and error-prone process of correctly configuring an interface between multiple devices.

6.2. Future Work

The interface was shown to be sufficient for Swissloop's application. However, due to their tight development program, it is only partially implemented on the pod in the 2022/2023 season. The implemented interface is based on an earlier development version, without EBR-based register file. Therefore, full integration is left for the next season.

The Interface Manager was designed to be expandable to more protocols. In the next step, support for more protocols and interfaces, such as I^2C or UART, could be added to generate code based on the same configuration file.

Some properties of the interface implementation could be evaluated in more detail in future work. In particular, the setup and hold times required for the access logic from the FPGA are still unknown. In addition to that, the behavior and influence of the optimizer in the synthesis tool could be investigated in more detail. While it was shown that the optimizer makes a clock frequency of 49.2 MHz unfeasible for the SPI interface, there was not enough data available to determine whether this is the case for all configurations.

Despite achieving all goals set for this thesis, the implementation still offers room for improvement. The proper handling of simultaneous access to a register file was omitted (Section 4.3.2) for this thesis but could be implemented more robustly, e.g. with handshake signals. Additionally, the protocol chosen for this thesis limits the possible address depth to 7 bits, which potentially limits the viability of this interface for other applications.

C. Project Files

This chapter covers the code written for this thesis by giving an overview of the project structures and providing links to the GitLab repositories.

C.1. Interface Implementation

The directory of the interface implementations contains the following components:

- SystemVerilog modules
- SystemVerilog verification code
- Minimum Lattice project files
- C++ library
- Interface Manager
- User Manual

The directory tree is shown and explained below:

```
/
├─ FPGA/
│  ├─ project/ ........................ Lattice Diamond project files.
│  └─ source/ ......... SystemVerilog module and testbench files.
│     └─ tb/ .......................... Testbench and report files.
├─ InterfaceManager/
│  ├─ inputs/ ............................... Configuration files.
│  ├─ source/ ............... Code generators and helper functions.
│  │  └─ spi_generators/ ................ Code generating scripts.
│  └─ configure.py ....... Main function of the Interface Manager.
├─ MCU/
│  └─ source/ ............................. Generated C++ library.
└─ README ......................... README file with user manual.
```

URL: `https://gitlab.ethz.ch/bachelor-thesis-swissloop/interface-mcu-fpga`

C.2. Benchmark Suite

The directory of the Benchmark Suite contains the following components:

- Copies of the interface implementation and microcontroller programming template provided by Swissloop
- Python scripts to automate testing
- Python scripts to analyze and present the findings of the reports
- Data obtained during physical characterization testing
- Reports from multiple benchmarking iterations
- Plots
- TCL scripts and macros to automate testing

The tree of the directory is shown below:

```
/
├── interface_icu_swlp/ ........ Template to program the MCU, provided by Swissloop
├── interface_mcu_fpga/ ........................ Copy of the interface implementation
├── measurements/ ............ Measurements obtained during physical characterization.
├── plots/ ..................................................... Generated plots.
├── reports/ ...................... Report files from multiple benchmarking iterations.
├── continue_in_power_calculator.ahk
├── main.py
├── par.tcl
├── plotters.py
├── power.tcl
└── README.md
```

URL: `https://gitlab.ethz.ch/bachelor-thesis-swissloop/benchmark-suite`

D. Testing Environment

This chapter provides additional information and resources related to the testing conducted for this thesis.

D.1. Post-Implementation Testing

Listing 6 shows the constraints used for the PAR at the end of testing. All parameters not listed here were constrained to default values by Lattice Diamond.

```
BLOCK RESETPATHS ;
BLOCK RESETPATHS ;
BLOCK ASYNCPATHS ;
LOCATE COMP "cs_ni" SITE "R5" ;
IOBUF PORT "cs_ni" IO_TYPE=LVCMOS33 ;
LOCATE COMP "sclk_i" SITE "P6" ;
IOBUF PORT "sclk_i" IO_TYPE=LVCMOS33 ;
LOCATE COMP "miso_o" SITE "T6" ;
IOBUF PORT "miso_o" IO_TYPE=LVCMOS33 HYSTERESIS=NA ;
LOCATE COMP "mosi_i" SITE "P13" ;
IOBUF PORT "mosi_i" IO_TYPE=LVCMOS33 ;
LOCATE COMP "debug1" SITE "L4" ;
IOBUF PORT "debug1" IO_TYPE=LVCMOS33 ;
LOCATE COMP "debug2" SITE "N14" ;
IOBUF PORT "debug2" IO_TYPE=LVCMOS33 ;
TEMPERATURE 25.0 C ;
FREQUENCY PORT "sclk_i" 50 MHZ ;
FREQUENCY PORT "clk_i" 66.6 MHZ ;
```

Listing 6: Contents of the LPF constraint file used during post-implementation simulation.

E. Additional Material

E.1. Oscilloscope Measurements

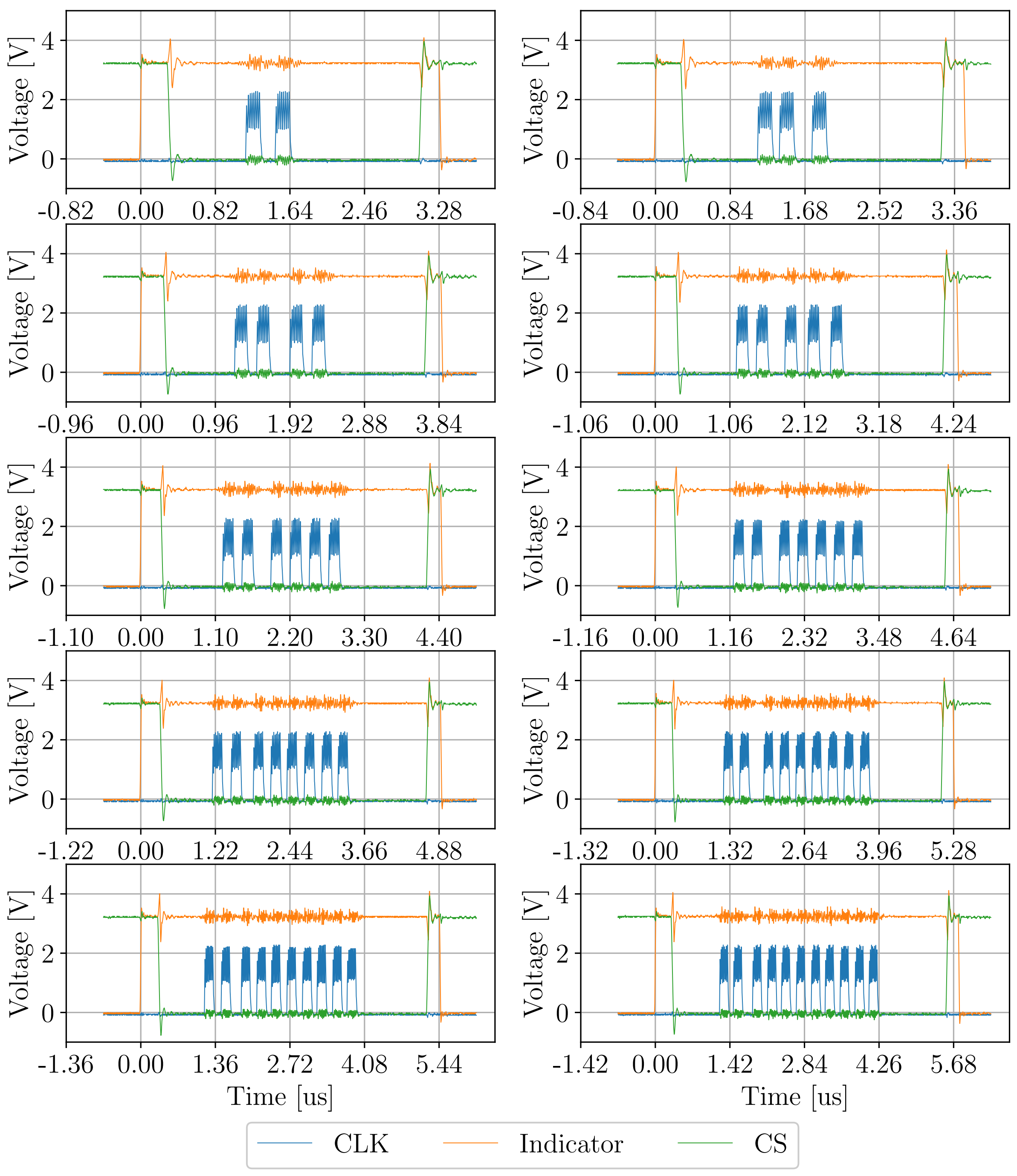


Figure E.1.: All transactions measured at 49.2 MHz.

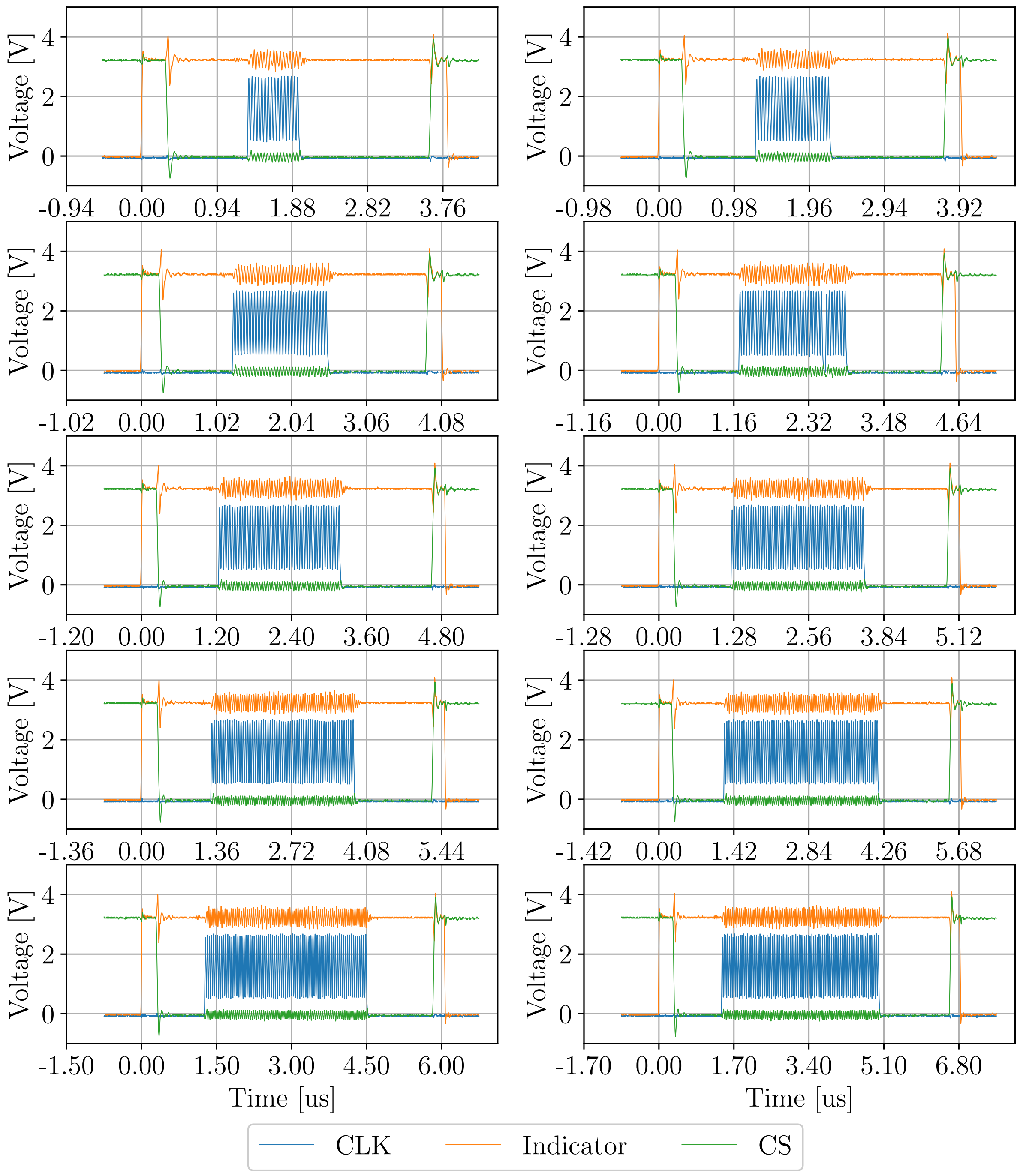


Figure E.2.: All transactions measured at 24.6 MHz.

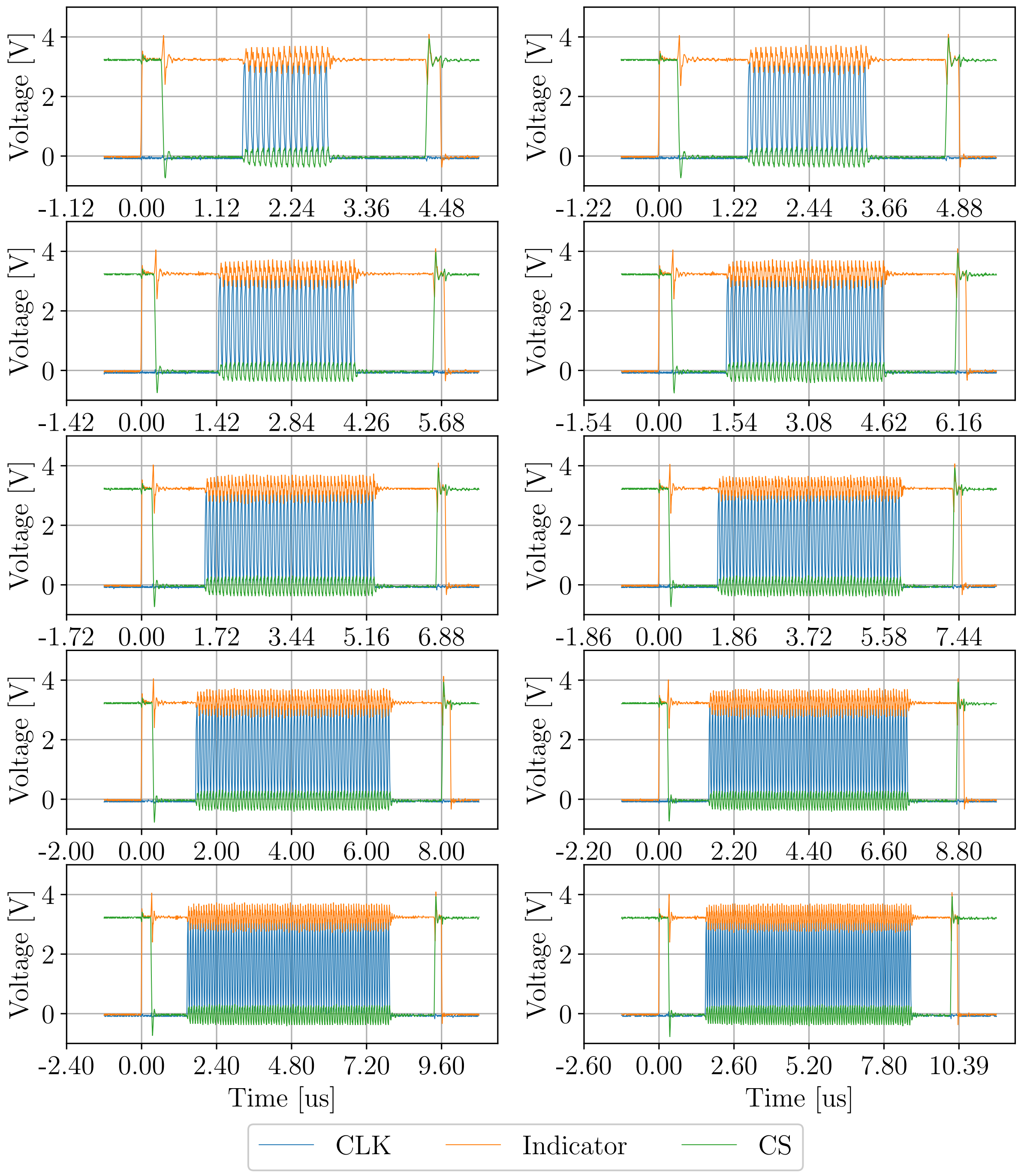


Figure E.3.: All transactions measured at 12.3 MHz.

Glossary

Active Low Active low refers to the interpretation of a signal. An active low signal is high in its idle state and is pulled to a low voltage level when activated.

Convolutional Neural Network A CNN is a type of neural network typically applied to analyze visual inputs. It applies a filter matrix, called *kernel*, to extract information and patterns from the input. In this process, it uses matrix multiplication and convolution operations, giving it its name.

Data Serialization Data serialization is the process of converting an object or data structure into a stream of bytes to more easily save or transmit it. Popular serialization formats include XML, JSON, YAML, and Protocol Buffers.

Deep Neural Network A DNN is a neural network where the inputs are processed in multiple layers before reaching the output. These layers can recognize complex patterns in the input data. Typical applications of DNNs are image and speech recognition, natural language processing, and computer vision.

Flip-flop A flip-flop is an electronic circuit that has two stable states and can store a single bit of data. The state is typically changed at the rising edge of the clock signal connected to the flip-flop, where the state of an input signal is captured.

Full-duplex Full-duplex describes a mode of communication, where data can be exchanged simultaneously in both directions. The term *half-duplex* refers to a mode where two-directional communication is possible, but the date can't be exchanged at the same time.

GPIO GPIO stand for *General Purpose input/output* and describes a pin of an Integrated Circuit (IC) with no specific function. It is fully controllable by software and can be used as either an input or an output, or both.

Latency Latency describes the time it takes for data to travel from one point to another. In a communication network, it usually describes the delay between the transmission of information from the sending device to the reception of this information by the receiving device.

Master Device In a communication interface, the master device is connected to one or multiple devices referred to as slave devices and controls the communication over the interface. Other names include *sender* and *controller*

Netlist The netlist is a representation of an electronic circuit, e.g., an IC, that describes the components of the design (LUTs, flip-flops, etc.) and the logical connections between them.

Shift Register A shift register uses a cascade of flip-flops connected to a common clock signal, which shifts the contents over in every cycle. They can be used to convert serial data to a parallel representation or vice-versa.

Slave Device In a communication interface, the slave device only communicates in response to commands sent by the master device and is unable to initiate a data transaction on its own. Other names include *receiver* and *responder*.

Slice Slices are part of logic blocks, the fundamental component of FPGAs. They contain LUTs, flip-flops, and multiplexers. The number of components per slice varies between different FPGA models. Slices are used to implement basic logic functions, which are connected to implement a logic circuit.

Swissloop Swissloop is a student-led initiative associated with the ETH Zurich, developing technology and infrastructure for eventually implementing the hyperloop concept in the real world. Every season, a new vehicle - so-called *pod* - is developed and demonstrated at the EHW competition.

Synthesis Synthesis describes the process of transforming HDL-code to a gate-level representation, called *netlist*. During synthesis, the described circuit is optimized by the optimizer in the synthesis tool. The optimizer is influenced by constraints set by the user. Synthesis can be considered analogous to the compilation process for code written in general-purpose programming languages.

Bibliography

[1] Elon Musk. *Hyperloop Alpha*. 2013. URL: https://www.tesla.com/sites/default/files/blog_images/hyperloop-alpha.pdf (visited on 06/02/2023).

[2] *European Hyperloop Week*. European Hyperloop Week. URL: https://hyperloopweek.com/event/ (visited on 06/02/2023).

[3] Piyu Dhaker. “Introduction to SPI Interface”. In: (2018).

[4] Dawoud Shenouda Dawoud and Peter Dawoud. “Serial Communication Protocols and Standards RS232/485, UART/USART, SPI, USB, INSTEON, Wi-Fi and WiMAX”. In: *Serial Communication Protocols and Standards RS232/485, UART/USART, SPI, USB, INSTEON, Wi-Fi and WiMAX*. Conference Name: Serial Communication Protocols and Standards RS232/485, UART/USART, SPI, USB, INSTEON, Wi-Fi and WiMAX. River Publishers, 2020, pp. i–xl. ISBN: 978-87-7022-153-5. URL: https://ieeexplore.ieee.org/document/9227663 (visited on 05/14/2023).

[5] Intel. “How USB Became the Most Successful Interface in Computing History”. In: (). URL: https://www.intel.com/content/www/us/en/standards/usb-two-decades-of-plug-and-play-article.html (visited on 05/24/2023).

[6] Texas Instruments. *KeyStone Architecture Serial Peripheral Interface (SPI) - User Guide*. Mar. 2012. URL: https://www.ti.com/lit/ug/sprugp2a/sprugp2a.pdf (visited on 05/14/2023).

[7] Neeraj Pandey, Nikhil Mehta, and Shreya Basu Roy. “Semiconductor Pricing Strategy in USB Market: A Market Leader’s Dilemma”. In: *Business Perspectives and Research* 5.1 (Jan. 2017), pp. 1–10. ISSN: 2278-5337, 2394-9937. DOI: 10.1177/2278533716671614. URL: http://journals.sagepub.com/doi/10.1177/2278533716671614 (visited on 05/14/2023).

[8] M. van Osch and S.A. Smolka. “Finite-state analysis of the CAN bus protocol”. In: *Proceedings Sixth IEEE International Symposium on High Assurance Systems Engineering. Special Topic: Impact of Networking*. Proceedings Sixth IEEE International Symposium on High Assurance Systems Engineering. Special Topic: Impact of Networking. ISSN: 1530-2059. Oct. 2001, pp. 42–52. DOI: 10.1109/HASE.2001.966806.

[9] Frederic Leens. “An introduction to I2C and SPI protocols”. In: *IEEE Instrumentation & Measurement Magazine* 12.1 (Feb. 2009). Conference Name: IEEE Instrumentation & Measurement Magazine, pp. 8–13. ISSN: 1941-0123. DOI: 10.1109/MIM.2009.4762946.

[10] Anand N et al. “Design and implementation of a high speed Serial Peripheral Interface”. In: *2014 International Conference on Advances in Electrical Engineering (ICAEE)*. 2014 International Conference on Advances in Electrical Engineering (ICAEE). Jan. 2014, pp. 1–3. DOI: 10.1109/ICAEE.2014.6838431.

[11] Thomas Kugelstadt. “Extending the SPI bus for long-distance communication”. In: (2011). URL: https://www.ti.com/lit/an/slyt441/slyt441.pdf (visited on 05/24/2023).

[12] Texas Instruments. *TMS320x280x, 2801x, 2804x Serial Peripheral Interface - Reference Guide*. Feb. 2009. URL: https://www.ti.com/lit/ug/sprug72/sprug72.pdf?ts=1684931613931&ref_url=https%253A%252F%252Fwww.google.com%252F (visited on 05/24/2023).

[13] Altera. *Cyclone V Hard Processor System Technical Reference Manual*. Intel. Nov. 14, 2022. URL: `https://www.intel.com/content/www/us/en/docs/programmable/683126/21-2/texas-instruments-synchronous-serial.html` (visited on 05/15/2023).

[14] J.D. Day and H. Zimmermann. "The OSI reference model". In: *Proceedings of the IEEE* 71.12 (Dec. 1983). Conference Name: Proceedings of the IEEE, pp. 1334–1340. ISSN: 1558-2256. DOI: `10.1109/PROC.1983.12775`. URL: `https://ieeexplore.ieee.org/document/1094702`.

[15] Lattice. "MachXO3 Family Data Sheet". In: (July 2021). URL: `https://www.latticesemi.com/-/media/LatticeSemi/Documents/DataSheets/MachXO23/FPGA-DS-02032-3-1-MachXO3-Family-Data-Sheet.ashx?document_id=50121`.

[16] J Serrano. "Introduction to FPGA design". In: (Nov. 22, 2016). URL: `https://cds.cern.ch/record/1100537/files/p231.pdf`.

[17] *Memory Usage Guide for MachXO3 Devices*. Mar. 2020. URL: `https://www.latticesemi.com/-/media/LatticeSemi/Documents/ApplicationNotes/MO/FPGA-TN-02060-1-2-Memory-Usage-Guide-MachXO3.ashx?document_id=50515`.

[18] M.C. McFarland, A.C. Parker, and R. Camposano. "The high-level synthesis of digital systems". In: *Proceedings of the IEEE* 78.2 (Feb. 1990), pp. 301–318. ISSN: 00189219. DOI: `10.1109/5.52214`. URL: `http://ieeexplore.ieee.org/document/52214/` (visited on 06/09/2023).

[19] Daouda Diakite. "High-level synthesis (HLS) on FPGA for inverse problems : application to tomography and radioastronomy". PhD thesis. Dec. 15, 2022.

[20] STMicroelectronics. *STM32H725xE/G Datasheet*. Dec. 2021. URL: `https://www.st.com/en/microcontrollers-microprocessors/stm32h725-735/documentation.html` (visited on 03/23/2023).

[21] STMicroelectronics. *STM32CubeMX for STM32 configuration and initialization C code generation*. Feb. 2023. URL: `https://www.st.com/resource/en/user_manual/dm00104712-stm32cubemx-for-stm32-configuration-and-initialization-c-code-generation-stmicroelectronics.pdf` (visited on 05/15/2023).

[22] STMicroelectronics. *Description of STM32F4 HAL and low-layer drivers - user manual*. Aug. 3, 2023. URL: `https://www.st.com/resource/en/user_manual/um1725-description-of-stm32f4-hal-and-lowlayer-drivers-stmicroelectronics.pdf` (visited on 05/15/2023).

[23] Lattice. "Lattice Diamond User Guide". In: (June 2011). URL: `https://www.latticesemi.com/-/media/LatticeSemi/Documents/UserManuals/JL/LatticeDiamondUserGuide.ashx?document_id=41988`.

[24] Texas Instruments. *BQ76942 3-Series to 10-Series High Accuracy Battery Monitor and Protector for Li-Ion, Li- Polymer, and LiFePO4 Battery Packs*. Dec. 2021. URL: `https://www.ti.com/lit/gpn/BQ76942` (visited on 06/12/2023).

[25] NXP Semiconductors. *PCA8561 Automotive 18 × 4 LCD segment driver, rev 5*. Mar. 15, 2021. URL: `https://www.nxp.com/docs/en/data-sheet/PCA8561.pdf` (visited on 05/05/2023).

[26] Google. *Other Languages*. Section: reference. URL: `https://protobuf.dev/reference/other/` (visited on 05/30/2023).

[27] Yijin Guan et al. "FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates". In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). Apr. 2017, pp. 152–159. DOI: `10.1109/FCCM.2017.25`.

[28] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* 2015. URL: `https://www.tensorflow.org/`.

[29] Emanuele Del Sozzo. "ON HOW TO EFFECTIVELY TARGET FPGAS FROM DOMAIN SPECIFIC TOOLS". In: (2018).

[30] Andre Chang et al. "Compiling Deep Learning Models for Custom Hardware Accelerators". In: (July 31, 2017).

[31] Vinayak Gokhale et al. "Snowflake: An efficient hardware accelerator for convolutional neural networks". In: *2017 IEEE International Symposium on Circuits and Systems (ISCAS).* 2017 IEEE International Symposium on Circuits and Systems (ISCAS). ISSN: 2379-447X. May 2017, pp. 1–4. DOI: `10.1109/ISCAS.2017.8050809`.

[32] Adam Paszke et al. "Automatic differentiation in PyTorch". In: (2017). URL: `https://pytorch.org/`.

[33] Stylianos I. Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. "Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions". In: *ACM Computing Surveys* 51.3 (May 31, 2019), pp. 1–39. ISSN: 0360-0300, 1557-7341. DOI: `10.1145/3186332`. URL: `https://dl.acm.org/doi/10.1145/3186332` (visited on 05/24/2023).

[34] *Open source silicon root of trust (RoT) — OpenTitan.* URL: `https://opentitan.org/` (visited on 06/12/2023).

[35] *reggen & regtool: Register Generator - OpenTitan Documentation.* URL: `https://opentitan.org/book/util/reggen/index.html` (visited on 06/12/2023).

[36] Philip Schilk. *Reginald.* 2022. URL: `https://github.com/schilkp/reginald/tree/80ee99a8b92125e4eaa591261bfbb12166483f45`.

[37] *lowRISC Style Guides.* original-date: 2019-06-13T10:45:22Z. May 31, 2023. URL: `https://github.com/lowRISC/style-guides/blob/7869b7d664f40dc6593bcfba747af51100fc8f7c/VerilogCodingStyle.md` (visited on 05/31/2023).

[38] *FPGA Libraries Reference Guide.* URL: `https://www.latticesemi.com/-/media/LatticeSemi/Documents/UserManuals/EI/FPGALibrariesReferenceGuide31.ashx?document_id=50452`.

[39] *Commercial and Industrial Grade Products.* URL: `https://www.cactus-tech.com/wp-content/uploads/2019/03/Commercial-and-Industrial-Grade-Products.pdf` (visited on 06/01/2023).

[40] *Extended Temperature Device and Testing Advancements for COMs and SFF SBCs Enable Reliable, Long Life Systems.* URL: `https://www.kontron.com/download/download?filename=/downloads/white_papers/whitepaper-extended-temperature_en.pdf` (visited on 06/01/2023).