Høgskulen på Vestlandet

Bachelor thesis:

FPGA solution for communication between IPbus and a power control unit using a custom USART

Martin Eggen
Jakob Hauser

28/05/2022

Document control

Thesis title: FPGA solution for communication between IPbus and a power control unit using a custom USART	*Date/Version* 28. May. 2022 / 1.10 *Report number:* BO22EB-08
Authors: Martin Eggen martineggen99@gmail.com Jakob Hauser jakob.hauser72@gmail.com	*Study:* 19HEEL *The number of pages including attachments: 53*
HVL's supervisor/guide: Svein Haustveit svein.haustveit@hvl.no	*Grading:* Open
Other notes: We accept that the report may be published	

Client: Department of Physics and Technology - University of Bergen	*Clients reference:* BO22EB-08
The client's contact person (including contact information): Johan Alme +47 55 58 28 71 johan.alme@uib.no	

Revision	**Date**	**Status**	**Performed by**
0.10	26. Apr. 2022	First draft.	Martin Eggen & Jakob Hauser
0.20	3. May 2022	Second draft.	Martin Eggen & Jakob Hauser
0.21	6. May 2022	Restructure.	Martin Eggen & Jakob Hauser
0.30	19. May 2022	Third draft.	Martin Eggen & Jakob Hauser
1.00	26. May 2022	Release draft.	Martin Eggen & Jakob Hauser
1.10	27. May 2022	Release version.	Martin Eggen & Jakob Hauser

Acknowledgements

First and foremost, we would like to thank our supervisor, Professor Johan Alme, for being a great supervisor. Helping us throughout our whole project of writing this bachelor. From the very start, he was very helpful and encouraging giving us a good start to the project. He could help us with every part of the process as well. No matter if it was giving technical advice, teaching about GitHub and Linux, or helping us write the thesis.

We also want to thank our project neighbours Birger Olsen and Håvard Birkenes for being very active in the project, and always prepared to do joint testing or problem-solving. As well as being very helpful and willing to help, when asked about parts of the project.

A big thanks also has to go to our long-time lecturer and supervisor, Assistant Professor Svein Haustveit, for giving us the necessary knowledge to fulfil our task at hand. Teaching us to look at VHDL code as physical circuits rather than just code. Giving us a more in-depth understanding of how our written code would translate to circuitry.

Abstract

The proton Computed Tomography (pCT) project in Bergen is a project aiming on designing a CT-scanner prototype using protons instead of photons. The current solution is using multiple layers of Monolithic Active Pixel chips (MAPS), called ALPIDE, to create a three-dimensional image scan. The ALPIDEs were developed at CERN for the inner Tracking System in a large ion collider experiment (ALICE).

A stable and reliable power supply chain is needed during operation. For that, a power supply is used for each layer, distributing even power to all layers. Custom PCBs housing microcontrollers are installed between the power supplies and the sensor layers. The PCBs are named Power Control Units (PCU). They are responsible for enabling the power to the layers and cutting the power if monitored currents, voltages or temperatures exceed programmable thresholds. An Ethernet connection is set up between the FPGA board and the control room computers, using a special protocol called IPbus developed at CERN. The PCUs use a custom Universal Synchronous/Asynchronous Receiver/Transmitter (USART) protocol for communication with the FPGA board. This thesis is a detailed VHDL solution for the FPGA that is responsible for this communication. The design consists of multiple different components that are to function on their own as well as in combination. For each component, a testbench is made, and documentation is presented. In the tests, the UVVM library [1] has been used to provide a structured methodology. The work is also available in a git repository [2].

The thesis contains a complete, and functioning solution for communication between IPbus and multiple USART slaves. In addition, two additional modules are planned for the design: (1) a version module including a version number and the git hash from the repository, and (2) a local housekeeping unit monitoring board-specific parameters such as current consumptions, voltage levels and temperatures. These modules were not realized due to time constraints.

The design has been simulated and verified using structural verification methodology. Hardware tests using an IPbus connection and preliminary versions of the PCU have also been conducted. These tests prove the design is operating as specified by the requirements.

Acronyms

ALICE	-	A Large Ion Collider Experiment
BUS	-	Electrical conductors that carry digital signals between components
DUT	-	Device Under Testing
FPGA	-	Field-programmable gate array
GUI	-	Graphical User Interface
HW	-	Hardware
I2C	-	Inter-Integrated Circuit
IPbus	-	Control link by CERN
LED	-	Light-Emitting Diode
LSB	-	Least Significant Bit
LVDS	-	Low-voltage differential signalling
MCU	-	Microcontroller
MJCU	-	Martin & Jakob’s Communication Unit
MSB	-	Most Significant Bit
MUX	-	Multiplexer (digital logic)
PCB	-	Printed Circuit Board
pCT	-	Proton Computed Tomography
PCU	-	Power Control Unit
PSU	-	Power Supply Unit
SOBP	-	Spread-Out Bragg Peak
SW	-	Software
UiB	-	Universitetet i Bergen
USART	-	Universal Synchronous/Asynchronous Receiver/Transmitter
UVVM	-	Universal VHDL Verification Methodology
VHDL	-	The Very High-Speed Integrated Circuit Hardware Description Language

Table of Contents

Document control 2
Acknowledgements 3
Abstract 4
Acronyms 5
List of Figures 8
List of Listings 9
List of tables 9
Outline 10
1 Introduction 11
1.1 Particle Therapy 11
1.1.1 Medical Imaging Methods 12
1.2 The Bergen protonCT Project 13
1.3 Requirements 16
2 Related work 17
2.1 IPbus 17
2.2 Connected projects 20
2.2.1 The Control room 20
2.2.2 The Power Control Unit 20
3 Problem analysis 21
4 Design 22
4.1 MJCU – top-level 23
4.2 com_module 24
4.2.1 com_module_reg 25
4.2.2 com_module_fifo 29
4.2.3 com_module_usart 31
4.3 global_module 34
4.4 dummy_module 35
4.5 version_module and housekeeping_module 35
4.5.1 Version_module 35
4.5.2 Housekeeper_module 35
5 Tests and verification 36
5.1 Verification 36

5.1.1 UVVM 37
5.1.2 DO-file 39
5.2 Hardware testing 39
6 Discussion 43
7 Conclusion and Outlook 44
7.1 Conclusion 44
7.2 Outlook 44
8 Bibliography 45
Appendix A Development tools 46
Appendix B Address map for all modules inside the MJCU 47
A.1 Module addresses 47
A.2 Address map 48
Appendix C DATA FLOW 50
Appendix D Picture of an ALPIDE sensor 52
Appendix E Project plan 53

List of Figures

Figure 1 - Graphical illustration of how the dose is delivered to the tissue [3] 11
Figure 2 - Simplified CAD of the pCT instrument.. 13
Figure 3 – Bergen pCT project overview 14
Figure 4 - Simplified overview of the pCT power control system 14
Figure 5 – power control unit.. 15
Figure 6 – PCU connected to a power supply unit 15
Figure 7 - The Xilinx KCU105Evaluation Board [4]... 16
Figure 8 - ipbus implemented in a top-file 17
Figure 9 - ipbus_fabric_sel with modules connected.. 18
Figure 10 - powermonitor repository - folder structure inside each module 22
Figure 11 - Block diagram of MJCU as payload 23
Figure 12 - com_module block diagram... 24
Figure 13 - IPbus wait state slaves 25
Figure 14 – zero-wait state read 26
Figure 15 – zero-wait state write 27
Figure 16 - part one in a pointer-based FIFO illustration. 29
Figure 17 – part two in a pointer-based FIFO illustration. 29
Figure 18 - part three in a pointer-based FIFO illustration. 30
Figure 19 - part four in a pointer-based FIFO illustration. 30
Figure 20 - Custom USART protocol between the FPGA and PCUs.. 31
Figure 21 – The state machine in com_module_usart. 32
Figure 22 - Example of a wave diagram from a small testbench 36
Figure 23 - MJCU writing 0x00AA to PCU-address 0x01 (asynchronous)... 40
Figure 24 - MJCU reading 0x00AA from PCU-address 0x01 (asynchronous) 40
Figure 25 - Mean bit error using 115200 baud 42
Figure 26 - Mean bit error using 921600 baud 42
Figure 27 - Project plan 43
Figure 28 - Project plan for BO22EB-08... 53

List of Listings

Listing 1 - Connecting a slave to IPbus 19
Listing 2 - How ACK follow strobe 25
Listing 3 - IPbus slave - read process 26
Listing 4 - IPbus slave -write process 27
Listing 5 - IPbus slave - address map 28
Listing 6 - UVVM log report 37
Listing 7 - UVVM do-file 38

List of tables

Table 1 - IPbus signals between master and slave [5] 18
Table 2 - address map com_module 28
Table 3 - Address map global_module 34
Table 4 - Address map dummy_module 35
Table 5 - Bit-error test with varying baud rates, performing 1000 read operations 100 times. 41
Table 6 - Module addresses 47
Table 7 - Address map for MJCU 49

Outline

Chapter 1: Introduction. This chapter contains an introduction to the project that this thesis is a part of. Also included are the requirements for the project connected to this thesis.

Chapter 2: Related work. The related work chapter focuses on giving background information on what work was needed for this project to work. IPbus is explained and given an introduction. Both adjacent master theses are also presented.

Chapter 3: Problem analysis. This is the start of the project. Problem analysis is an analysis of the requirements and the earlier work, to conclude with an idea on how to solve the task.

Chapter 4: Design. Designing the VHDL solution was the biggest part of creating a working solution. In this chapter, a detailed walkthrough of all the different components in this solution is given. As well as some in-depth information on the workings of each component.

Chapter 5: Tests and verification. In this chapter, the focus lies on what tests were performed on and around the project. Including what type of tests were performed, and what tools were used to perform said tests. Thereby looking into what UVVM is and why it was used.

Chapters 6: Discussion, and **Chapter 7: Conclusion and Outlook.** These chapters evaluate the work done in this thesis. Outlook also gives a discussion on the future work needed.

1 Introduction

1.1 Particle Therapy

Radiotherapy is the traditional form of cancer treatment using irradiation. In radiotherapy, high-energy electromagnetic radiation is used to kill cancer cells. Radiotherapy is an important method for cancer treatment. It can be used alone, or in combination with other methods like surgery, chemotherapy, and/or immunotherapy. The goal of the treatment is to damage the cancer cells either directly or indirectly. Stopping them from reproducing or causing them to die.

Sending ionizing radiation through the body is effective for killing cancer cells, but it also damages healthy cells in the process. Healthy tissue has an advantage over tumorous tissue though. It heals faster. Treatment is therefore distributed over multiple sessions of irradiation. Each dose administered is weak enough for the healthy tissue to recover in between the treatments. The photon beams are sent through the body from several directions to further spread out the damage. A high level of precision is important to make sure that the healthy tissue receives a low amount of radiation. With higher precision comes the possibility of safely increasing the dosage. Resulting in fewer sessions to complete the treatment.

The biological structure varies from tumour to tumour. It can even vary within a single tumour. Oxygen is a mediator for indirect damage to DNA. Parts of a tumour, which have reduced access to oxygen, can be more resistant to radiation. Samples, MR- and PET pictures are used to provide information about the tumour's biology. This makes it possible to individualize the treatment of different tumours [3].

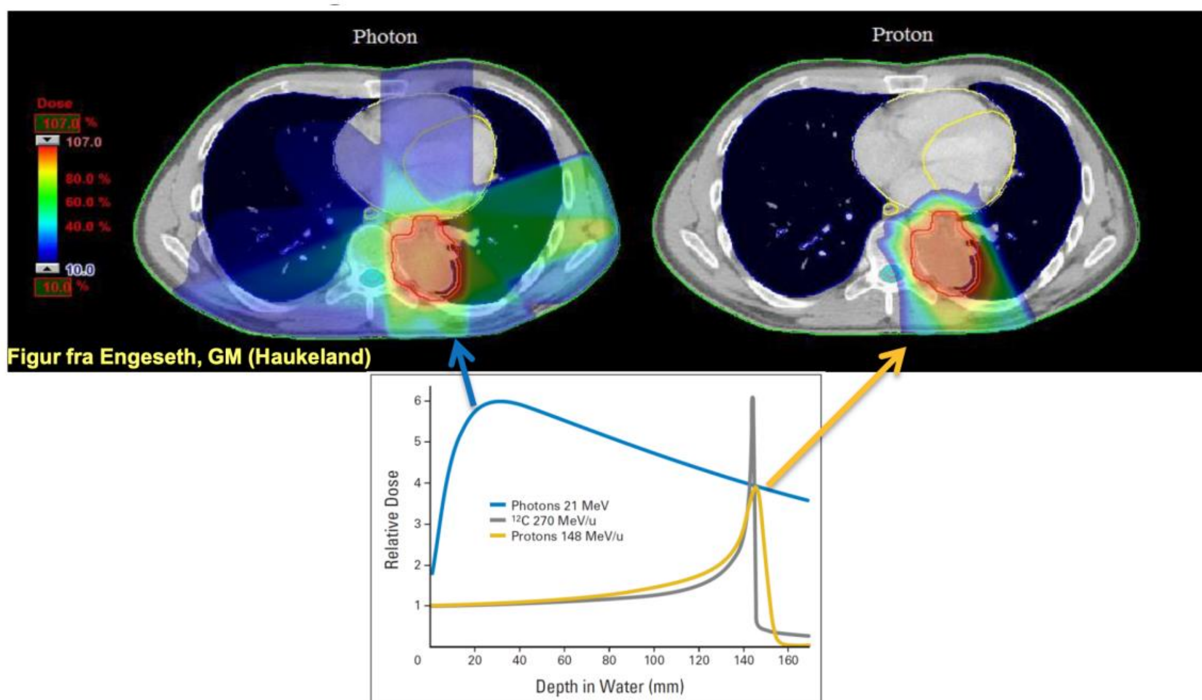


Figure 1 - Graphical illustration of how the dose is delivered to the tissue [3]

Particle therapy uses positively charged ions like protons or carbon ions instead of the traditional photons. This opens the possibility for treatment of delimited areas inside a patient, whilst damaging DNA more efficiently. Compared to radiotherapy, particle therapy results in fewer side effects such as organ damage or secondary cancers because it reduces the radiation to the surrounding tissue.

The two methods differ the most in how the energy is deposited in the tissue. Figure 1 shows how the energy is transferred using radio- and particle therapy.

The photons (blue) transfer a large amount of energy to the tissue a short distance after they penetrate the skin. Therefore, the tumour is radiated from different angles to achieve the correct amount of radiation. The yellow line illustrates how the protons deposit a major part of their energy directly into the tumour. This peak is called the Bragg peak. It can be adjusted to target where the tumour is located. The Bragg peak is not wide enough to cover the whole tumour. A modified beam is therefore created to get a broader range. This beam is called Spread-Out Bragg Peak (SOBP). The graph also shows the potential of using carbon-ions with an even more effective peak.

1.1.1 Medical Imaging Methods

Computed Tomography

A Computed Tomography (CT) scan combines multiple images taken from different angles of a body using beams of photons. The device is rotated around the body, and the body is moved horizontally to create a three-dimensional image.

Proton Computed Tomography

Proton Computed Tomography (pCT) uses protons instead of photons. The images are created by sending a beam of protons with enough energy to pass directly through the patient so that the Bragg peak ends up in the calorimeter. This reduces the amount of irradiated tissue. The protons will interact with electrons and/or the nuclei of an atom. These interactions will cause them to change paths. The path of the protons after exiting the body is then reconstructed from the data collected by the calorimeter.

It is possible to achieve higher accuracy when using pCT-scanning for dose planning compared to a CT scan. Currently, the proton treatment is estimated by the Houndsfield units from CT scanning. This estimation causes an error of 2-3% [3]. This implies that by using proton CT, less of the healthy tissue will be irradiated and the risk of late effects is reduced.

1.2 The Bergen protonCT Project

The University of Bergen is the leading part of the Bergen proton CT project. The project consists of multiple parts, as shown in Figure 3. The main part of this project is the calorimeter depicted in Figure 2. It is made out of 43 layers with 108 ALPIDE sensors on each layer. They are monolithic active pixel sensors (MAPS) created for the CERN ALICE experiment. These layers are responsible for collecting information about the proton's velocity and position.

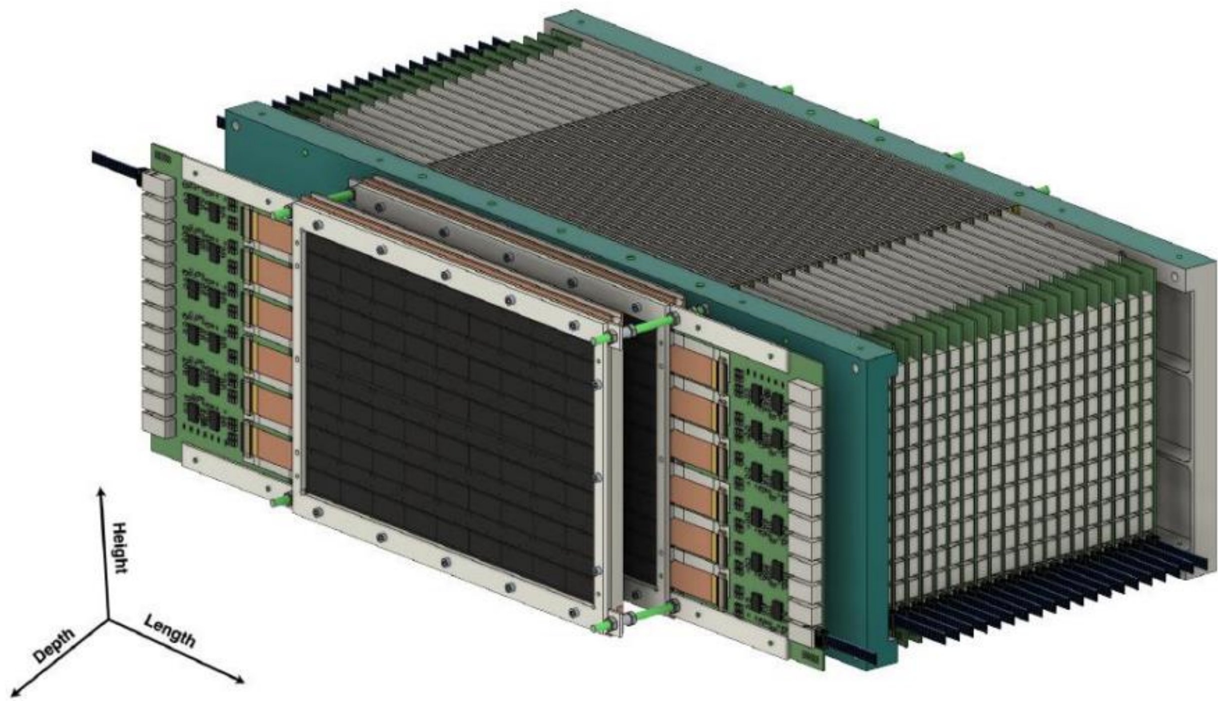


Figure 2 - Simplified CAD of the pCT instrument

Each of the layers is connected to individual transition cards and custom power control units (PCU) that deliver the power from the power supply units (PSU) to the layer. This thesis is aimed to make a system that could write and read information to and from the PCUs and forward it to a computer inside the control room. An FPGA board is being used to achieve this.

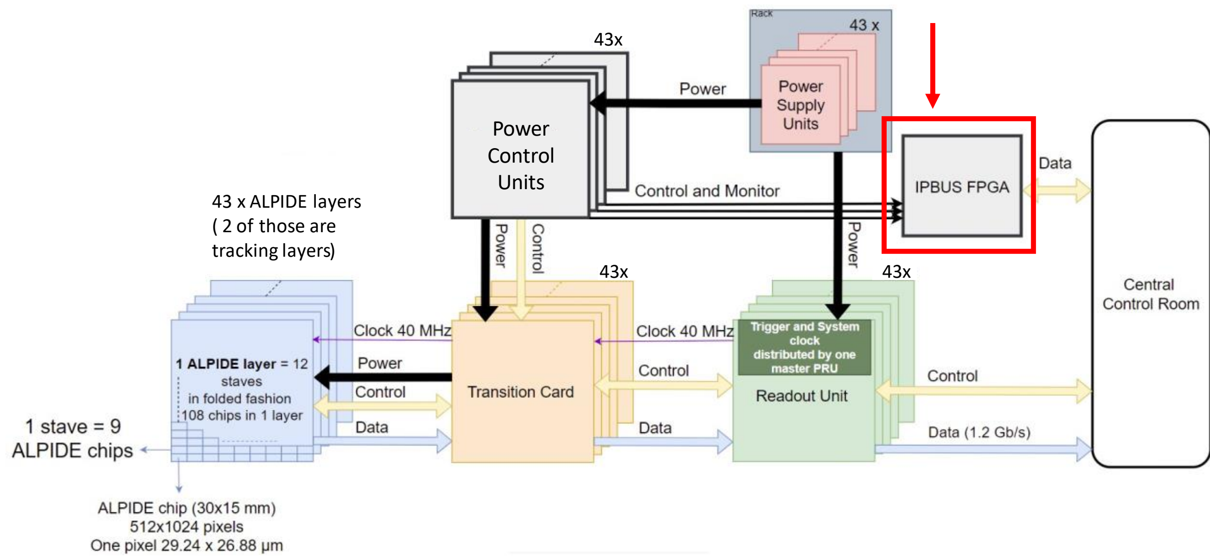


Figure 3 – Bergen pCT project overview

Figure 4 shows how the different components in the pCT power control system are connected to this thesis. The computer in the control room connects to the FPGA board with an ethernet cable using the IPbus protocol. The FPGA board is connected to 43 PCUs via ethernet cables, using a custom USART protocol for communication.

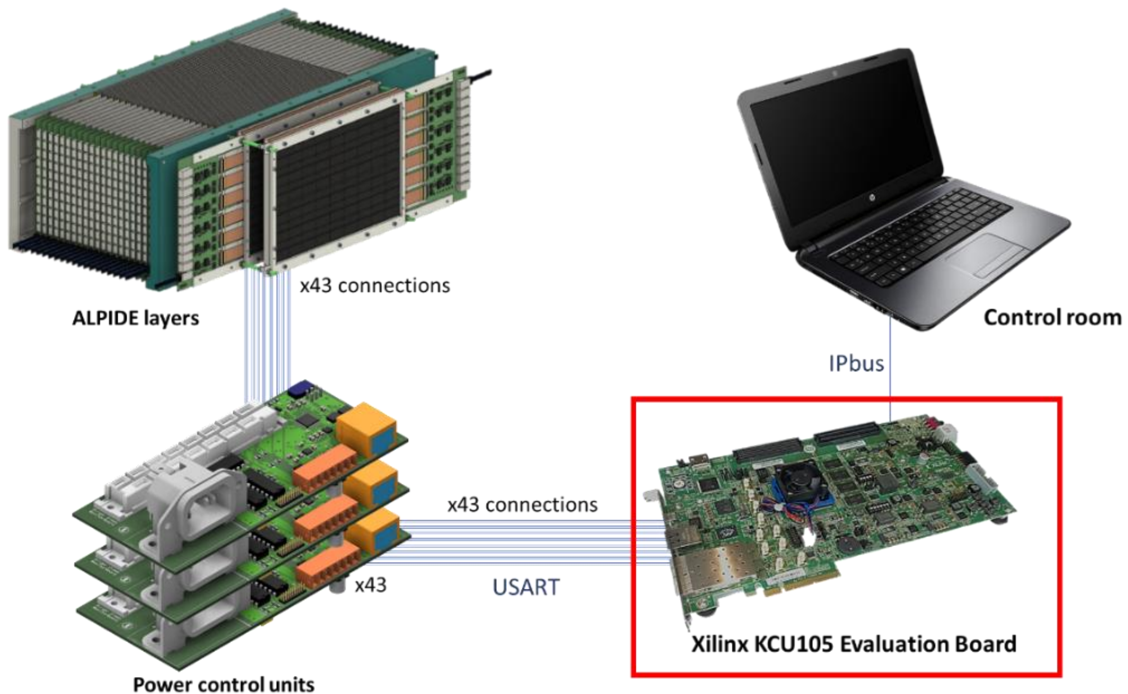


Figure 4 - Simplified overview of the pCT power control system

Figure 5 shows what in Figure 4 is labelled as “Power control units”. These are the custom-made circuit boards that transfer power from the PSUs (shown in Figure 6) to the ALPIDE layers. The connections are depicted in Figure 3. 43 of these boards will be connected to the FPGA board through an RJ45 rack.

The PCUs have full control over the power given to the ALPIDE layers by the PSUs. They can control the maximum voltage and current on its outputs and shut down the connected layer if a maximum temperature is reached. All the information about different values and thresholds is available to read and modify by the user through the FPGA.

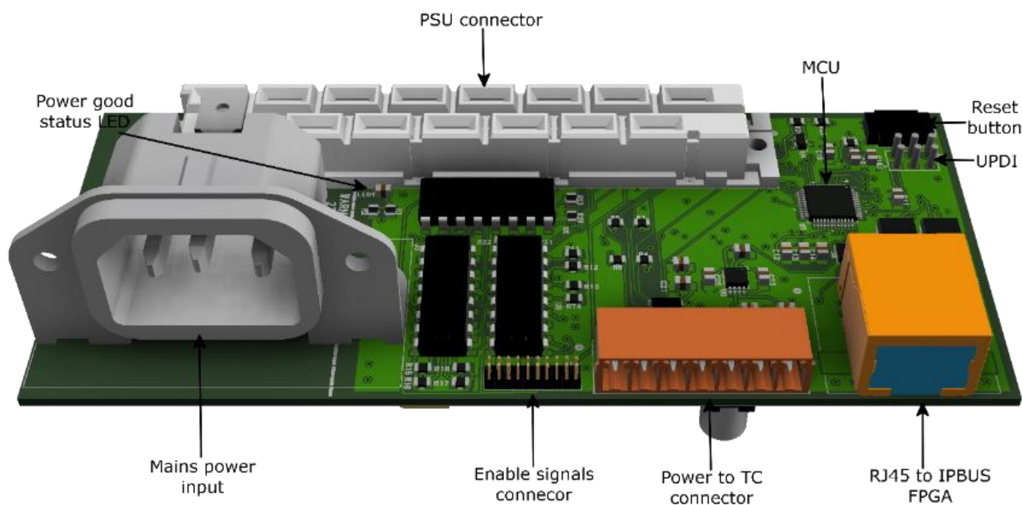


Figure 5 – power control unit

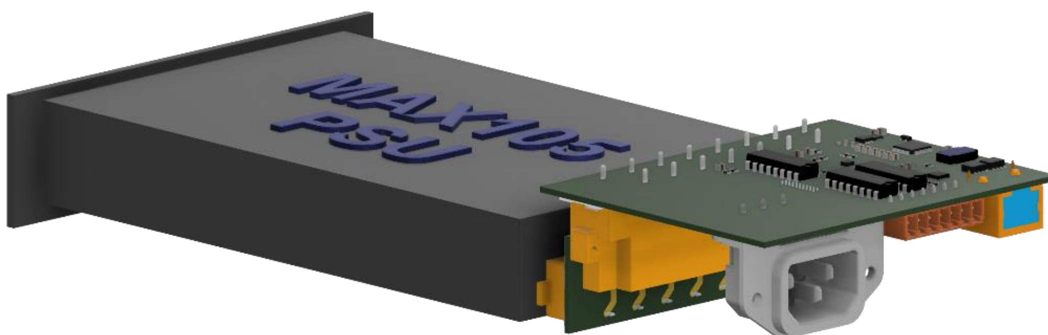


Figure 6 – PCU connected to a power supply unit

1.3 Requirements

The prime objective of this thesis is to make a stable communication link between a computer inside a control room (using IPbus by CERN) and 43 external devices (using a custom USART protocol). Designing it to be as stable and reliable as possible. Prioritizing stability over efficiency.

The requirements are as follows:

- FPGA design fit to a Xilinx KCU105 Evaluation Board containing
    - An IPbus gateway.
    - A bus connected to multiple slaves containing
        - 43 slaves for communication with external devices
        - A global module overarching all 43 communication slaves
        - A test module, for confirming functionality
        - Slaves managing onboard functions and a version GitHash
- Developed using a type of GIT version control system.
- Developing every component with testbenches using the UVVM library.
- Good documentation for every part of the system.

The Xilinx KCU105 Evaluation Board features the Kintex XCKU040-2FFVA1156E FPGA. It has all the I/O resources that would be necessary for this project. The required resources contain an RJ45 port for connecting to the control room, some GPIO for prototyping, and two FMC connectors (high/low pin-count) for connecting the PCUs.

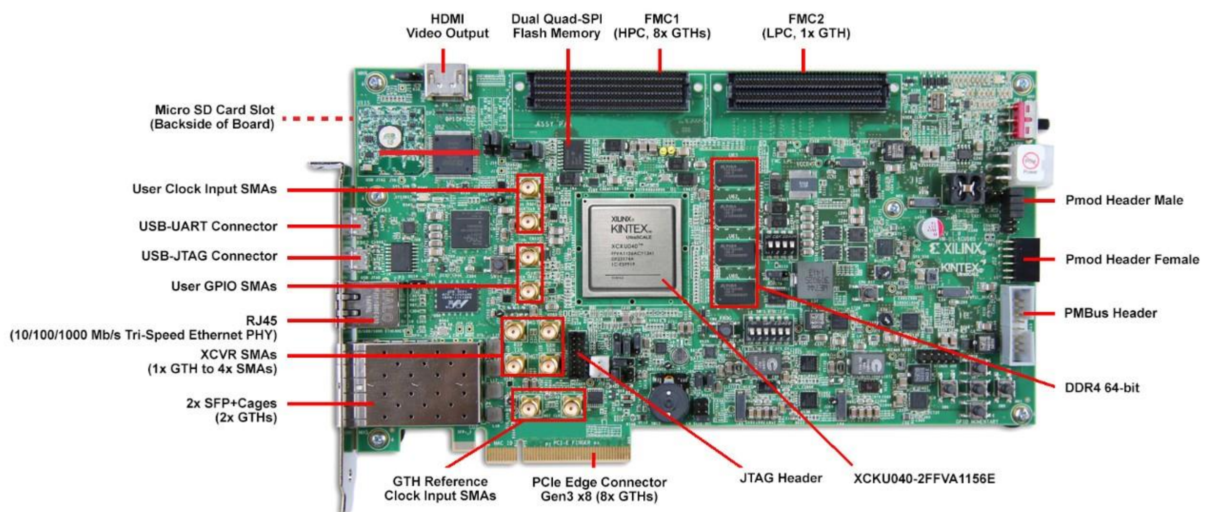


Figure 7 - The Xilinx KCU105Evaluation Board [4]

2 Related work

2.1 IPbus

IPbus is a packet-based control protocol made by CERN for communicating with address-aware hardware inside an FPGA [5]. They have made both the software and the firmware side that communicate with each other through 1Gbit ethernet. It does not support devices with auto-negotiation. The software side has a Control Hub that uses an IPbus reliability mechanism to correct for losses or duplications. The firmware side can be connected to multiple slaves made by a user, using high-performance and reliable connections.

The IPbus firmware module is an on-chip system. All the calculations and timings are being made and processed inside the FPGA chip. No external devices are needed. This makes it compatible with a wide range of FPGAs regardless of what type of sensors or circuitry surrounds the chip. It is designed to be a common module that can run by itself alongside a user's main code inside the same FPGA. This makes it easy to integrate into a project.

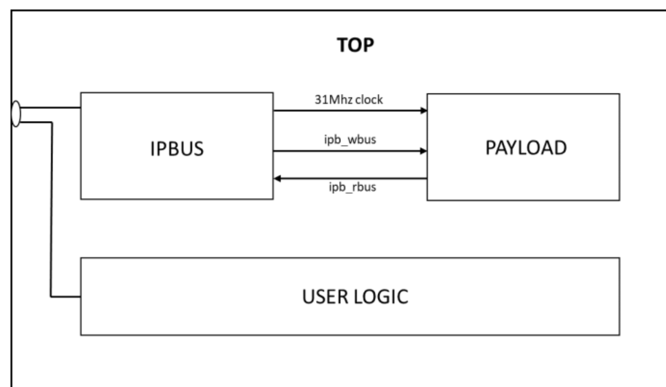


Figure 8 - ipbus implemented in a top-file

CERN has made several example designs for different development boards. This is currently made for Xilinx devices only, but the code is adaptable to fit onto Intel devices as well.

The firmware side of IPbus acts like a master within the FPGA and communicates with connected slaves located inside the PAYLOAD-block using a 31Mhz clock and seven signals shown in Table 1. These seven signals are located inside two types named *ipb_wbus* and *ipb_rbus*. *Ipb_wbus* has all the signals going from master to slave, and *ipb_rbus* got the ones going from slave to master. These signal types can be used to declare signals or ports like this:

```
IPBUS_IN    : in      ipb_wbus;
IPBUS_OUT   : out     ipb_rbus;
```

The reduction of signals makes it easier to follow the routing of the signals between the different modules and components.

Signal	Direction	Width	Description
ipb_addr	Master to slave	32	Bus address
ipb_wdata	Master to slave	32	Data to be written to a slave
ipb_write	Master to slave	1	Asserted for a write cycle, deserted for a read cycle
ipb_strobe	Master to slave	1	Qualifies address and data; assertion marks start of a cycle
ipb_rdata	Slave to master	32	Data read from a slave
ipb_ack	Slave to master	1	Acknowledge flag; assertion marks end of cycle
ipb_err	Slave to master	1	Bus error flag; assertion marks end of cycle

Table 1 - IPbus signals between master and slave [5]

The slaves themselves do not need to handle the address signal from IPbus. This is handled by a component inside PAYLOAD named ipb_fabric_sel shown in Figure 9. Ipb_fabric_sel is responsible for which slave gets to communicate with the IPbus. This makes it possible to make slaves that do not know what address to respond to. They are just waiting for a strobe signal from fabric_sel. As a result, all slaves can interpret the signals the same way. They just need to take care of any internal registers.

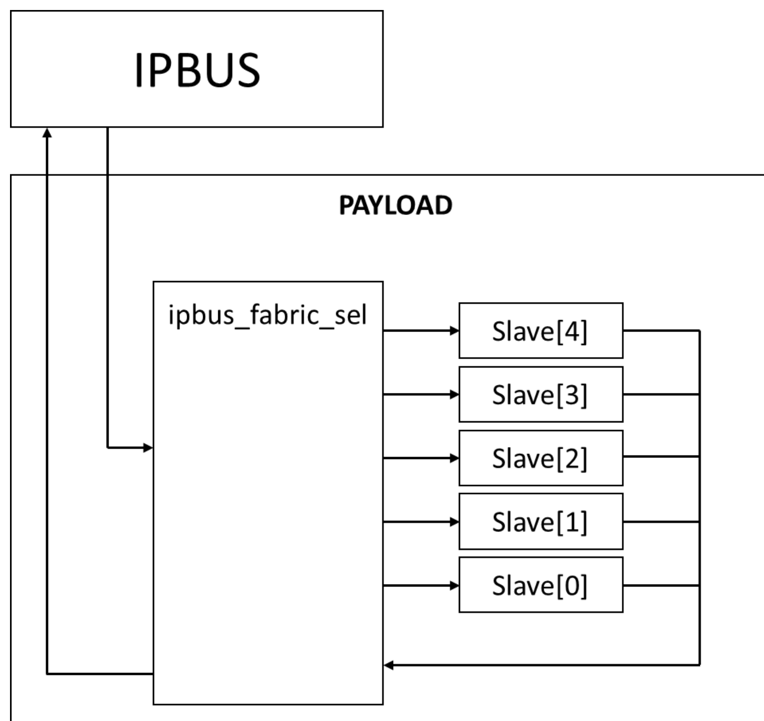


Figure 9 - ipbus_fabric_sel with modules connected

The implementation of fabric_sel also makes it easier to add more slaves to the system if needed. Using the port map of the dummy_module slave as an example:

```
-- Slave 67 : dummy_module
dummy_module : entity work.dummy_module
port map(
CLK         => ipb_clk,
RST         => ipb_rst,
IPBUS_IN    => ipbw(67),
IPBUS_OUT   => ipbr(67)
);
```

Listing 1 - Connecting a slave to IPbus

Dummy_module will answer to module address 67 because it is connected to in- and output number 67 on fabric_sel. If more slaves are introduced, they can be connected to pin 68, 69 and so on. They will then automatically answer to the corresponding module address.

2.2 Connected projects

In the previous chapter, a mention was given to the wider project that stretches beyond the borders of this assignment. Where in Figure 3 and Figure 4 a complete overview and a simplified view of the entirety of the project was shown. In relevance to the FPGA, the connecting parts of the project can be split into two parts. The “control room” and the “power control units”. These two parts have each their own master student working on them.

2.2.1 The Control room

The software for the control room is designed by Håvard Birkenes [6]. The control room as it is referred to is a computer connected to all information channels of the project. The section Håvard is responsible for contains all that is connected to the power control system. It is responsible for handling the control, configuration, and monitoring of the power system. The configuration data are stored in a MongoDB database, and the monitoring data are stored in a time-based database (inFluxDB). Grafana is used to visualize the monitored values.

Python is used as the design language in this part, and this enables the possibility to communicate with the API directly using script-based functions. This is extremely valuable in the test and development phase.

2.2.2 The Power Control Unit

The PCU and the concept of the distributed power system is designed by Birger Olsen [7], Originally, the plan was that all layers should be powered by one large power supply, and there should be one central power control unit for all layers. However, the solution proposed by [7] proved cheaper, more elegant and easier to implement.

The PCU, which sits directly on the power supply, includes filters to reduce the power supply noise, as well as a microcontroller that enables the power to the sensor layers and monitors the currents, voltages and temperatures. Additionally, it contains programmable thresholds. The power is cut if the monitored values exceed these thresholds. This gives a short response time which is needed to avoid damage to the sensor layers and lowers the speed requirement for the rest of the power control system. Making it possible to prioritize reliability before speed. It is important to stress that since the sensor layers measure ionizing radiation, errors can be expected that for instance generates high current consumption due to single event effects in the sensors.

3 Problem analysis

Writing a design to an FPGA requires a VHDL design tool. In this case, since the selected FPGA is from Xilinx, we were given access to a program called Vivado. This was made available to us by the University in Bergen.

The Xilinx KCU105 Evaluation Board featuring the Kintex UltraScale XCKU040-2FFVA1156E FPGA was chosen on the premise that it featured everything that was needed in terms of IO-resources and FPGA. It was also the most convenient choice since other options would have to include a custom PCB design. The need for prototypes during development and the low quantity production would have led it to be more expensive than buying a finished product. This would also require a design team that would increase the costs even further.

With the requirements in mind, there are still multiple options on how to structure the VHDL design. The FPGA's task is to transfer data between a control room and the multiple power control units. One way of designing a system like this would be to make multiple components that take care of different requirements and then connecting them together. This way of design results in a tidy, and easy to test and troubleshoot. Each of the components can be verified in a controlled testbench. Making it easy to add more components in the future if needed. A goal of the design is to make it as efficient as possible, to make sure it's not the bottleneck of the system. One of the ways to accomplish this is to make sure the system can communicate with the PCUs in parallel. Since the solution is an FPGA design, it is possible to have the communication between one PCU take the same amount of time as communicating with all 43 of them at the same time. It is important to state that the system does not have a strict timing requirement. It is however important that the configuration time does not feel slow for the user, and that the monitoring is frequent enough. In this case, reliability is more important than speed. One of the ways to make it more reliable is by writing the data that has been written to a given address to the control room using what is called a handshake. The PCUs already have this as a built-in function in their custom USART protocol. It confirms which address was reached by sending a copy of the address back as soon as it is received. This confirmed address is then put together with the data received inside the FPGA to formulate a handshake to the control room as well.

Since this bachelor thesis is a part of a bigger system, it was important to make the project structured and understandable for other people that may modify or reuse it after this bachelor thesis has ended. UiB were already using a custom version of GitHub namely git.app.uib as a Git version control system. Which we used as well in the development of this project.

The project has multiple modules with independent testbenches and files, so it was important to keep everything organized and well structured. To be consistent with the naming of folders and components was also a high priority.

4 Design

The main essence of this project was to make an FPGA solution that would take in data from an already existing bus design (IPbus) and convert said data into a serial data transfer toward the PCUs, and the other way around. The protocol for the PCUs was made a few weeks after this project began but it was still prone to change.

It was made clear that the goal should not be to complete every requirement given, but to make sure everything that was constructed was working as intended. A lot of time went into the creation of testbenches because of this.

To make this project well-structured and easy to understand, each of the modules was created separately. This made it possible to connect them to different testbenches to show how the individual parts of the system worked. To make sure the git repository [2] was well structured, all the folders for the different modules got the same layout. Every module has its own folder including the subfolders components, scripts, source files (src) and testbench (tb). This makes navigating to the desired files easier, and each testbench can be compiled the same way. The reason structure is so important is because someone else should be able to do further work and development on it in the future if it is needed.

Name	Last commit	Last update
..		
components	FIFO behaviour	1 week ago
scripts	Update MJCU_tb log.txt	1 week ago
src	Update MJCU_pkg.vhd	4 days ago
tb	FIFO behaviour	1 week ago

Figure 10 - powermonitor repository - folder structure inside each module

4.1 MJCU – top-level

MJCU is an acronym for Martin and Jakob's Communication Unit. Figure 11 shows a simplified block diagram of how the modules are connected to each other, to IPbus and to the power control units. Since the modules have internal registers, a decision was made that ipb_fabric_sel should use *ipb_addr(15 downto 8)* to determine which module to select. Leaving *Ipb_addr(7 downto 0)* available for selecting internal registers inside the modules. This should give a sufficient amount of both potential slaves and slave addresses. The modules in this design are com_module, global_module and dummy_module.

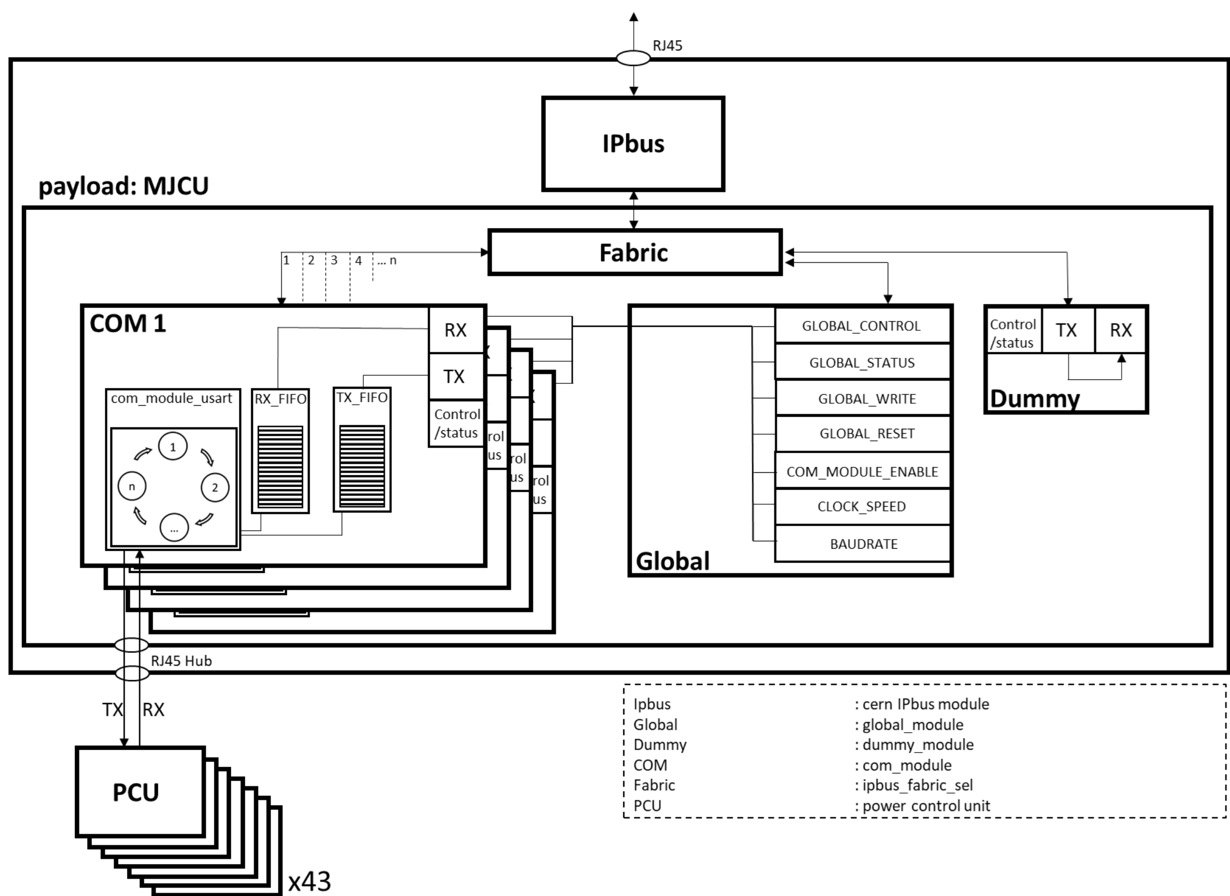


Figure 11 - Block diagram of MJCU as payload

4.2 com_module

Com_module is short for communication module. It is the largest of the modules in this design and is responsible for communicating with the power control units. This module is meant to be duplicated as many times as the amount of power control units connected to the system. One com_module will be connected to one PCU.

Com_module consists of four components. Com_module_reg, two com_module_fifos and com_module_usart. Com_module_reg handles the communication with IPbus. It can read data from one com_module_fifo and write to the other. FIFO stands for "First In First Out", and is used to temporary queue up data. Every com_module has two FIFOs. One for the data from com_module_reg to com_module_usart (TX_FIFO), and one for the data from com_module_usart to com_module_reg (RX_FIFO). Com_module_usart communicates with a power control unit using a custom 8-bit USART protocol. This protocol sends data in bulks of 8 bits, one bit at a time with a start bit for every transaction. The bits are being sent with a given baud rate. Because the baud rate (default set to 115200 bits/s) is much slower than the clock speed the modules are using (31 MHz), it is necessary to use FIFOs to prevent the loss of data.

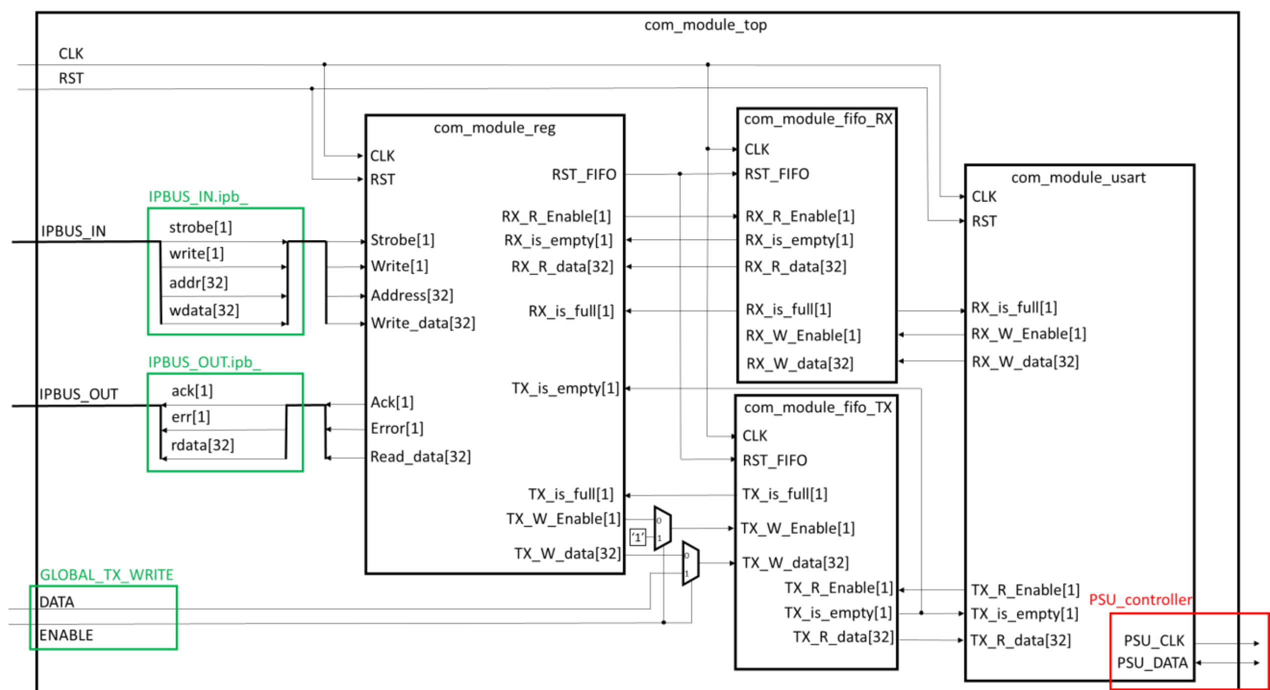


Figure 12 - com_module block diagram

4.2.1 com_module_reg

Com_module_reg is the component that handles the signals between the IPbus interface and any given com_module. The goal was to make com_module a “zero wait state”-slave, meaning that it can receive or transmit data on every clock cycle. Figure 13 shows two write transactions between IPbus and two slaves. The slave written to in cycle 0 is what is referred to as a “one wait state”-slave. It uses one clock cycle before it responds with *ipb_ack.* IPbus is made in a way that the slave can have multiple wait states. IPbus will wait for ack before proceeding. Cycle 1 shows a slave with zero wait states. These types of slaves are the most efficient ones because they never require IPbus to wait for a response.

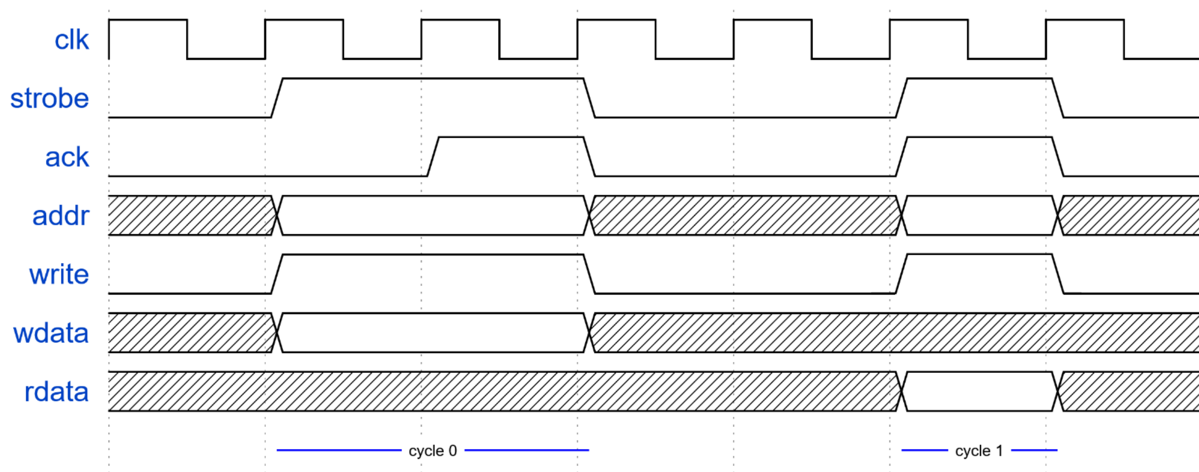


Figure 13 - IPbus wait state slaves

To make “zero wait state”-slaves, it was necessary to have the *ipb_ack* mirror the *ipb_strobe*. In the first iteration, *ack* was given *strobe*’s value directly. But because of the way ipbus_fabric_sel (Figure 9) handles the *ack*-signal, it had to be given its value via a MUX like this:

```
IPBUS_OUT.ipb_ack        <= '1' when IPBUS_IN.ipb_strobe = '1' else '0';
```

Listing 2 - How ACK follow strobe

The first plan was to make one process for both read- and write operations. This would cause the slave to require one wait state on every read operation because they both would be following the clock. Therefore, it has separate read and write processes.

IPbus read transaction

The signal diagram in Figure 14 shows a read transaction with a com_module_reg. The communication is made possible by reacting to every signal coming from IPbus. In a read transaction between IPbus and a zero-wait state slave, IPbus will change the signals going to the slave on one clock cycle. The slave can then interpret these signals and change *rdata'*s value accordingly. IPbus will then read from *rdata* on the next clock cycle.

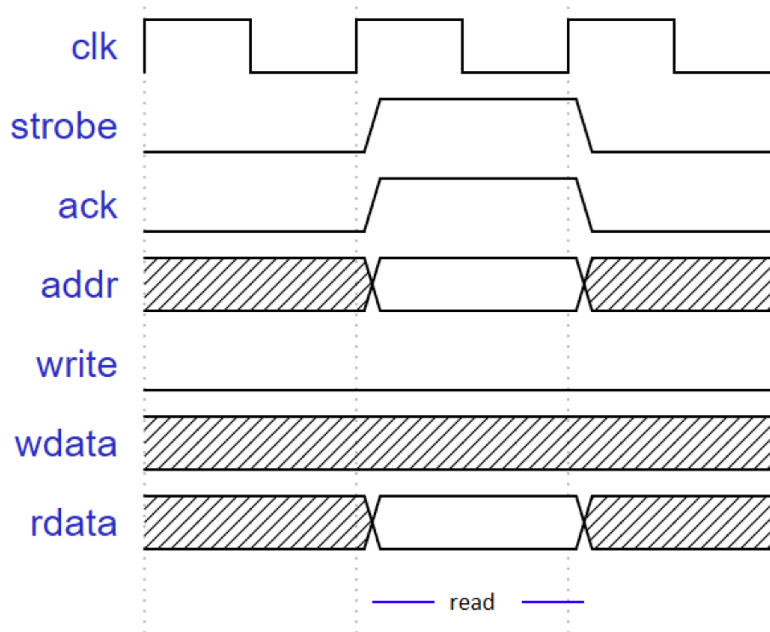


Figure 14 – zero-wait state read

The p_read process monitors every signal inside IPBUS_IN and reacts if any of them change. If *strobe* is high and *write* is low, *DATA* is written to *rdata* for the IPbus to read.

```
p_read : process(IPBUS_IN) -- Module writes data to IPBUS
begin
IPBUS_OUT.ipb_rdata <= (others => '0');
  if IPBUS_IN.ipb_strobe = '1' and IPBUS_IN.ipb_write = '0' then

      IPBUS_OUT.ipb_rdata     <= DATA;

  end if; -- STROBE & READ & MODULE_ADDRESS
end process p_read;
```

Listing 3 - IPbus slave - read process

IPbus write transaction

The timing in the write transaction in Figure 15 is synchronized with the clock. The slave checks if *strobe* and *write* are high on every rising edge of the clock. If so, it reads the data from *wdata*.

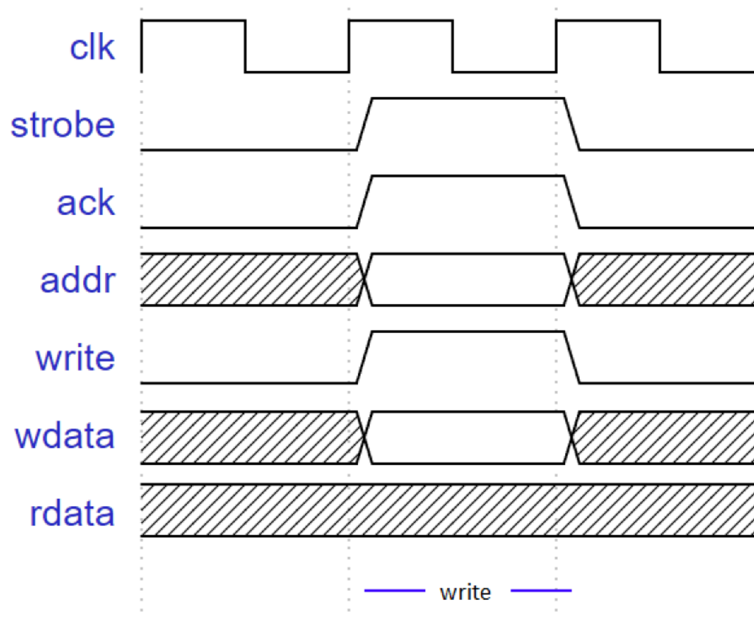


Figure 15 – zero-wait state write

```
p_write : process(CLK) -- Module reads data from IPBUS
begin
  if rising_edge(CLK) then
    if RST = '1' then
      else
        if IPBUS_IN.ipb_strobe = '1' and IPBUS_IN.ipb_write = '1' then

          DATA    <=     IPBUS_IN.ipb_wdata;

        end if; -- STROBE & WRITE & MODULE_ADDRESS
      end if; -- RST
  end if; -- CLK
end process p_write;
```

Listing 4 - IPbus slave -write process

The process p_write also includes the reset signal to make sure this only happens on a rising edge of the clock.

Address map

As already mentioned, *ipb_addr(7 downto 0)* from IPbus is available for selecting internal registers inside the modules. Com_module_reg takes care of this selection. Table 1 shows the address map of a com_module. The first two registers are reserved as control and status. Nothing is implemented here, as there was no need to do so. This was just a habit from earlier designs and makes it easier to implement more features later if needed. The data written to register 3 is sent to com_module_usart through one of the FIFOs and further down towards the power control units. Register 4 holds the data received by the power control units. Register 5 is a trigger register that resets both FIFOs. It does not matter if the data sent to register 5 is all zeros or all ones, everything is considered a reset if the register is reached.

Name	Register address	Width	Access	Default Value	Description
COM_MODULE (module address 0x01 – 0x40)					
CONTROL	0x01	32	W	0x00	Not in use
STATUS	0x02	32	R	0x00	[31..4] Not in use [3] TX is full (1 = full) [2] TX is empty (1 = empty) [1] RX is full (1 = full) [0] RX is empty (1 = empty)
WRITE	0x03	32	W	0x00	Writes data to the corresponding PCU. [31] R/$\overline{W}$ bit towards PCU [30..24] Not in use [23..16] PCU reg address [12..0] Data to PCU
READ	0x04	32	R	0x00	Reads a 32-bit vector from the corresponding PCU. [31] R/$\overline{W}$ bit towards PCU [30..24] Not in use [23..16] PCU reg address [12..0] Data from PCU
RESET_FIFO	0x05	32	W	0x00	Resets both TX and RX FIFO to clear them.

Table 2 - address map com_module

One of the goals with com_module_reg was to make the code behind the address map easy to understand and modify for future use. That's why both *p_read* and *p_write* have a case statement, shown underneath, to interpret *ipb_addr(7 downto 0)*. To add more registers, simply add one or more lines with *"when X"..." => "* in either or both *p_read* or *p_write* depending on if the register should be read-only, write-only or read/write.

```
case IPBUS_IN.ipb_addr(7 downto 0) is
      when c_CONTROL =>
      when c_STATUS  =>
      when c_WRITE   =>
      when c_READ    =>
      when c_RESET   => -- RESET FIFO
      when others    =>
end case; -- REGISTER
```

Listing 5 - IPbus slave - address map

4.2.2 com_module_fifo

The FIFO is an acronym for "First In First Out". The main and only purpose of a FIFO is to act as a temporary backed up data storage. We decided to use a pointer-based design. Using two pointers that are moving through a list of saved messages as they are written or read. Where one of them is responsible to read and the other one to write to their current location. An example of this is illustrated in Figure 16 and Figure 17. In Figure 16 there have been eight messages sent into the FIFO, two of these messages have then been read. That means that there are now six messages in the FIFO, messages three through eight. In Figure 17, another twelve messages have been sent into the FIFO, and now a total of nine messages have been read. At this point, messages ten through twenty are still in the FIFO, and one through nine are out. This shows how the "pointer in" has moved through the whole list and moved back to the start, but it isn't a problem because enough messages have already been read. The place where messages five through nine were stored is now "free space".

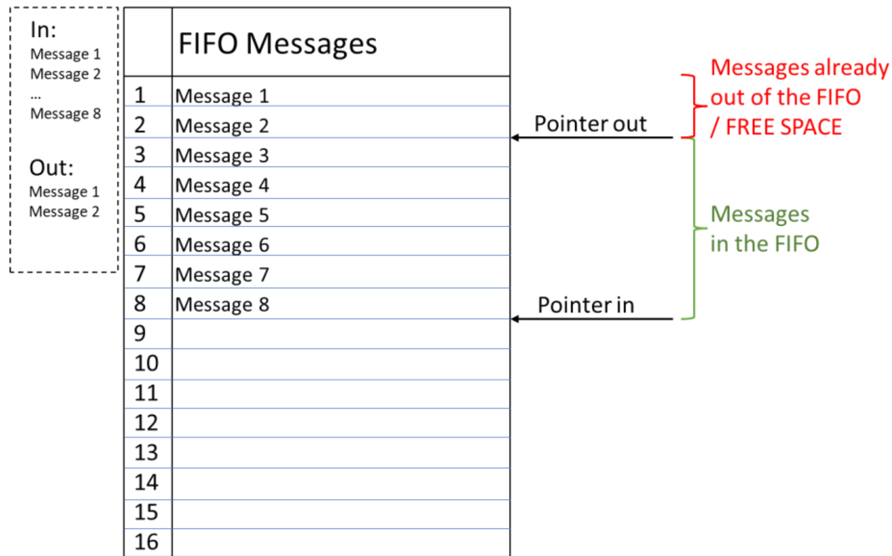


Figure 16 - part one in a pointer-based FIFO illustration.

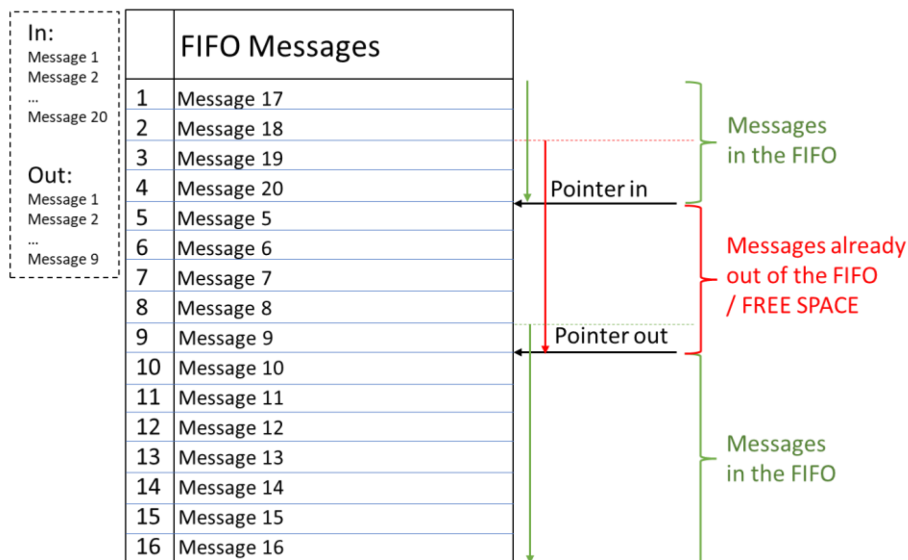


Figure 17 – part two in a pointer-based FIFO illustration.

There are some important corner cases to take into consideration when working with a FIFO. First and foremost, you will encounter one of these if you only write messages into the FIFO, but never read any out. This will end up in the FIFO being full. In this case, this FIFO will not accept any more messages, and any message that is still trying to get in will just be forgotten. However, when this happens the signal FIFO_IS_FULL, which is an output, will be set to 1. This corner case is illustrated in Figure 18.

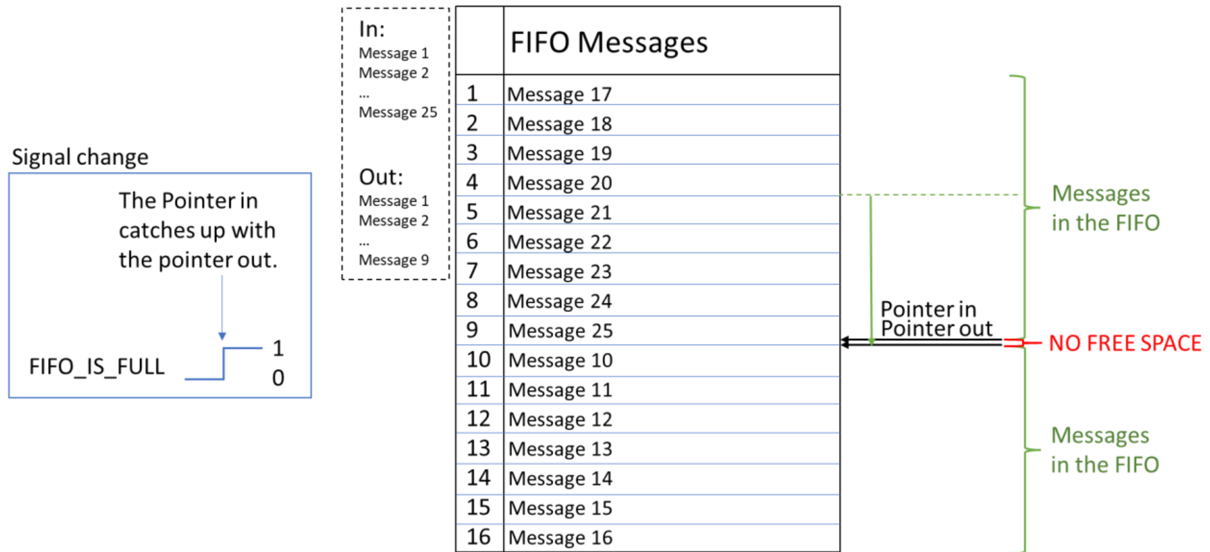


Figure 18 - part three in a pointer-based FIFO illustration.

Another corner case to take into consideration is what happens if you only try to read from the FIFO. At some point, there will be no more messages inside, and the FIFO will be empty. In this case, the FIFO will return only 0. As well as when full, there is an out signal "FIFO_IS_EMPTY" that is set to 1 when this corner case is hit. Illustration of this case in Figure 19.

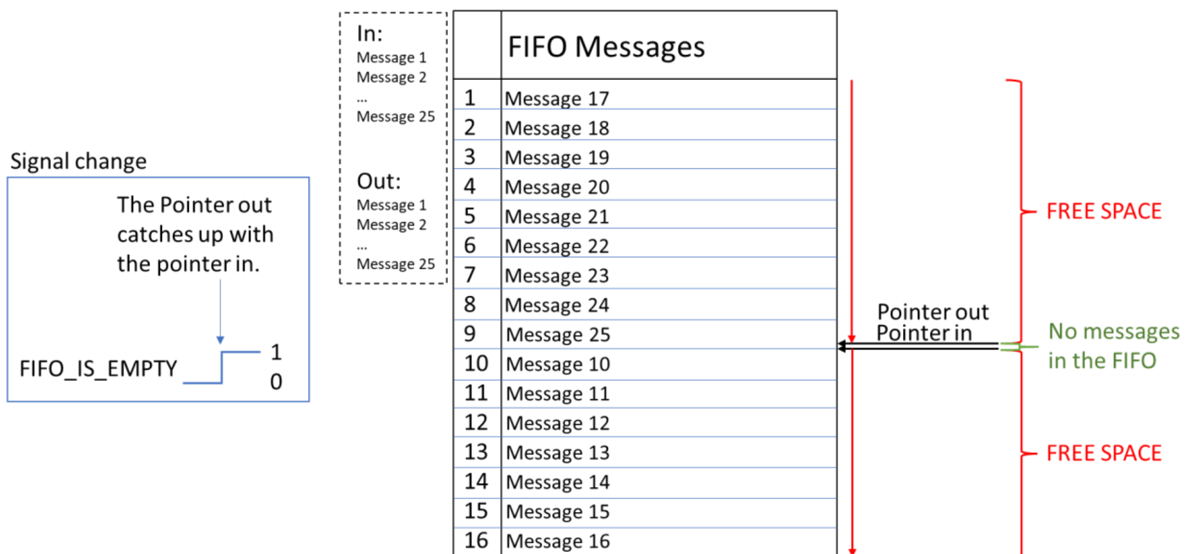


Figure 19 - part four in a pointer-based FIFO illustration.

4.2.3 com_module_usart

The main part of the com_module is the com_module_usart. This part is responsible for the communication between the MJCU and the PCUs. When it came to choosing how to design this component, the choice was very clear. In earlier subjects in our study, we have had multiple experiences with state machines. So, it felt natural to choose a state-machine solution for the design.

The TX_FIFO sends out one 32-bit message at a time every time the com_module_usart tells it to do so. The FIFO has two additional information outputs, “FIFO_IS_EMPTY” and “FIFO_IS_FULL”. The component continuously tells the FIFO to send out a new message as long as “FIFO_IS_EMPTY” is false. This is to make the component as time efficient as possible.

In Figure 20 a signal diagram is showing the protocol that is used for communicating. Starting every transaction with an initializing bit where high goes to low. A 7-bit address is sent followed by a R/W bit. Then for every read or write transaction a handshake is sent. The handshake is just the slave repeating what it has just received. Depending on if it is a read or a write, either master or slave then proceeds to send its data.

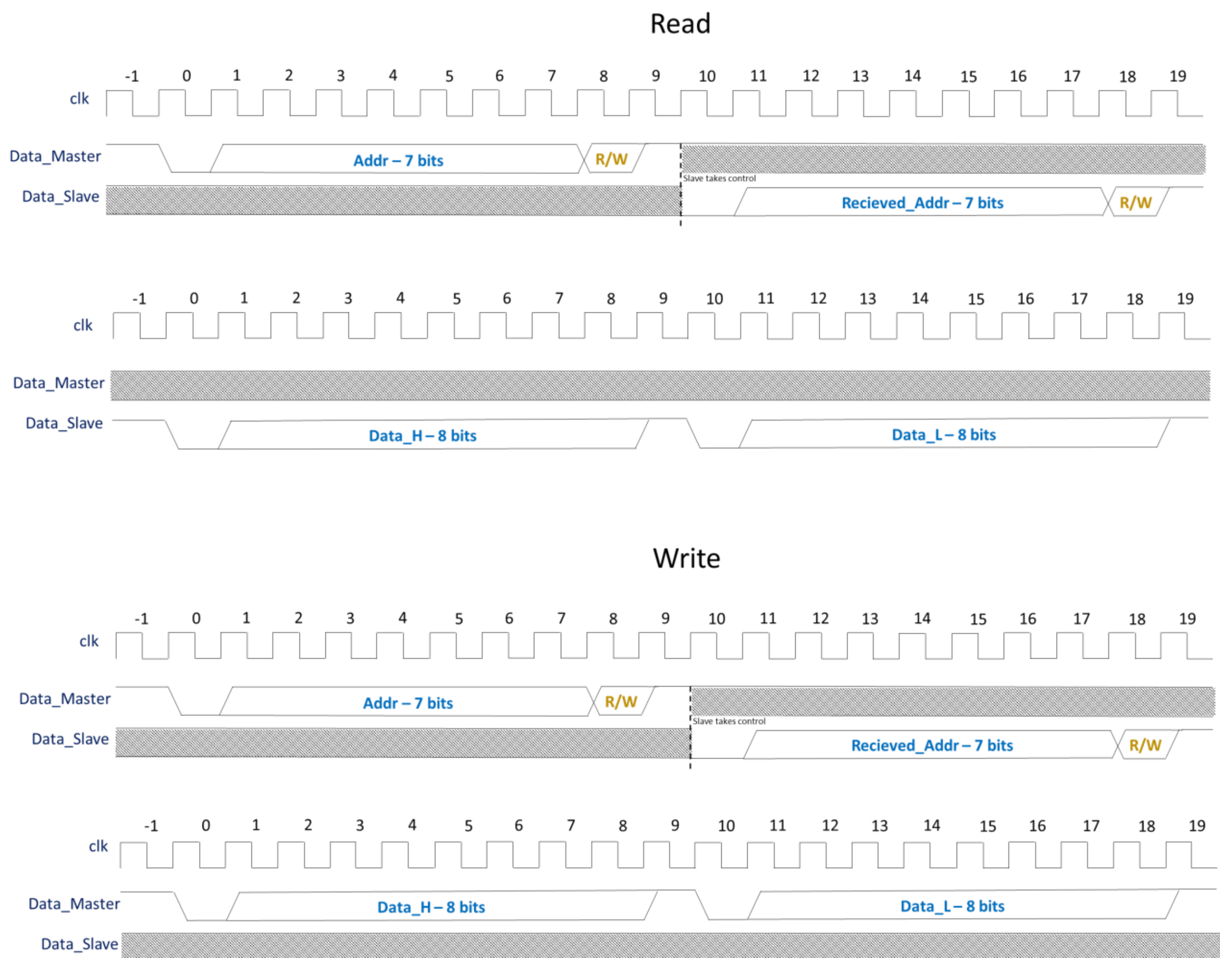


Figure 20 - Custom USART protocol between the FPGA and PCUs

The state machine

The state machine contains the following states: IDLE, NEW_ADDRESS, ADDRESSING_PART1, ADDRESSING_PART2, RUNNING, FINISHING.

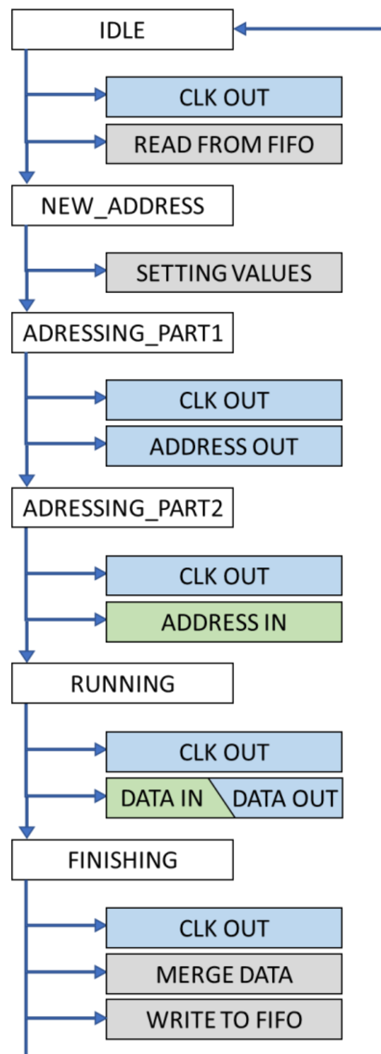


Figure 21 – The state machine in com_module_usart.

The component will be in IDLE until FIFO_IS_EMPTY goes from 1 to 0. Then it moves into NEW_ADDRESS to set up all the different signals and reset the counters. After this state was created, the next part was to figure out how to do the transaction itself. There is one master (com_module_usart) and one slave (PCU). The serial transaction is split into 3 parts.

The first part is the master writing to the slave which one of the internal registers of the PCU it wants to read or write to. Then in part two, the slave responds with the same read/write-bit and register address that it received from the master. This is called a handshake. With a handshake, the master can be certain that the correct address has been reached, and that the slave is active and responding correctly.

When the handshake is received, the next part depends on if the message is a “read” or a “write”. In both cases, 16 bits split into two chunks of 8 bits will be transferred between master and slave. When it is a “read” that is being executed, the slave will know, because of the read/write-bit, to write its data to the bus, and the master will read it. When a “write” is being executed, the master sends data to the bus, and the slave reads.

This sequence contains the states ADDRESSING_PART1, ADDRESSING_PART2, and RUNNING.

ADDRESSING_PART1 takes care of sending the read/write-bit and the register address.

ADDRESSING_PART2 reads the read/write-bit and the address that the slave responds with.

RUNNING includes both the “read” and “write” sequences. *Read* reads 16 bits from the slave and *write* writes 16 bits to the slave. The read/write-bit that was sent during the ADDRESSING_PART1 decides which one of the two sequences to run. Either way, the whole RUNNING state must get through the same number of bits. This means that the only thing that changes is whether to write to the slave or to read from the slave.

Then there is one more state, FINISHING. This state serves the purpose of delivering the data package to the RX_FIFO and telling the TX_FIFO that it is ready for the next transaction.

Then it switches back to IDLE.

While all this is happening, a 32-bit vector is constructed in the same format as the one read from the TX_FIFO. This data package includes if it was a read or write that was executed, what register it spoke with, and the data received/sent.

There is one more important part. Because this is USART, it needs an additional clock signal to synchronize the data transfer. There is a separate process that takes care of the USART clock signal. This signal is generated at a given baud rate that is controlled by the global_module. The main process that handles the communication follows this signal when writing or reading.

How the data is read

In all parts of the state machine where data is being read or written, it is only one bit at a time. This means that the serial data that goes in or out needs to be converted to a vector at some point. This is solved by using a counter that iterates through a vector and writes one bit at a time into the vector. It would have been more space-efficient to use a shift register rather than a pointer-based design. A shift register uses as many flip-flops as there are bits in the vector. This is not a problem for this project because the FPGA is as big as it is. The pointer-based design gave options for easy modification while developing. Coincidentally this ended up being very useful. The reason is that there was some confusion regarding if the most significant bit should come first or the least significant bit. In the end, the data needed to be flipped. All data transactions in the USART are LSB to MSB as of the current version.

4.3 global_module

The global_module is just a modified com_module_reg with a couple more outputs towards the com_modules. The first two registers are reserved to control and status. Nothing is implemented here, as there was no need to do so. This was just a habit from earlier designs and makes it easier to add more features in the future if needed. The data written to register 3 is sent to every com_module in the system. This makes use of a big advantage the FPGA has over a standard computer or an MCU. Since all com_modules get the same message, they can begin communicating with all power control units at the same time. In other words, communicating with one PCU takes the same amount of time as communicating with all of them. Register 4 does not do anything because this module never gets any of the data from the PCUs. Writing to register 5 will reset all other modules. This includes resetting the FIFOs inside every com_module. Register 6 and 7 are used to control the baud rate between the com_modules and PCUs. Write these values in binary, and they will be converted internally.

Name	Register address	Width	Access	Default Value	Description
GLOBAL_MODULE (module address 0x00)					
GLOBAL_CONTROL	0x01	32	W	0x00	Not in use
GLOBAL_STATUS	0x02	32	R	0x00	Not in use
GLOBAL_WRITE	0x03	32	W	0x00	Writes data to every com_module in the system. [31] $R/\overline{W}$ bit towards PCU [30..24] Not in use [23..16] PCU reg address [12..0] Data to PCU
GLOBAL_RESET	0x05	32	W	0x00	Resets all modules except the global_module
COM_ENABLE_0	0x06	32	R/W	0xFFFFFFFF	These 32 bits enable com_modules[1→32] '1' = ENABLED Bit[n] controls com _module_[1+n] This function is made in the case that modules are not connected to a PCU.
COM_ENABLE_1	0x07	32	R/W	0xFFFFFFFF	These 32 bits enable com_modules[33→64] '1' = ENABLED Bit[n] controls com _module_[33+n] This function is made in the case that modules are not connected to a PCU.
SYSTEM CLOCK SPEED	0x08	32	R/W	0x1D905C0	The speed of the clock used by the com_modules. This is used to calculate the baud rate towards the PCUs. Default: 31Mhz (clock from IPbus)
PSU BAUD RATE	0x09	32	R/W	0x1C200	The baud rate to use towards the PCUs

Table 3 - Address map global_module

4.4 dummy_module

Dummy_module is the simplest module, and it was also the first one to be completed. After the creation of com_module_reg, one signal was added to store a 32-bit vector. The thought behind having a dummy_module is to make sure the communication between IPbus SW and an IPbus HW-slave works properly. Write something to register 3 and it can be read back from register 4.

Name	Register address	Width	Access	Type	Default Value	Description
DUMMY_MODULE (module address 0x43)						
WRITE	0x03	32	W	STAT	0x00	Write anything to store a new value
READ	0x04	32	R	STAT	0xDEADBEEF	Read back the stored value

Table 4 - Address map dummy_module

4.5 version_module and housekeeping_module

4.5.1 Version_module

The version_module is meant to include the iteration of code being used on the FPGA and the githash connected to it. The githash is meant to be implemented automatically when compiling the project in Vivado.

4.5.2 Housekeeper_module

The KCU105-board has some internal sensors. The housekeeper_module is meant to get information from these sensors.

5 Tests and verification

Testing and verification are important parts of every development process. It is important to have data to back up the claims about how the part performs. Under development, testing is also used to confirm correct behaviour along the way. The testing performed while developing are often virtual testing.

5.1 Verification

Testbenches in VHDL are a great way to make sure the code is working as intended. Testbenches provide the possibility to simulate every part of a system with user-defined stimuli in a controlled and stable environment. This is a repeatable and reliable way of testing because it is not affected by external interference since everything is happening in SW. After the simulation is complete, a wave diagram is used to show the results. This diagram shows the state of every desired signal throughout the simulation. This can make it easier to debug any errors because it is possible to see where they happened. A wave diagram is also a good way of explaining what the code does. A diagram is often more intuitive to understand than straight-up code.

MJCU has a master testbench including all of the modules where every com_module is connected to a dummy version of the PCUs. This testbench covers every task that should ever occur in the system it is built for. All of the modules in MJCU also have their own testbenches to make it possible to have a closer look at the modules individually.

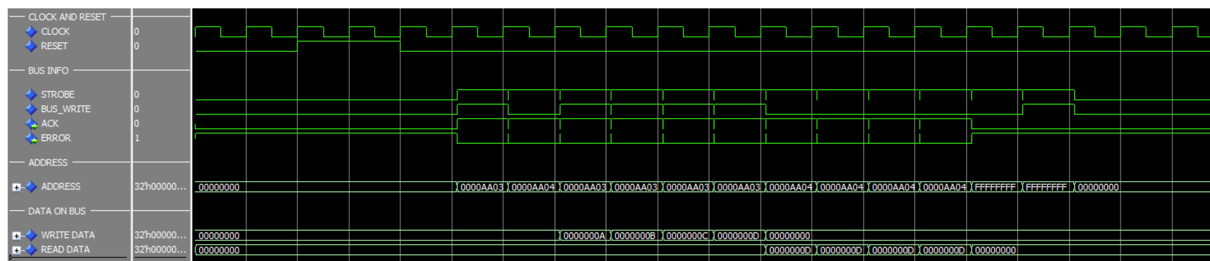


Figure 22 - Example of a wave diagram from a small testbench

In small testbenches, like the one shown in Figure 22, it is possible to confirm that the behaviour is correct. But when the testbenches get longer and more complex, this would be a tedious task to do manually. Therefore, there is an automatic way to give a summary of the testbench at the end of the simulation. This tool is called Universal VHDL Verification Methodology (UVVM) [1].

5.1.1 UVVM

UVVM [1] is an open-source tool, made by a Norwegian company named Bitvis, for making a structured testbench in VHDL. It provides useful functionality that can make sure that the outputs of the design under testing (DUT) have the expected values. It can wait on a signal to change if the timing varies, and it can stop the simulation if a given time has passed without getting an answer from the DUT. After the simulation has finished, it will give a report written in a text file including all the information about what types of tests were performed and the result. The report looks something like this:

```
UVVM:==========================================================================================
UVVM: ID_LOG_HDR_LARGE  0.0 ns  TB seq.          Start Simulation of TB for MJCU_tb
UVVM:==========================================================================================
UVVM:
UVVM: ID_LOG_HDR        40.0 ns  TB seq.          Write to PSU address 1 -> 5 via module_01
UVVM: -----------------------------------------------------------------------------------------
UVVM:
UVVM: ID_LOG_HDR        91.0 ns  TB seq.          Write to PSU address A -> D via module_43
UVVM: -----------------------------------------------------------------------------------------
UVVM:
UVVM: ID_LOG_HDR        900131.0 ns  TB seq.      Writing/reading to/from DUMMY_MODULE
UVVM: -----------------------------------------------------------------------------------------
UVVM:
UVVM: ID_LOG_HDR        901151.0 ns  TB seq.      Read from PSU address 1 -> 5 via module_01
UVVM: -----------------------------------------------------------------------------------------
UVVM:
UVVM: ID_LOG_HDR        901201.0 ns  TB seq.      Read from PSU address A -> D via module_43
UVVM: -----------------------------------------------------------------------------------------
UVVM:
UVVM: ID_LOG_HDR        911241.0 ns  TB seq.      Changes baud rate to 3Mbit/s
UVVM: -----------------------------------------------------------------------------------------
UVVM:
UVVM: ID_LOG_HDR        911251.0 ns  TB seq.      Global write to all PSU reg A
UVVM: -----------------------------------------------------------------------------------------
UVVM:
UVVM:==========================================================================================
UVVM:      *** FINAL SUMMARY OF ALL ALERTS ***
UVVM:==========================================================================================
UVVM:                               REGARDED   EXPECTED  IGNORED      Comment?
UVVM:               NOTE         :      0         0         0         ok
UVVM:               TB_NOTE      :      0         0         0         ok
UVVM:               WARNING      :      0         0         0         ok
UVVM:               TB_WARNING   :      0         0         0         ok
UVVM:               MANUAL_CHECK :      0         0         0         ok
UVVM:               ERROR        :      0         0         0         ok
UVVM:               TB_ERROR     :      0         0         0         ok
UVVM:               FAILURE      :      0         0         0         ok
UVVM:               TB_FAILURE   :      0         0         0         ok
UVVM:
UVVM:==========================================================================================
UVVM:      >> Simulation SUCCESS: No mismatch between counted and expected serious alerts
UVVM:==========================================================================================
UVVM:
UVVM:
UVVM:
UVVM:
UVVM: ID_LOG_HDR        1813771.0 ns  TB seq.     SIMULATION COMPLETED
UVVM: -----------------------------------------------------------------------------------------
```

Listing 6 - UVVM log report

The UVVM-libraries required for a given testbench must be compiled before they can be used. UVVM uses do-files to do this. The file named "compile_all.do" is located inside the "script"-folder and is executed by calling it with three arguments.

```
# Compile UVVM Dependencies (compile_all.do)
#     This file can be called with three arguments:
#     arg 1: Part directory of this library/module
#     arg 2: Target directory
#     arg 3: Path to custom component list file
# do [simulation file (.do)] [ arg 1 ][ arg 2 ][ arg 3 ]
```

Listing 7 - UVVM do-file

The component list mentioned above is a text file that includes the names of the libraries that the user wants to compile.

5.1.2 DO-file

The use of do-files to compile libraries, and one example project that used it to run a simulation, inspired the use of do-files in this thesis. All the modules have individual testbenches with a do-file to run them. All of the compiling and simulation in Multisim or QuestaSim can be done through the terminal manually. This is a tedious task if the DUT has multiple files. A do-file is a tool for automating this process. The ones in this project create a folder named “sim”. Then it compiles all the UVVM- and IPbus-libraries necessary for the testbenches and components into the folder. After all the external libraries are done, it compiles all of the components for the DUT and the package file made for MJCU. Then it simulates the testbench and opens the wave diagram.

To simulate a module or the MJCU itself in either Multisim or Questa Sim, navigate to the “script” folder in the desired module and run the do-file by typing “do sim.do” in the terminal. This will compile all of the necessary libraries, package files and components and simulate the testbench. It is also possible to run the sim.do file with an argument to save time the next time the user wants to simulate the module. It’s the same commando but followed by an argument. For example, “do sim.do example”.

5.2 Hardware testing

Physical testing is performed to test if the parts work as intended in the real world. This should theoretically be done in the same environment as the part will be living in. This is because it could be affected by external interference. There is also a possibility that the communication protocol was misunderstood by one or both ends of a transmission line. This happened more than once in this case. The first error between the MJCU and the PCUs was the Read/write bit. It was the wrong way around. Then it became clear that the chip on the PCUs that deals with the communication sends the data with LSB first and MSB last. The reason this didn’t cause an error in the simulation was that both the master and slave were created by the same person. This was a known risk when it was created. It is often easier to create a working communication if the same person creates both sides. If the protocol is misunderstood, it may not matter because both the master and the slave would have the same flawed protocol. This is the reason why good documentation is important.

Oscilloscope verification

Oscilloscopes are a good tool to verify correct behaviour in smaller data transfers. They make it possible to see how signals behave instead of just looking at the result on the computer screen. The two figures, Figure 23 and Figure 24 below show the first successful test between the control room and a prototype of the PCU via the MJCU. The prototype was not configured to synchronize the transfer with the clock signal generated by the MJCU. Instead, it was using a baud rate of 115200 just like the standard. The timings were not perfect, so it is possible to see some shifting in Figure 24.

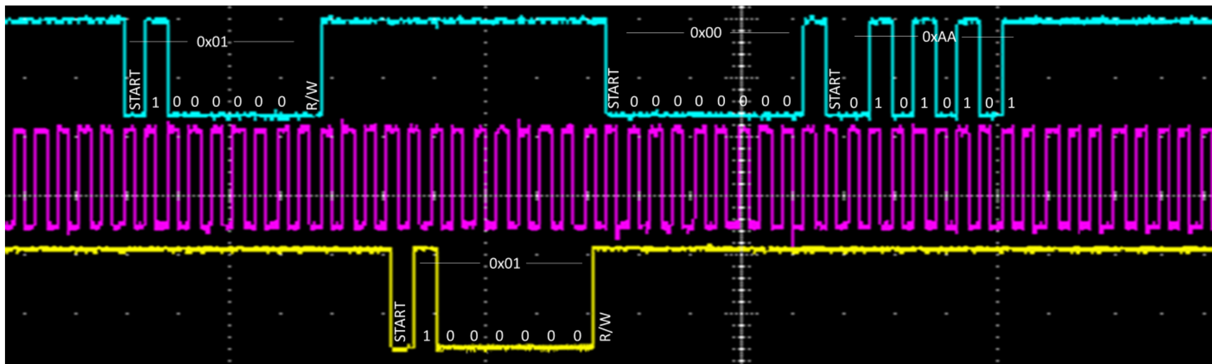


Figure 23 - MJCU writing 0x00AA to PCU-address 0x01 (asynchronous)

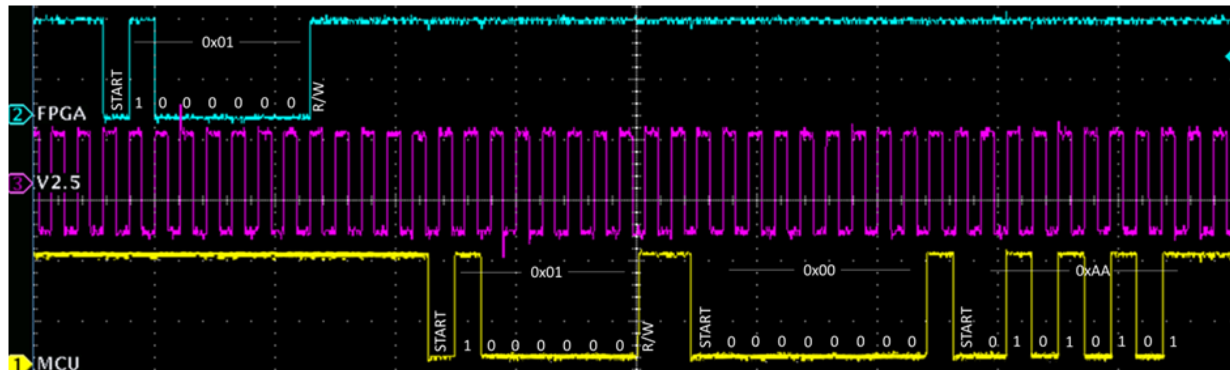


Figure 24 - MJCU reading 0x00AA from PCU-address 0x01 (asynchronous)

Large scale tests

The first large sample size test was done after the PCU prototype was able to use the USART clock to synchronize data transfer. The test was to write a random 16bit-value to a given register inside the PCU, then read from the register 1000 times. This was repeated 100 times to get a mean number of errors. The errors only occurred on the data bits, never the address bits. This test was performed multiple times to find the error rate of different baud rates. That resulted in a bit error of around 4% when using 57 600, 115 200, 230 400 and 460800. While 921 600 and 1 000 000 gave an error rate of 0%.

Baud rates	Mean bit error
57 600	**4.3%**
115 200	**4.0%**
230 400	**4.2%**
460 800	**4.4%**
921 600	**0.0%**
1 000 000	**0.0%**

Table 5 - Bit-error test with varying baud rates, performing 1000 read operations 100 times.

The reason for this behaviour is unclear. A possible cause might be a delay in the communication. At lower baud rates a notable delay can be noticed in the response from the PCUs. The delay is observed to be different from transaction to transaction, which could explain the spread of errors on the graph in Figure 25.

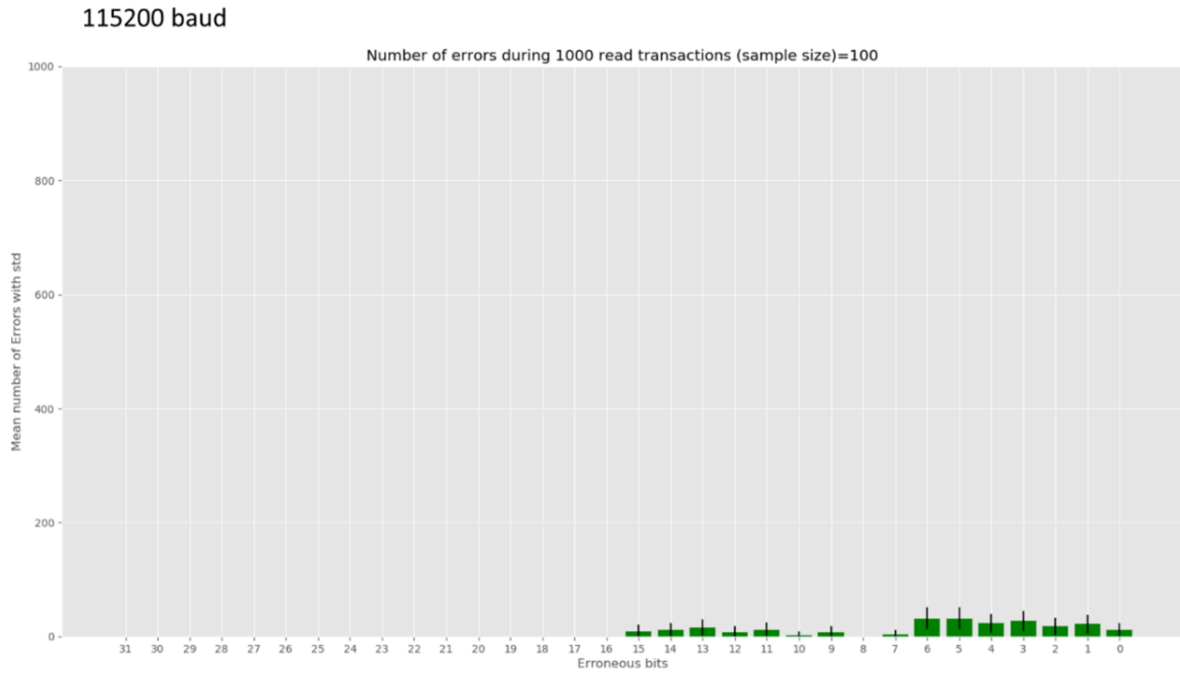


Figure 25 - Mean bit error using 115200 baud

The last test was done over a period of 17 hours. 60 million transfers were successfully transferred using a baud rate of 921600 without a single error. Depicted in Figure 26.

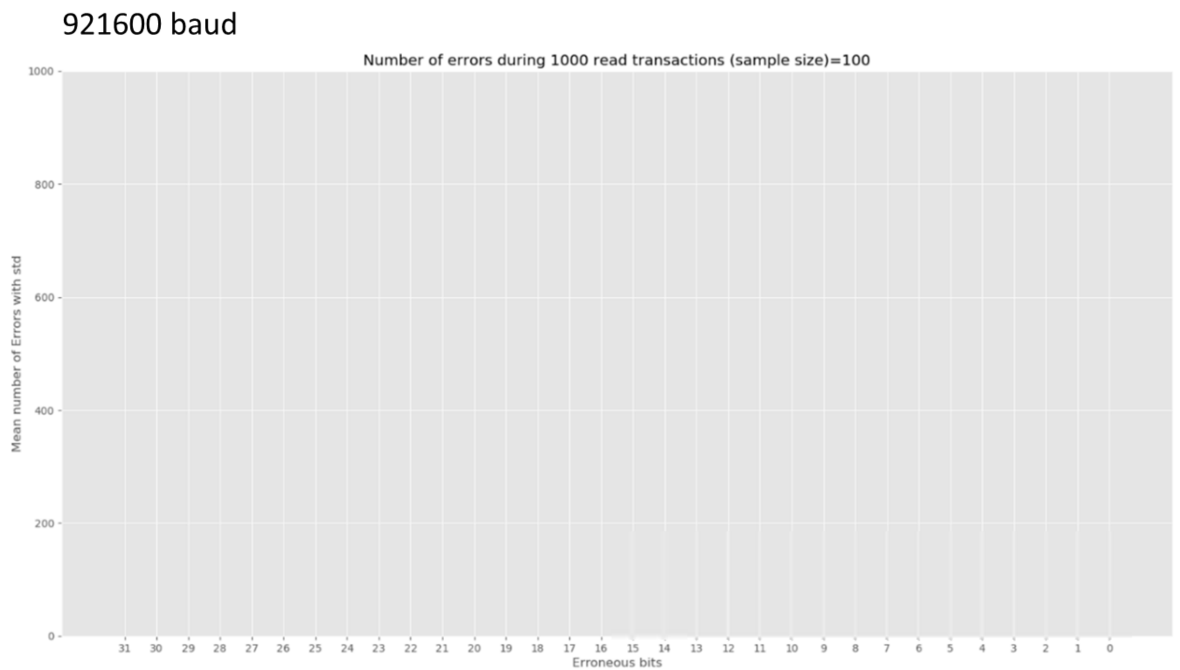
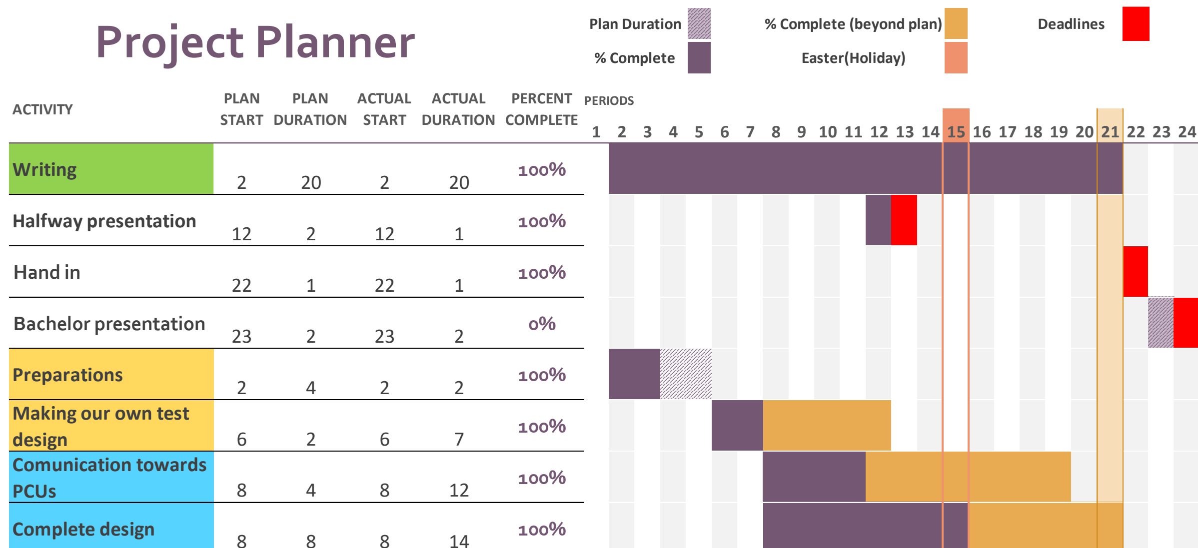


Figure 26 - Mean bit error using 921600 baud

6 Discussion

The project plan shown in Figure 27 was created to make sure we had enough time to learn how to use the different programs, Linux, and GitHub before starting the development of the MJCU. The time required was a bit shorter than expected for all activities except one. We were supposed to make a test design in Vivado and upload it to the FPGA. Nothing more advanced than a blinking LED, or something along those lines. Unfortunately, there were some troubles regarding the licenses we had available for using Vivado, so this activity had to be delayed until the licences were working. But because everything else took less time, we ended up being ahead of the schedule. The communication towards the PCUs was estimated to take four weeks because it was similar to something we had created before, and the protocol was already established. This ended up being the most time-consuming part of the project because the protocol changed a bit under the development and testing.

The goal was to finish the design before the Easter holiday. That would leave us with enough time to comfortably create the MJCU. This goal was meant to be a bit out of reach to make us work harder in advance of the holiday. In reality, the design was ready for its first physical test before the deadline. The result of this test was that the communication between the control room and FPGA was flawed. This was a quick and easy fix, and it ended up working the week after the holidays. The next four weeks went into further development on the communication towards the PCUs. In the third week, it was kind of working but with some hiccups. In the fourth week, this was corrected and it was working as intended.



ACTIVITY	PLAN START	PLAN DURATION	ACTUAL START	ACTUAL DURATION	PERCENT COMPLETE
Writing	2	20	2	20	100%
Halfway presentation	12	2	12	1	100%
Hand in	22	1	22	1	100%
Bachelor presentation	23	2	23	2	0%
Preparations	2	4	2	2	100%
Making our own test design	6	2	6	7	100%
Comunication towards PCUs	8	4	8	12	100%
Complete design	8	8	8	14	100%

Figure 27 - Project plan

7 Conclusion and Outlook

7.1 Conclusion

An FPGA solution (MJCU) that can transfer data between a control room and multiple Power Control Units has been developed and is working as intended. The communication towards the control room is using IPbus. A well-documented and tested communication protocol developed at CERN for communicating with address-aware hardware inside an FPGA. The control room is the master of the power control system in the Bergen pCT project. It uses the SW side of IPbus to communicate with the MJCU. MJCU consists of multiple types of slaves: 43 com_modules, a global module, and a dummy module. The 43 com_modules are responsible for communicating with all of the PCUs using a custom USART protocol. The global module can forward data to all of the com modules at the same time. The dummy module is a feature to test the communication between the control room and MJCU.

Every module has its own testbench that could be simulated using a script. Download the powermonitor repo, navigate to a desired module, and run the "sim.do"-script using either Questa Sim or Multisim. This will compile everything necessary and run the simulation to show how the module behaves and what features it has. The MJCU itself has a testbench where it is connected to dummy_PCUs to show its behaviour and functionality. All the testbenches use UVVM for verification and logging.

In conclusion, the design has been proven to work as intended, and the design process has been conducted with a focus on documentation and verification. Hardware tests have shown that the communication from top-level software to the PCU microcontroller is reliable and stable. The requirements that were set at the start are met. Extensive testbenches are designed, not only for the complete build, but also at the level of each individual component.

7.2 Outlook

The plan was to design two more modules. A *version control* and a *housekeeper*. The version module would keep track of the version and GitHash and have it available for the control room to read. There are some internal sensors on the Xilinx Evaluation Board that are possible to read from via I2C. The housekeeper module was supposed to take care of this.

These two modules are left to be designed. What comes next would be to connect all three parts of the system, the control room, the KCU105 Evaluation Board and the PCUs together physically. The communication with the PCU has only been done using single-ended communication signalling for the clock and the two data connections (TX and RX). This remains to be swapped with bidirectional low-voltage differential signalling (LVDS). This means that the RJ45 connector rack needs to be set up, and testing needs to be done to make sure everything works at the intended scale. The control room and the PCUs need to be finalized as well and then operate as a complete working system.

8 Bibliography

[1] Bitvis, “GITHUB - UVVM,” 5 5 2022. [Online]. Available: https://github.com/UVVM/UVVM. [Accessed 11 5 2022].

[2] M. Eggen and J. R. Hauser, “GitAppUiB,” 5 2022. [Online]. Available: https://git.app.uib.no/pct/powermonitor.

[3] T. Bodova, “High-Speed Signal and Power Distribution of,” Department of Physics and Technology - University of Bergen, Bergen, 2020.

[4] “AMD - Xilinx,” 2022. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/kcu105.html#specifications. [Accessed 21 05 2022].

[5] ipbus - cern, “cern.ch,” 20 06 2021. [Online]. Available: https://ipbus.web.cern.ch/doc/user/html/firmware/bus.html.

[6] H. Birkenes, *Design of Configuration and Monitoring System for Power Supply in ProtonCT Project,* Bergen: Institute of Physics and Technology - University of Bergen, Soon to be released.

[7] B. Olsen, *Power and Monitor Solution for the ProtonCT,* Bergen: Institute of Physics and Technology - University of Bergen, Soon to be released.

Appendix A Development Tools

Software:

- Vivado 2019
- Questa Sim 2020.4
- GitHub

Hardware:

- KCU105 Evaluation Board featuring the Kintex UltraScale XCKU040-2FFVA1156E FPGA

Appendix B Address map for all modules inside the MJCU

A.1 Module addresses

All the slaves inside the MJCU have different addresses.

From IPbus SW -> IPbus HW-slaves

They are as follows.

NAME	ADDRESS	
	HEX	DEC
GLOBAL_MODULE	0x00	0
COM_MODULE_1	0x01	1
COM_MODULE_2	0x02	2
...	...	…
VERSION	0x41	65
HOUSEKEEPER	0x42	66
DUMMY_MODULE	0x43	67

Table 6 - Module addresses

A.2 Address map

Address map for all internal slaves inside the MJCU including two examples.

From IPbus SW -> IPbus HW-slaves

Name	Register address	Width	Access	Default Value	Description
GLOBAL_MODULE (module address 0x00)					
GLOBAL_CONTROL	0x01	32	W	0x00	Not in use
GLOBAL_STATUS	0x02	32	R	0x00	Not in use
GLOBAL_WRITE	0x03	32	W	0x00	Writes data to every com_module in the system. [31] R/$\overline{W}$ bit towards PCU [30..24] Not in use [23..16] PCU reg address [12..0] Data to PCU
GLOBAL_RESET	0x05	32	W	0x00	Resets all modules except the global_module
COM_ENABLE_0	0x06	32	R/W	0xFFFFFFFF	These 32 bits enable com_modules[1→32] '1' = ENABLED Bit[n] controls com _module_[1+n] This function is made in the case that modules are not connected to a PCU.
COM_ENABLE_1	0x07	32	R/W	0xFFFFFFFF	These 32 bits enable com_modules[33→64] '1' = ENABLED Bit[n] controls com _module_[33+n] This function is made in the case that modules are not connected to a PCU.
SYSTEM CLOCK SPEED	0x08	32	R/W	0x1D905C0	The speed of the clock used by the com_modules. This is used to calculate the baud rate towards the PCUs. Default: 31Mhz (clock from IPbus)
PSU BAUD RATE	0x09	32	R/W	0x1C200	The baud rate to use towards the PCUs
COM_MODULE (module address 0x01 – 0x40)					
CONTROL	0x01	32	W	0x00	Not in use
STATUS	0x02	32	R	0x00	[31..4] Not in use [3] TX is full (1 = full) [2] TX is empty (1 = empty) [1] RX is full (1 = full) [0] RX is empty (1 = empty)
WRITE	0x03	32	W	0x00	Writes data to the corresponding PCU. [31] R/$\overline{W}$ bit towards PCU [30..24] Not in use [23..16] PCU reg address [12..0] Data to PCU

Address map for all modules inside the MJCU

Name	Register address	Width	Access	Default Value	Description
READ	0x04	32	R	0x00	Reads a 32-bit vector from the corresponding PCU. [31] R/$\overline{W}$ bit towards PCU [30..24] Not in use [23..16] PCU reg address [12..0] Data from PCU
RESET_FIFO	0x05	32	W	0x00	Resets both TX and RX FIFO to clear them.
DUMMY_MODULE (module address 0x43)					
WRITE	0x03	32	W	0x00	Write anything to store a new value
READ	0x04	32	R	0xDEADBEEF	Read back the stored value

Table 7 - Address map for MJCU

Address for reaching [NAME] register: 0x[MODULE ADDRESS] & [REGISTER ADDRESS]

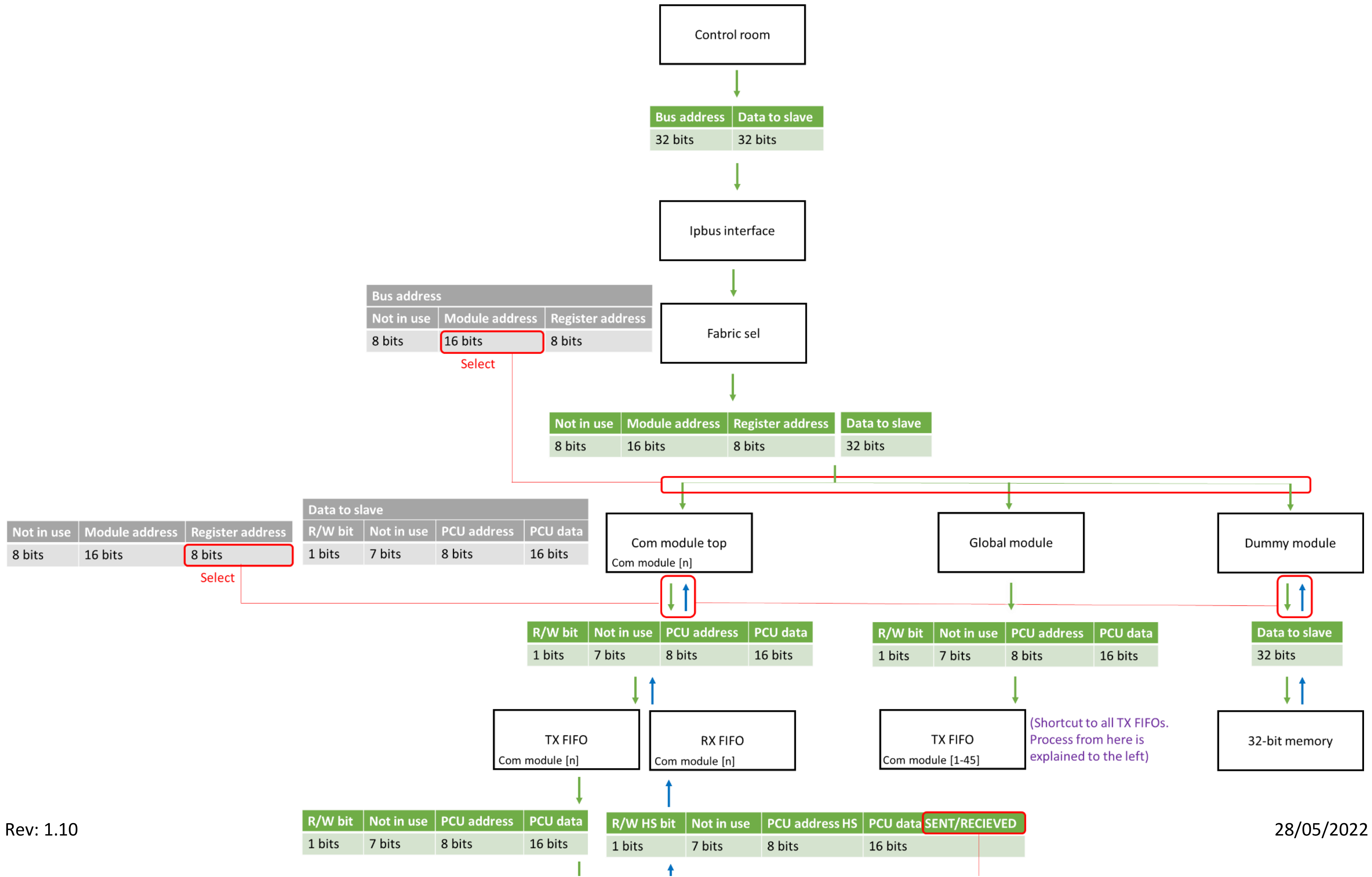
Example:

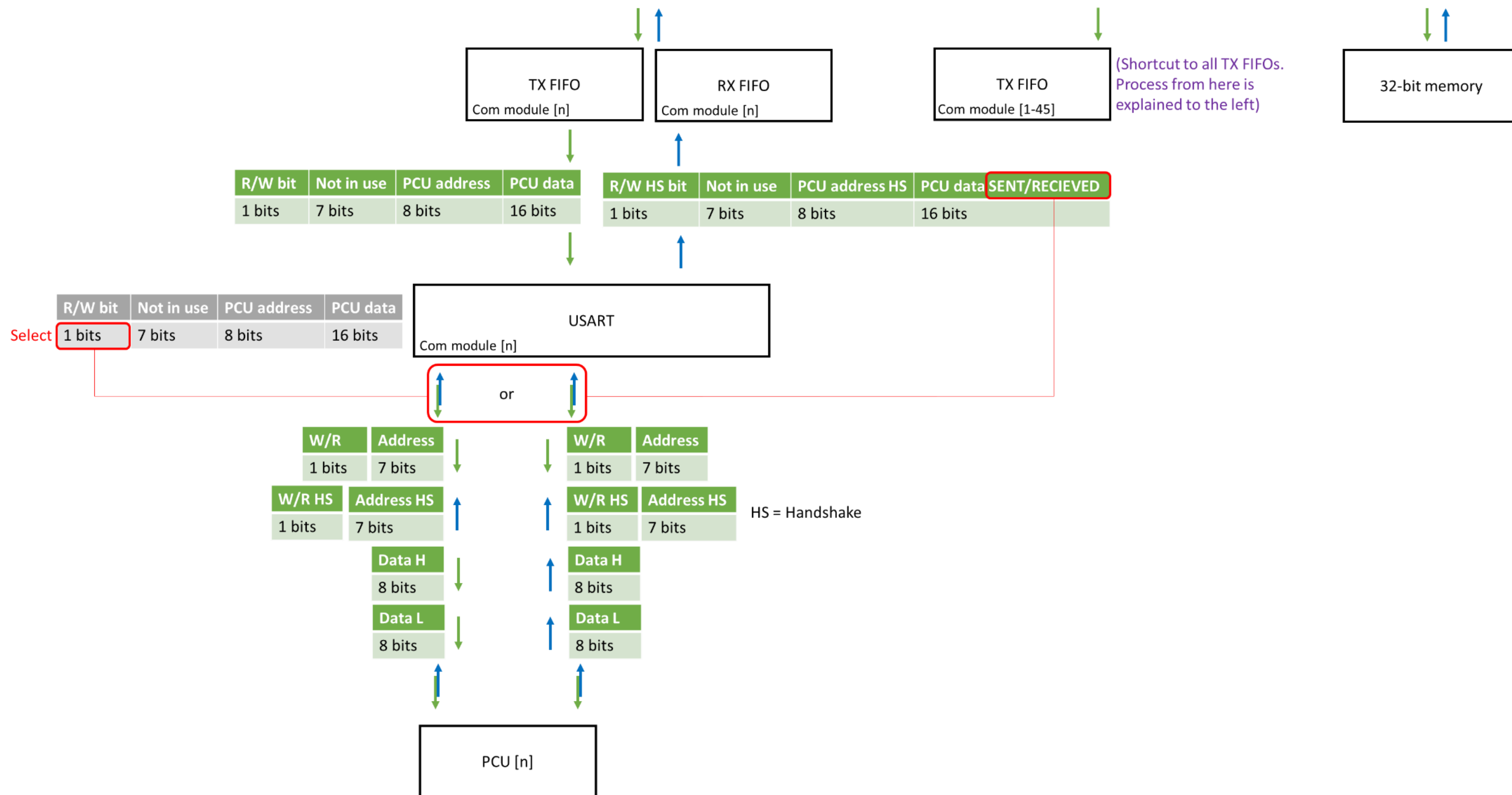
To reach WRITE in global_module: 0x0003

To reach READ in com_module_23: 0x1704

Appendix C DATA FLOW

Data flow from the Control room to the PCUs and back

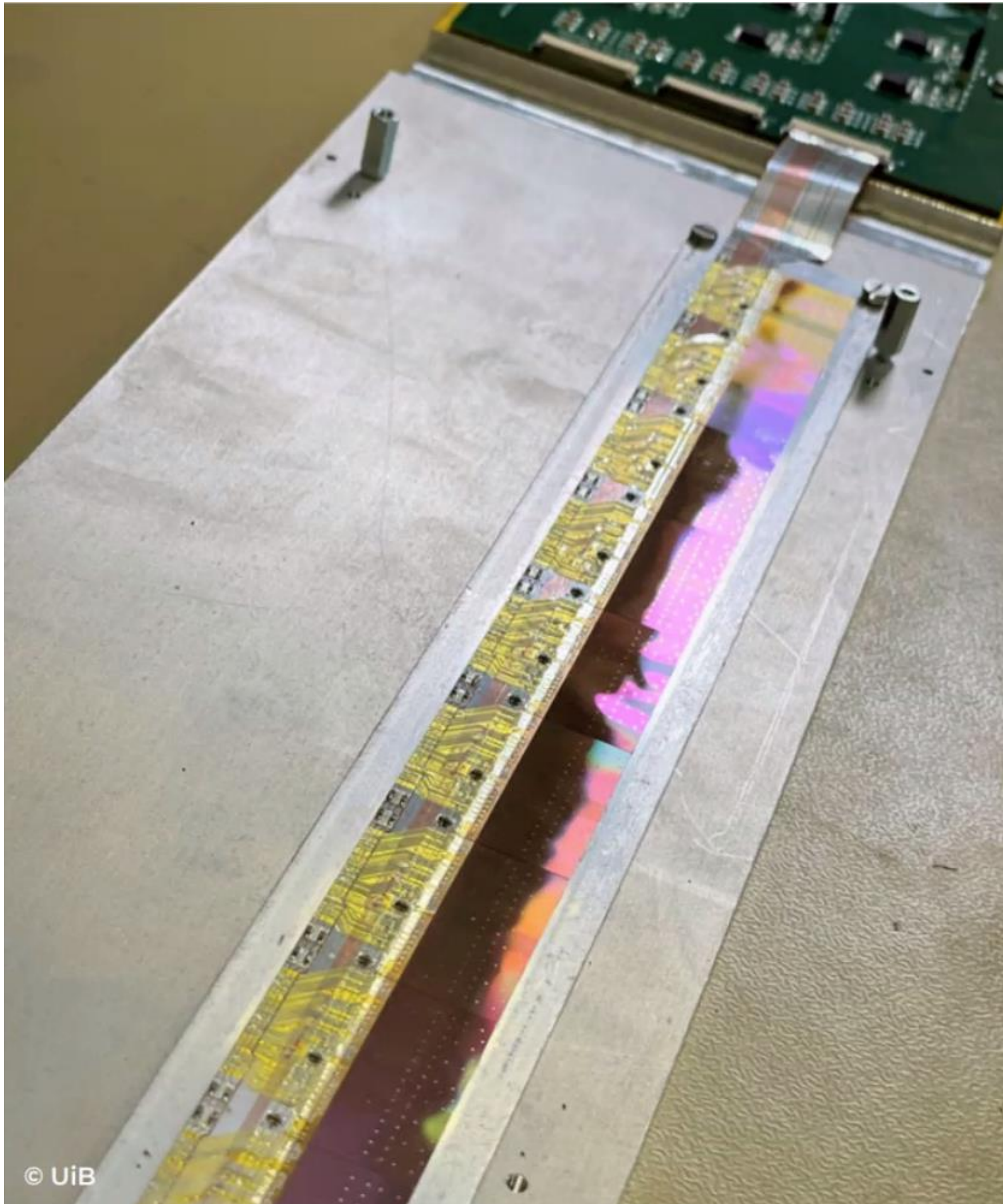




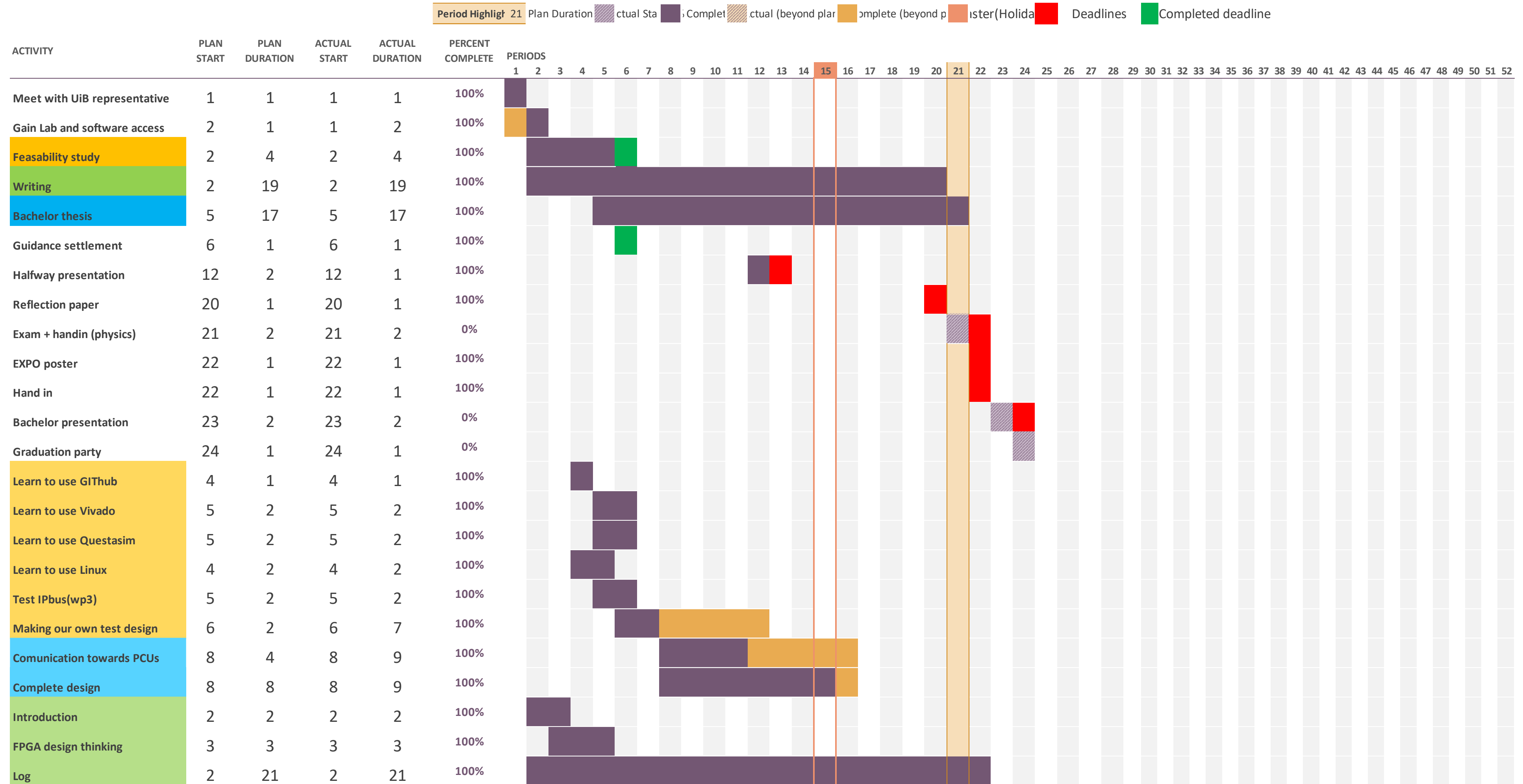
TX FIFO
Com module [n]
RX FIFO
Com module [n]
TX FIFO
Com module [1-45]
(Shortcut to all TX FIFOs. Process from here is explained to the left)
32-bit memory
R/W bit
Not in use
PCU address
PCU data
1 bits
7 bits
8 bits
16 bits
R/W HS bit
Not in use
PCU address HS
PCU data
SENT/RECIEVED
1 bits
7 bits
8 bits
16 bits
R/W bit
Not in use
PCU address
PCU data
Select
1 bits
7 bits
8 bits
16 bits
USART
Com module [n]
or
W/R
Address
1 bits
7 bits
W/R HS
Address HS
1 bits
7 bits
Data H
8 bits
Data L
8 bits
W/R
Address
1 bits
7 bits
W/R HS
Address HS
1 bits
7 bits
HS = Handshake
Data H
8 bits
Data L
8 bits
PCU [n]

Appendix D Picture of an ALPIDE sensor

This picture shows the sensors used inside the calorimeter



Appendix E Project plan



ACTIVITY	PLAN START	PLAN DURATION	ACTUAL START	ACTUAL DURATION	PERCENT COMPLETE
Meet with UiB representative	1	1	1	1	100%
Gain Lab and software access	2	1	1	2	100%
Feasability study	2	4	2	4	100%
Writing	2	19	2	19	100%
Bachelor thesis	5	17	5	17	100%
Guidance settlement	6	1	6	1	100%
Halfway presentation	12	2	12	1	100%
Reflection paper	20	1	20	1	100%
Exam + handin (physics)	21	2	21	2	0%
EXPO poster	22	1	22	1	100%
Hand in	22	1	22	1	100%
Bachelor presentation	23	2	23	2	0%
Graduation party	24	1	24	1	0%
Learn to use GIThub	4	1	4	1	100%
Learn to use Vivado	5	2	5	2	100%
Learn to use Questasim	5	2	5	2	100%
Learn to use Linux	4	2	4	2	100%
Test IPbus(wp3)	5	2	5	2	100%
Making our own test design	6	2	6	7	100%
Comunication towards PCUs	8	4	8	9	100%
Complete design	8	8	8	9	100%
Introduction	2	2	2	2	100%
FPGA design thinking	3	3	3	3	100%
Log	2	21	2	21	100%

Figure 28 - Project plan for BO22EB-08