CHALMERS
UNIVERSITY OF TECHNOLOGY

Network Implementation in an FPGA

Master's thesis in Engineering Physics

ANDERS FURUFORS

Department of Physics
Chalmers University of Technology
Gothenburg, Sweden 2019

Network Implementation in an FPGA

Anders Furufors

Department of Physics
Division of Subatomic and Plasma Physics
Chalmers University of Technology
Gothenburg, Sweden, 2019

Network Implementation in an FPGA
Anders Furufors

Examiner: Andreas Heinz
Supervisor: Håkan T. Johansson

Master's Thesis 2019: Department of Physics
Division of Subatomic and Plasma Physics
Chalmers University of Technology
SE 412-96 Göteborg
Sweden

Chalmers ReproserviceTypeset in LaTeX
Gothenburg, Sweden 2019

Network Implementation in an FPGA
Anders Furufors
Department of Physics
Division of Subatomic and Plasma Physics
Chalmers University of Technology

Abstract

A limited network stack is implemented on an FPGA for use in nuclear physics experiments to transfer data directly from front-end electronics to a PC over Ethernet. This is done using an FPGA board equipped with a connector attached to a PHY chip. Code was written for this FPGA to handle the physical and most of the link layer. Higher layers of network communication were implemented in the FPGA, using a hardware design called Fakernet. Fakernet handles the rest of the link layer as well as network and transport layer protocols.

This results in a design that sufficiently handles protocols needed for establishing a connection between nodes on a network (ICMP, ARP) and to be configured using a UDP interface. It is able to send data to a PC using TCP, reaching line speed with the PHY chip's 100 Mb/s interface while at the same time having low resource usage, occupying less than 2 per cent of the LUTs of an FPGA used in the type of experiment the design is considered for. This design was tested for various situations and restrictions and it is concluded that it has potential for being used in real experiments in the future even though Fakernet is not yet fully developed.

Keywords: Data Acquisition, Network Stack, Ethernet, FPGA, Hardware Description, Data Transport, Nuclear Physics, Nuclear Physics Experiments

Acknowledgements

I am most grateful for the support I have received in my efforts to write this thesis. I would like to thank Håkan T. Johansson for teaching me more than I knew I wanted to know, for always helping me no matter how often I got stuck and for constantly knowing what to do next. I wish to thank Andreas Heinz for checking in on me daily, making sure the work progressed and that I had everything I needed. Furthermore I would like to thank Björn Jonsson and Thomas Nilsson for giving me the opportunity to visit CERN and participate in a real nuclear physics experiment. Thanks also to Giovanni Bruni who shared his office with me and guided me at CERN. Lastly I wish to thank everyone else at the department for makig my time there enjoyable.

Contents

Glossary

ADC - Analog to Digital Converter

ARP - Address Resolution Protocol, network layer

Bandwidth - Transfer rate of information

BDP - Bandwidth-Delay Product

Bit, b - Smallest unit of information with only two possible values

Bitstream File - Firmware File loaded onto the FPGA, defining its behavior

BRAM - Block RAM, memory structure in FPGAs

Bottleneck - Network component with smallest bandwidth, limiting the whole network's data transfer rate

Byte, B - Smallest chunk of addressable memory; an octet of bits when used in this text

C - A general purpose programming language

CPU - Central Processing Unit

DPRAM - Dual ported RAM

Ethernet - Computer network technology, link layer

Fakernet - Partial network stack in VHDL written by Håkan T. Johansson

Firmware - Software that determines the hardware behavior at a low level

Flip-flop - Electronics component that stores data between clock cycles

FPGA - Field Programmable Gate Array

HDL - Hardware Description Language

Hub - Hardware unit that forwards packets to all its ports on a network (c.f. switch)

ICMP - Internet Control Message Protocol, network layer

IP - Internet Protocol, network layer

LED - Light Emitting Diode

LUT - Lookup Table, an electronics component

MAC Address - Media Access Control Address, 6 octets, for example (hexadecimal notation) 01:12:ab:2f:78:99

Netcat, nc - Software used to establish, read from and write to a TCP to a connection from the command line

Network Stack - Software implementation of a family of communication protocols

Nibble - Chunk of data, smaller than a byte. 4 bits in this project.

Octet - Eight bits of information

Overhead - Data of a protocol that contains information about the communication rather than the message

Packet - Collection of information defined by protocols, with a beginning and end

Payload - The part of a packet that is the message and not overhead

PHY - Physical layer chip

Ping - Software used to measure RTT and packet loss

Place and Route - Design stage that consists of placing and interconnecting logic elements on the FPGA

RAM - Random Access Memory

Receive Window - Amount of data that the sender in a TCP connection is allowed to send before receiving acknowledgments

RJ45 - Standard jack for Ethernet

RTT - Round-Trip Time, time between sending a packet to and receiving the answer from another node

Store-and-Forward - Mode on some network switches that makes it receive the whole packet before forwarding it

Switch - Hardware unit that forwards packets to the port associated with the destination on a network (c.f. hub)

Synthesis - The process of realizing a HDL algorithm in terms of logic gates. Similar to compilation

TCP - Transmission Control Protocol, transport layer

UART - Universal Asynchronous Reciever and Transmitter

UDP - User Datagram Protocol, transport layer

USB - Universal Serial Bus

VHDL - Very High Speed Integrated Circuit Hardware Description Language

Vivado - Software used for compiling HDL to a bitstream file, used in Xilinx chips

Wireshark - Software used to monitor and analyze network traffic

1 Introduction

In experimental setups, such as the R^3B setup at GSI [1], there may be over ten detectors, many of them with thousands of electronic channels, with accompanying ADCs, making up the front end electronics [2]. In such setups, the use of custom protocols and custom-designed hardware is a problem because they are, due to their limited use, both costly and prone to bugs [3]. The purpose of this project is to reduce the need for those by implementing a limited but sufficient network stack (software implementation of communication protocol layers) on a Field Programmable Gate Array (FPGA) in order to transport the data to the PC over Ethernet.

Using Ethernet has several advantages. It is a standardized technology, of which there is plenty of documentation available. It is also tested and well-developed after many years of use. One important factor due to the large number of detectors in these kinds of experiments is the price, and using Ethernet allows for using readily available, well-developed and affordable commercial equipment rather than using custom-designed hardware and protocols. Such a design would allow for a standard system that is usable for all front-end electronics. Furthermore, having fewer bugs would decrease downtime during experiments, which wastes valuable beam time.

In modern nuclear physics experiments, the data acquisition is done in multiple steps. The goal is to record data, which allows for studying the reactions of the experiment. These reactions are studied indirectly by measuring properties of their resulting reaction products. Their properies are measured by detectors that generate electronic signals. The signals are digitized in analog-to-digital converters (ADCs) and from there transported to a PC, or other type of computer, and stored in permanent memory. See figure 1. The transport of data from the ADC to the computer is the main interest of this work. It can be done in different ways, but most often involves an FPGA to control the ADC (a CPU being to slow, see section 2.2) and to do on-board signal processing, which then through hardware and protocols sends the data to a PC.

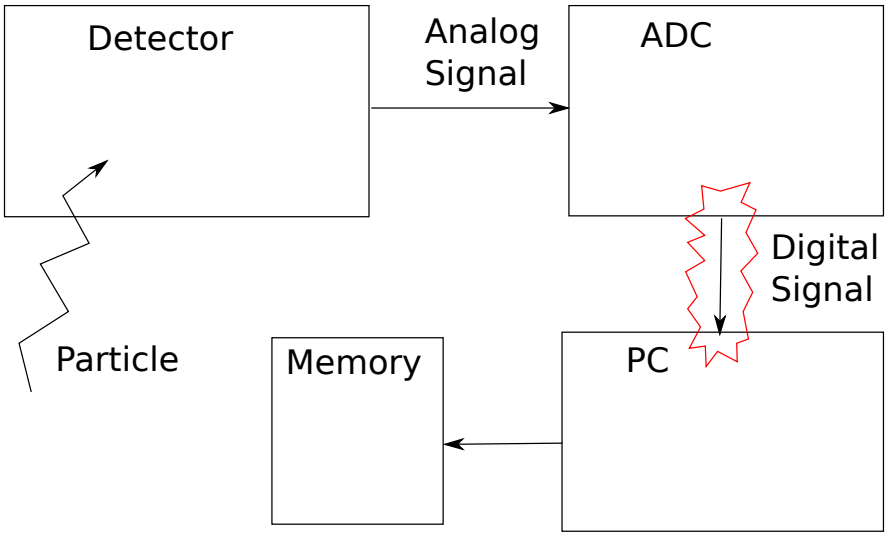

Figure 1: Schematic overview of an event. Some particle hits a detector, induces a voltage that is digitized and saved in some memory of a PC. The red cloud encloses the main concern of this report, the path between the ADC and the PC, both hardware and protocols.

The general idea of this work is to use an FPGA board, capable of receiving and transmitting signals over Ethernet, to implement a limited but sufficient network stack called Fakernet. It should be able to allow for connections to be established with other nodes on the network and communicate with them using data transfer protocols. This is to be used in sending data from an FPGA that controls an ADC, to a PC connected to the network, which is then able to receive and store the data. This design should be made suitable for use in, for example, nuclear physics experiments, handling the transport of digital signals.

2 Background

In the following chapters, a few topics that are relevant to the project are discussed, to give the reader some ground to stand on for the following chapters.

2.1 Nuclear Physics Measurement Techniques

Nuclear physics experiments can be complex, in part due to the scales involved (small sizes, high speeds, short time intervals and high reaction rates). One common way of studying nuclei is to induce reactions by a beam of particles hitting a target. These reactions can cause the emission of photons and massive particles that in turn we detect. One can gain knowledge of the reaction by analyzing their properties, thereby learning about the physics behind the processes.

The experimental setups used in these kinds of experiment often involve many instruments. There are particle accelerators of different kinds (linear, cyclotrons and synchrotrons), vacuum pumps, magnets, detectors of different kinds, ADCs and data acquisition systems.

Data acquisition is very important for this project. Generally when particles from the reactions hit detectors, they deposit energy and the material in the detector is excited or ionized, so that a voltage difference can be measured. These signals are converted from analog to digital signals using ADCs that are most often controlled by FPGAs. Eventually this data will end up in a PC, or another type of computer, where it is written to some file in permanent, non-volatile memory, to later be analyzed. See figure 1 for a schematic overview of the data acquisition. The data acquisition is triggered upon an event, that is a signal from a detector, saving data for a period of time. During this time, more particles may be detected, increasing the multiplicity of the event. In this way coincidences can also be measured. There are also methods of zero supression to reduce the amount of data transported and saved [3].

2.2 FPGA

Of the mentioned electronics, the FPGA is the main focus of this thesis. A field-programmable gate array (FPGA) is a reconfigurable integrated circuit

that is configured by the user to perform the desired calculations [4] [5]. It differs from a central processing unit (CPU) found in regular PCs in that the CPU can execute arbitrarily ordered instructions. This means that different programs can be run on the CPU; an FPGA on the other hand is limited to the task determined by its configuration. This means an FPGA is not as versatile, but what we lose in versatility, we gain in speed. While a regular CPU only performs a few operations during each clock cycle, the FPGA can perform a very large number of operations (limited only by the number of components on the chip). So when speed or short response time are desired for a set of parallel tasks, an FPGA may be preferable to a regular CPU [5].

In accelerator-based nuclear physics experiments, FPGAs are useful due to the requirement of fast measurements. To allow for high event rates, the logic controlling various components (such as ADCs) must be fast, making FPGAs a suitable technology.

When designing code for an FPGA, it may be useful to use a development board. These are equipped with tools that make debugging easier, such as indicator lights, switches and IO pins. The reason for this is that debugging is much harder because of the lack of, for example, standard console output that we are used to. The indicator lights can be used for showing in what state a program is, or represent the value of certain signals, while the switches can be used to interact with the program. Schematics of a develpment board are shown in figure 4.

An FPGA consists of a large number of electronic components that allow for storing logic values and performing simple operations on the signals such as the logic *and* or the logic *or*. Lookup tables (LUTs), flip-flops, Block RAM (BRAM), global buffers and Phase Locked Loops (PLLs) are some of the most important components. FPGAs are most often configured by writing a program in a hardware description language (HDL) such as VHDL (see section 4.1.4 and appendix A for details and code examples, respectively). Synthesis software will then translate the code into a design for the electronic components to perform the desired tasks [4]. The number of components inside an FPGA is not unlimited, however, so there is a limit as to how large the program can be [5].

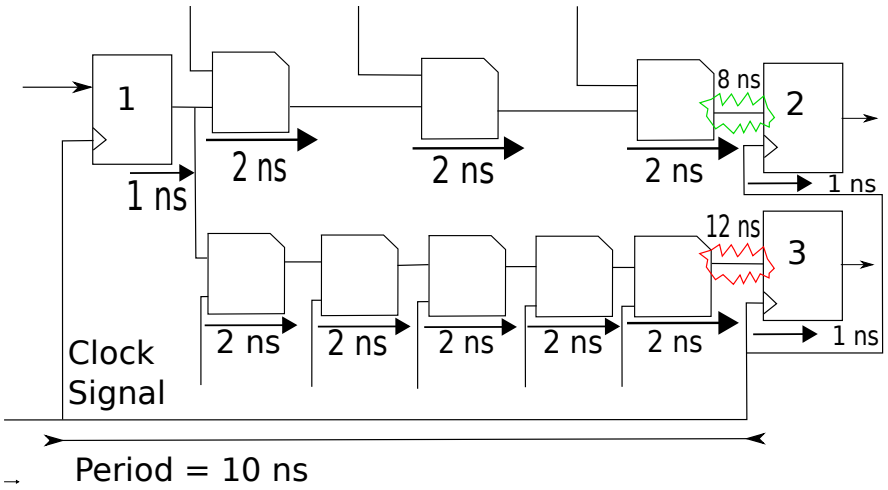

Figure 2: Schematic example of two blocks of logic elements, rectangles with cut corners, between flip-flops, rectangles enumerated 1 to 3. The operations of sampling input (upper left of flip-flop) and setting output (right of flip-flop) of the flip-flops take 1 ns each. These operations are triggered (lower left of flip-flop) by a clock signal with a period of 10 ns. The logic elements compare two signals (left side) and set the output (right side) to the result; the operation takes 2 ns. The signal path from flip-flop 1 to 2 leaves the flip-flop, passes through three logic elements and is sampled in flip-flop 2. This adds up to 8 ns, which is less than the clock period of 10 ns. This means the signal (enclosed in the green cloud) that is set as the output of flip-flop 2 is stable and has the value intended by the code. The lower path, flip-flop 1 to 3, on the other hand has more logic elements, making the propagation time add up to 12 ns, which is more than the clock period of 10 ns. This means the signal (enclosed in the red cloud) that is set as the output of flip-flop 3 does not necessarily have the value that is described in the code.

A key aspect of any FPGA program design is that of timing. In an FPGA, signal values are stored in single-bit registers, often called flip-flops (this can be compared to how variables are stored on a PC, with different levels of CPU cache and RAM). Flip-flops work by having an input signal and an output signal and also a trigger input. It will sample its input signal upon triggering and possibly change and then keep its output at that new value until the next trigger. The trigger input is usually a clock signal that is generated by an on-board oscillator. Because of the interdependency of the signals, the goal is that all flip-flops trigger at the same time, and the number

of logic elements between successive layers of them is low enough that the new signal has time to properly propagate and can be stably (in the sense that it has to the value intended by the code and is not oscillating) sampled at the next flip-flop the following clock cycle. Therefore there is a limit for the number of operations, which can be performed on a signal between the flip-flops, thus limiting how much can be done per clock cycle. The way programs are executed on a regular CPU differs from FPGA programs in that the operations are performed sequentially, meaning the variables are not dependent on each other at every cycle. This means writing HDL code for an FPGA involves the additional challenge of timing to be considered while writing higher level code for a CPU does not. See figure 2 for a schematic example of timing issues.

2.3 Network Communication

Connecting electronic systems into networks is a technology which allows for information sharing between units. It is not trivial to implement, however. Information transport requires media, such as metal cables, optic fibre cables or electromagnetic waves. These allow for information to be transferred between two endpoints, for example cable jacks or antennae.

Between electronic units, the most common information transfer is that of binary values, resulting in a stream of bits (information units with only two possible values). On the physical level, this can be implemented as, for example, voltage differences. To send data like this, a message, called payload, must be broken into individual bits that are transmitted sequentially. Every unit connected to this kind of network therefore needs some common method for doing this, and, naturally, also be able to reassemble an incoming stream of bits into larger data words. This receiving and transmitting of bits is referred to as the physical layer of digital network communication.

To be able to communicate properly between many connected units, there need to be rules that are common for all units. These are defined in so-called protocols, which define what kind of message it is, how the data is sent, from where and to where. The protocol that defines how information packets are transported between nodes on a network is called the link layer. To further define how the packet is transported between nodes that are not necessarily adjacent, protocols called network layer protocols are used. The process of

data transfer itself is defined in the transport layer protocols, which define things such as what happens when a message is not received. This picture is called the layer model of network communication.

There is another concern, namely that of data transfer rate. For the protocols to even matter, the sender and receiver must agree on what rate is to be used in data transmission. This is most often done by letting the units negotiate this speed. Faster units are usually able to also work at a slower rate, meaning units can negotiate a common rate of transmission; it is most often the lowest of the two units' maximum transfer rates.

2.4 UART

Universal Asynchronous Reciever and Transmitter (UART) is a technology for sending and receiving data using asynchronous serial communication. This type of communication is very simple, having the packet's start and end defined by so-called start and stop bits, respectively, and sending the payload of an octet (eight-bit word of information) in between. Its simplicity makes it useful in some situations, being very easy to implement, but its large overhead (20 per cent) and slow transfer rates make it less useful in other situations.

2.5 Ethernet

Ethernet is a standardized technology used in computer networking, defining the link layer protocol. First introduced in 1980, Ethernet has become ubiquitous and is found in most homes, schools and workplaces [6].

Ethernet signals are usually transferred on twisted pair cables or optical fibres between nodes such as network cards on PCs, Ethernet switches and routers that are equipped with RJ45 jacks. On such a network, each unit that is the final recipient or original sender of packets will have a Media Access Control Address (MAC address). Data transmission over these networks consists of packets of data, whose structure is defined by the protocols in use. The data that make up the original message, disregarding the method of communication, is called payload. The packet will consist of more than the payload. For example we need to define the first and the last bit of the

payload and the address of the packet source and destination. This non-payload data is organized in various headers. An ethernet packet consists of a data sequence to declare the start of a packet (preamble and start frame delimiter, SFD), followed by MAC addresses for the destination and source as well as information about its kind of payload (ethertype). Thereafter follows the payload, and after it, a checksum and finally a data gap of a minimum length before the next packet can start. The mentioned payload is not the true payload (in the sense that true payload would be the actual data to transmit between the nodes), but instead an inner layer data packet (network layer) with its own overhead and payload, defined by the ethertype. This way we can regard the data transmission as layers of overhead information that the true payload is wrapped in. See figure 3 for a schematic depiction of a Transmission Control Protocol (TCP) packet Ethernet frame [6].

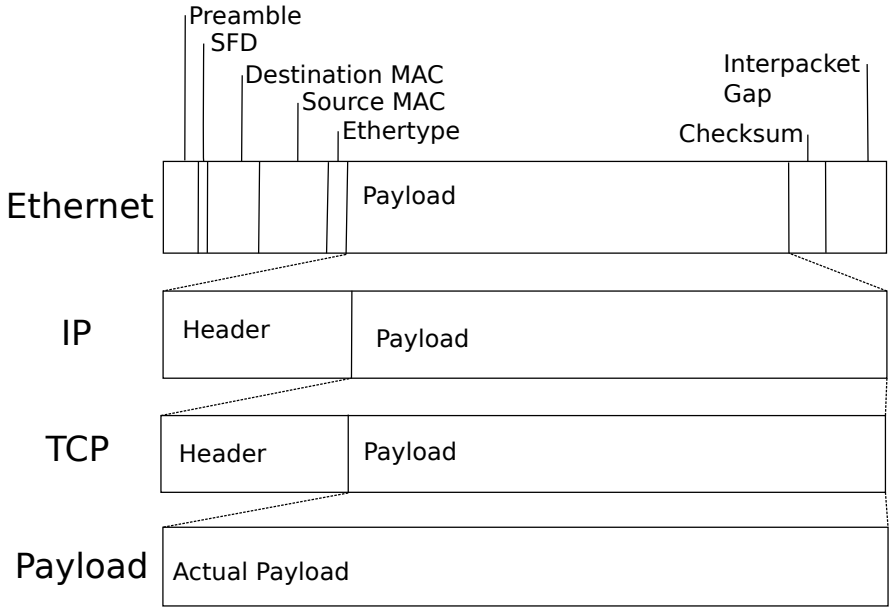

Figure 3: Format of an ethernet frame, containing the preamble, SFD, destination and source MAC addresses, ethertype, payload and a checksum. This frame is followed by a interpacket gap before next packet. The payload of this Ethernet frame makes up the IP frame, with a header and a payload. This payload in turn makes up the TCP frame with its own header and payload. This way, we can regard the actual payload as wrapped in transport, network and link layers, each with their own overhead.

This may seem like a complicated way of transferring data, but it gives flexibility and the handling of various layers can be cleanly separated in several hardware and software layers. Ethernet has been used and developed for a long time, and as a consequence the needed hardware is well-developed, commercially available and cheap. Current equipment can transfer data at rates in the order of hundreds of Gb/s with rates of 1 Tb/s being expected soon [6]. These data transfer rates along with its availability and low price make Ethernet a technology whose potential for nuclear physics experiments has not yet been fully utilised.

2.6 Network Protocols and Layers

In this report, the four outermost layers are of concern. These are the physical layer, the link layer, the network layer and the transport layer.

The physical layer defines how the information is transferred in general, without regard to anything more than how the raw bits are being sent, transported and received. On the FPGA board used in this project (see section 4.1.1), this is implemented by voltage signals in a physical layer chip (PHY chip) that receives and transmits small data words, *nibbles*, of four bits each.

The link layer defines how the packets are transported between network nodes. In this project, Ethernet is used exclusively for this. The payload of the ethernet package is the network layer and its payload is in turn the transport layer. For these two, some different protocols are of interest.

2.6.1 ARP, ICMP and IP

The Address Resolution Protocol (ARP) and Internet Control Message Protocol (ICMP) are two basic network layer protocols. ARP is used to gain knowledge of the MAC address of a unit on a network by sending a request which translates to the question 'who is at this IP address?'; this prompts the unit with that address to answer, replying that it has a certain MAC address.

ICMP is used to send control messages. An important control message is the *echo request* message, that is used to provoke the recipient to send a corresponding *echo reply* message, which measures the round-trip time (RTT)

between the nodes and the loss of packets in the process. A typical RTT within a single 1 Gb/s network segment is in the 100-300 μs range. Since ICMP works with IP addresses, successful operation requires also ARP to be functional, making it useful as a first-level debug tool. To be able to communicate with other units on a network, the FPGA, in this case, must be able to handle these protocols.

The Internet Protocol (IP) is the main protocol for communicating between different networks, making it the foundation of Internet traffic. IP is the most common network layer protocol used in data transfer in combination with the transport layer protocols TCP and UDP.

2.6.2 TCP and UDP

The Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) are transport layer protocols used for data transfer, and are most commonly used together with IP. UDP is an unreliable form of data transfer, being often jokingly referred to as Unreliable Datagram Protocol. The sender sends the packets and does not expect a confirmation if they were received. This can be useful if such a confirmation is not very important. Reliable communication can still be built upon UDP if a user application handles the confirmation and retransmission of packets.

TCP on the other hand aims to be a protocol that guarantees that every byte arrives correctly and in order. It does so by demanding acknowledgements from the recipient. For a large set of data to be sent, it is divided into smaller segments that make up the payload of the TCP packets. These have different sequence numbers that tell how much data has already been sent. The TCP connection starts by a so called three-way handshake, which consists of the two nodes sending their initial sequence number and then acknowledging that they have received the sequence number of the other. After this, the data transfer begins and the sender will send some packets onto the network and wait for acknowledgements for these packets, sending more as the acknowledgements arrive.

There is a limit to how much data the receiver is configured to have directed at it for any given moment which is called the receiver window. This means that the data transfer rate can be capped by this window if it is smaller

than the bandwidth times the time of transfer, called the bandwidth delay product (BDP). This window limit is usually 65 535 octets (8-bit words), but can be made larger by using an option in TCP called *window scaling*. This is especially useful if the network has a large round trip time to allow for larger bandwidths. As an example: at 1 Gb/s (125 MB/s) transfer bandwidth, a 64 kiB receiver window allows a maximum RTT of 524 μs.

3 Scope of Thesis Project

The aim of this work is to implement and test a limited network stack, called Fakernet, on an FPGA and to test it. To do this, code is written to handle the receiving of data from the PHY chip, that is the physical layer, and then process this data to implement a part of the link layer. The data is received as four bit nibbles, which is used to find the start of ethernet frames and forward the incoming packets as 16-bit words to the Fakernet component, which handles everything from there. The packets that Fakernet generates are divided into four bit nibbles and transmitted onto the network through the PHY chip.

To see if the design is fit for use in nuclear physics experiments, it is tested under various conditions. These tests serve to measure under what circumstances the round trip time and data transfer rate are sufficient for this purpose. Furthermore its behavior in a saturated network and the effects of receiver window size are tested.

3.1 Scope of Fakernet

The way an FPGA works puts some constraints on what can be programmed. As mentioned above (see section 2.2), the FPGA has a limited amount of resources and furthermore, the timing constraints of the signals in the FPGA must be fulfilled. With this in consideration it is an objective to write programs so they use as few components as possible on the FPGA, maximizing available resources for other tasks like ADC control and on-board data processing, and to have ample timing headroom (see section 2.2). This is the main reason why a full network stack is not implemented, satisfying all the standards, but rather as little as necessary to make this specific form of communication work. Because of this, the implemented program is not as useful in other circumstances than for sending data in a streaming fashion from e.g. from an ADC to a PC.

3.1.1 Limitations of the TCP Implementation

Not implementing a full network stack, but rather only what is necessary, puts some restrictions on what it can be used for. These are some of the restrictions for the TCP implementation:

- Data is only transmitted, never received (except for acknowledgements). Control communication, to allow for on-line configuration of other aspects of the firmware is performed via UDP.

- The payload is handled in units of 16 bit words instead of 8. This is not a disadvantage in nuclear physics experiments where the building blocks often are 16 or 32 bit words [3].

- No options for IP or TCP are implemented, for example window scaling (see section 2.6.2), which can make the window size a limiting factor of the data transfer rate for large delay distances. The FPGA is to be used on a small network, both in size and in nodes, with only one or two switches between it and the PC.

- The retransmission and bandwidth back-off, that is lowering and finding new transfer rate when the network is saturated, are simplified. This means the FPGA will only work efficiently on networks without limiting bottlenecks, minimizing packet loss.

4 Method

4.1 Tools

These are the hardware, software and languages that were used in development and testing.

4.1.1 Development Board: Arty A7 35T

The development board Arty A7 35T from the company Digilent is a board with an FPGA, various connectors, switches and indicators; see figure 4 for schematics of the relevant components. It has a Xilinx Artix 35T FPGA, with 20 800 LUTs and 41 600 flip-flops, that can be configured by connecting to a PC over Universal Serial Bus (USB) and load a bitstream file. This USB connection also serves as a connector for serial communication using an on-board USB-serial bridge. For simple debugging, it has four switches, four light emitting diodes (LED) and four multicolored LEDs [7]. One key feature for this project is the RJ45 jack with a physical layer chip (PHY chip) of type Texas Instruments DP83848j. This chip serves as the interface between the network and the FPGA, handing over recieved data as four bit nibbles, along with other information such as data validity and clock signal, and sending data as four bit nibbles from the FPGA with at most 100 Mb/s transfer rate [8].

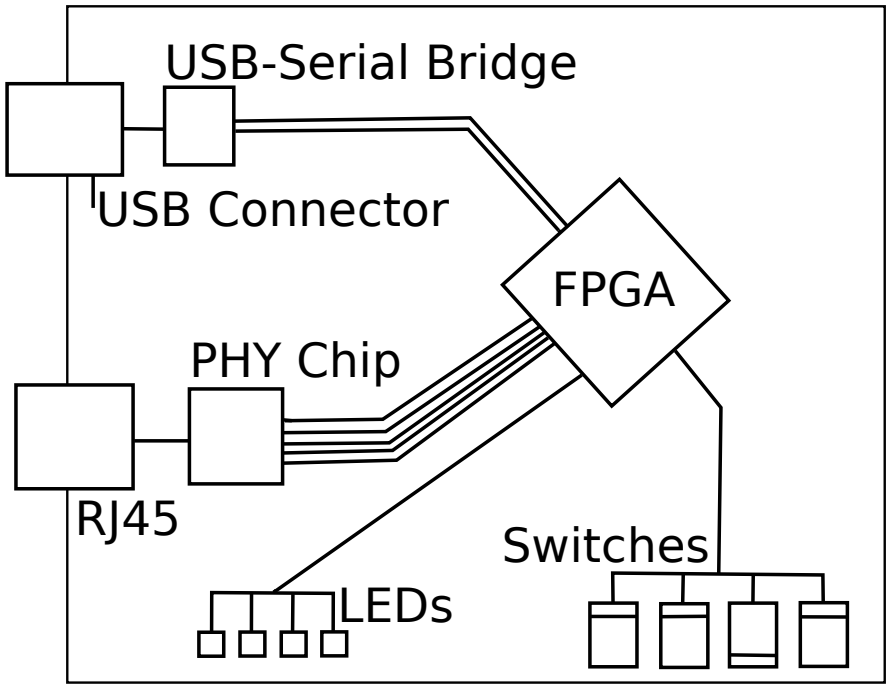

Figure 4: Schematics of the development board used in this project, showing the most relevant parts. The switches are input signals to the FPGA and the LEDs are controlled by output signals from it. The PHY chip and the serial interface are used for both input and output signals.

4.1.2 Network Equipment

All ethernet traffic is transmitted over twisted pair cables of lengths that make the travel time insignificant for the tests in this report (the signal speed is somewhere around 60 to 70 per cent of light speed in the cables, and their lengths are not more than a few meters, meaning the cable transport times are in the order of ns) [3]. The network switches used are Netgear GS105 with 1 Gb/s capabilities as well as store and forward mode, meaning that the packets are fully received before they are sent. The two hubs are Netgear DS104 and Netgear EN104 with 100 Mb/s and 10 Mb/s maximum data rate, respectively. These hubs do not have store and forward mode. Finally, the test PC has a 1 Gb/s interface and the FPGA board has a PHY chip with 100 Mb/s. All units with higher capabilities have the ability to negotiate a lower data rate when needed [9][10][11]. These properties make the switches and

hubs useful for different things. The hubs are used to force units to negotiate a lower transfer rate, and the switches are used for increasing round-trip times, such as they are used in the setups in figure 7.

4.1.3 Language: C

The C programming language is a general purpose programming language that has been in use since 1972 [12]. C is here used to write programs for communication between the FPGA and a PC, using a Universal Asynchronous Receiver/Transmitter (UART) interface. This is a simple serial interface that allows for sending data to the FPGA, which can then answer by sending data back. Furthermore, C-programs are used to interpret the data that is retrieved from the FPGA. The reason for using C for these tasks is simply that it was the language of choice for the author, while at the same time having already-written libraries for certain tasks, such as UART communication.

4.1.4 Language: VHDL

The Very High Speed Integrated Circuit Hardware Descritpion Language (VHDL) is a programming language that has been around since 1983 and is a common language for programming FPGAs. As a hardware description language, its code defines the structure and behavior of an electronic circuit [4][5].

Because of the design of FPGAs, they perform instructions simultaneously, with the logic defined by the program. This means VHDL has to be written with this in mind, making the code very different from code for systems that perform instructions sequentially. The variables used in VHDL are most often logic signals and arrays of these, called logic vectors. In the programs written for this project, some design choices are recurring, such as the *state machine*, the *process*, *counters* and *components*. A state machine gives the program some structure, allowing it to progress between different states, in which different tasks are performed. A process in VHDL is a way of organizing a task by having a trigger (usually a clock signal) that makes the FPGA execute a block of code (still simultaneously). Counters are the main way of keeping track of clock cycles or iterations, increasing the value for each time a certain thing happens, and resetting under certain circumstances; counters

are also the main way of measuring time. Finally, components are a method of abstraction, somewhat similar to functions in usual procedural languages such as C. They have a defined interface, with input and output signals, so that they can be regarded as black boxes, performing some task. To have components work together with the top level design (which is similar to a main function in procedural languages such as C) or each other, common clock signals are usually forwarded between them. See section A for two code examples that demonstrate some of these designs. They are slightly simplified versions of some code used in this project.

To use a design, the VHDL code is synthesized (translated to a hardware design) and compiled into a bitstream file (similar to an executable program), which is loaded onto the FPGA. The reason for using VHDL is that it is a widely used language with many available learning resources, in form of books, websites and demo programs, and that Fakernet (the network stack implmentation) was also being written in VHDL, simultaneously.

4.1.5 Vivado

The software used for synthesizing (see section 2.2) and implementing the VHDL code was Vivado by Xilinx. Vivado translates the VHDL code into a hardware design and then tries to find a way to place that design on the FPGA's resources, given a chosen FPGA. If the design does not use more resources than the FPGA has and there are no issues with timing then the design is compiled int,o a bitstream file. If not, Vivado provides error logs that can be used for debugging. The bitstream file can then be used in Vivado to program a connected FPGA.

4.1.6 Wireshark, Ping and Netcat

To observe ethernet traffic on the PC side, the software Wireshark was used [13]. It allows for observing and analyzing incoming and outgoing packets, thus enabling the user to check if the packets sent from the FPGA arrive, are correct and, in the case of faulty packets, find out what is wrong. This also allows for easily reading debug data that is sent in the payloads.

Ping is a program that sends ICMP packets of the type *echo request*. These are, in case things work, answered by *echo reply* packets from the other end.

This is a method of checking if units are connected to a network and if so, it also measures how long the round trip time is, that is time from sending of an *echo request* to arrival of the *echo reply*. There is an option in ping to change the size of the ICMP packets, allowing for studying the behavior of different packet sizes.

To test the TCP functionality, a program called Netcat (nc) was used. It is designed for reading and writing to network connections using TCP and UDP. In this project it is used in the PC to read TCP packets from the FPGA, also measuring the data transfer rate.

4.1.7 UART Component

A receiver and transmitter component for an asynchronous serial interface (UART) was written in VHDL to communicate between the FPGA and the PC. This was used to read the information saved in the dual ported random access memory (DPRAM) of the FPGA, using the serial bridge of the board. This way, the data could be saved in a text file in the PC for analysis and debugging.

4.1.8 Dual Ported RAM Component

A DPRAM component was written in VHDL with the abilities to change address as well as to read and write data. This was mainly used to record traces of signals, of the PHY chip or to debug signals from Fakernet by saving the signal values for each iteration. This allowed them to be read over a serial interface to a PC for further analysis and debugging.

4.1.9 Fakernet

Fakernet is a software VHDL component written by Håkan T. Johansson. It serves as an interpreter of incoming packets and generator of outgoing packets. The idea is not to implement a full network stack, but rather the minimum required for our purpose, that is transferring data from the FPGA to a PC. Fakernet is able to answer ARP requests and certain ICMP messages, establishing its connection as a node on the network. Furthermore, it implements a limited form of TCP, allowing it to establish a connection to a PC and sending data through TCP packets. A control interface via regis-

ters, setting and retrieving control settings is also implemented using UDP packets.

Fakernet works only on link level and inwards, so to use the component, it takes input in the form of 16-bit words along with a clock signal and a few more signals, such as when a word starts. The output from Fakernet is also 16-bit words that are to be transmitted using the PHY chip. So by using this input and output interface, it functions as a limited network stack. It is in this way Fakernet is used in the design of this project.

4.2 Development and Test Setup

The process of getting Fakernet to work on the FPGA can be split into some substeps, outlining a general plan of the project:

1. Write UART receiver and transmitter components for the FPGA and establish a connection between the PC and the FPGA using this serial interface.
2. Design a DPRAM component on the FPGA and use the serial interface to switch addresses, as well as read and write its contents from the PC.
3. Record traces of signals of the PHY chip into the DPRAM.
4. Dump the raw signals of received packets, decode them and verify that they are valid packets.
5. Generate and analyze predefined packets to more closely study the behavior of the PHY chip.
6. Implement the Fakernet component, with send and receive interfaces.
7. Test Fakernet under different conditions.

4.2.1 General Setup

The main setup for developing and testing the FPGA design consists of two PCs, the FPGA and a network hub or switch. One of the PCs, henceforth called the development PC, is used to write the programs and for communicating with the FPGA using UART. This is connected with a USB cable to

the FPGA, used both to program it and for the serial interface. The FPGA is also connected to a network hub or switch, thereby connecting it to the other PC, called the test PC, and at the same time to the department's local network. This way, the FPGA can communicate with the test PC using Ethernet. See figure 5 for a schematic overview of the general experimental setup.

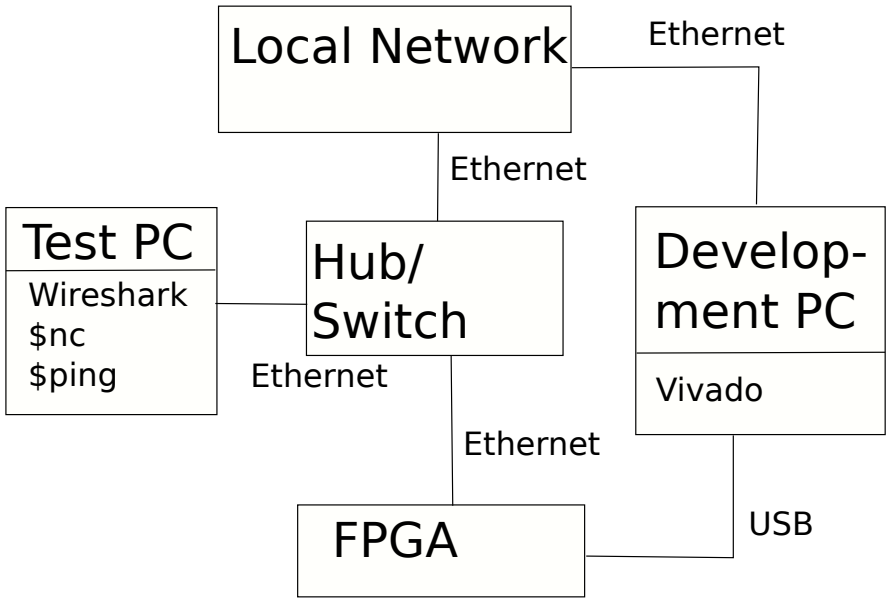

Figure 5: General setup for developing and testing programs for the FPGA. All other setups are, in essence, subsets of this general setup, in that they may be constructed by disconnecting one of the four ethernet connections. During testing, multiple hubs and switches were also connected in series to produce a delay effect.

4.2.2 Setup for writing the dual-ported RAM

The dual-ported RAM (DPRAM) was written in VHDL, together with receiver and transmitter for the serial interface, on the development PC, using vivado to generate a bitstream file. This was loaded onto the FPGA using a USB cable. To write, read and switch memory addresses of the DPRAM, a UART interface was used. Firstly this was done in a primitive way, manually sending and receiving octets in the development PC using the program Screen, but later on, C programs were written to send and receive on the PC

side. At this point, nothing was connected to the RJ45 of the FPGA board.

4.2.3 Setup for testing the PHY chip

To test the PHY chip, broadcast packets anyhow present on the department's local network were used, not caring about the specific contents of the packets. These packets came via a network hub (EN104, 10 Mb/s) that was connected to the local network and thus forwarded to the FPGA board. The PHY chip's *data valid* pin signaled when a valid packet was received, and this was used to trigger a loop in the FPGA, saving the trace of the packets in the DPRAM, with a recording of the PHY chip signals each clock cycle. This trace was later read by the PC using the UART interface and decoded by some C programs that were written for this task.

There is a difference in using this network hub and, for example, one of the switches. The main difference is that this hub has a 10 Mb/s interface while the switch has a 1 Gb/s interface [11][9]. This means using the hub forces the PHY chip to negotiate a slower transmission rate, making this initial investigation of the PHY chip simpler.

4.2.4 Setup for testing Fakernet

The test PC was used to generate packets for the FPGA to answer. These were sent from it to the ethernet switch and then to the FPGA. This communication is bi-directional, allowing the test PC to observe and analyze the packets using Wireshark. In addition, debugging was also performed using traces recorded by the DPRAM with additional debug signals from Fakernet.

Eventually, as TCP was implemented, it was tested by using Netcat on the test PC, reading TCP packets from the FPGA and measuring data transfer rates. At this stage, virtually all debugging was done using Wireshark on the test PC. As the TCP implementation grew more mature, its performance was tested by changing the software and hardware, observing the data transfer rate and round-trip time. By modifying the source code, it is possible to apply restrictions such as payload size of TCP packets or the receive window. Furthermore, the setup can be modified by changing the path between the FPGA and the test PC, for example by adding network switches.

5 Results

5.1 Packet Size Effects on TCP Transfer Rate

To test how the TCP implementation of Fakernet behaves with different packet sizes, its source code was changed to only permit payloads of certain lengths. Note that the overhead is constantly 78 octets, 38 from ethernet overhead and 20 each for TCP and IP overheads. The data transfer rate, R_d, should equal the bandwidth of the connection, R_c, multiplied by the fraction of the payload, D_p and the packet size which is payload plus overhead, D_o:

$$R_\mathrm{d} = \frac{D_\mathrm{p}}{D_\mathrm{p} + D_\mathrm{o}} R_\mathrm{c} \quad . \tag{1}$$

With a constant connection speed (100 Mb/s) and constant overhead (78 octets), the data transfer rate, R_d of equation (1), is a function of one non-negative integer variable, D_p. This function is zero for a zero payload, which is not surprising since there would be no data to send. Furthermore, as D_p approaches infinity, the function approaches R_c. Ethernet frames only permit (assuming *jumbo frames* that allow for larger packets are not implemented) payload lengths of 1500 octets, but the TCP and IP overhead (40 out of the 78 octets) are not included, such that in this case the payload is further reduced to 1460 octets. Using netcat, taking 100 second averages, the data transfer rates were measured. The measurements show good agreement with equation (1) for payload sizes larger than or equal to 6 octets (84 octets total packet length), and very poor agreement for smaller payloads (mean rates of 4 and 2 octets payload were lower than the predicted value by factors 53 and 2422 respectively and they were remeasured ten times each, showing insignificant relative variances of lower than 0.01). See figure 6 for a plot comparing measurements to the predicted values.

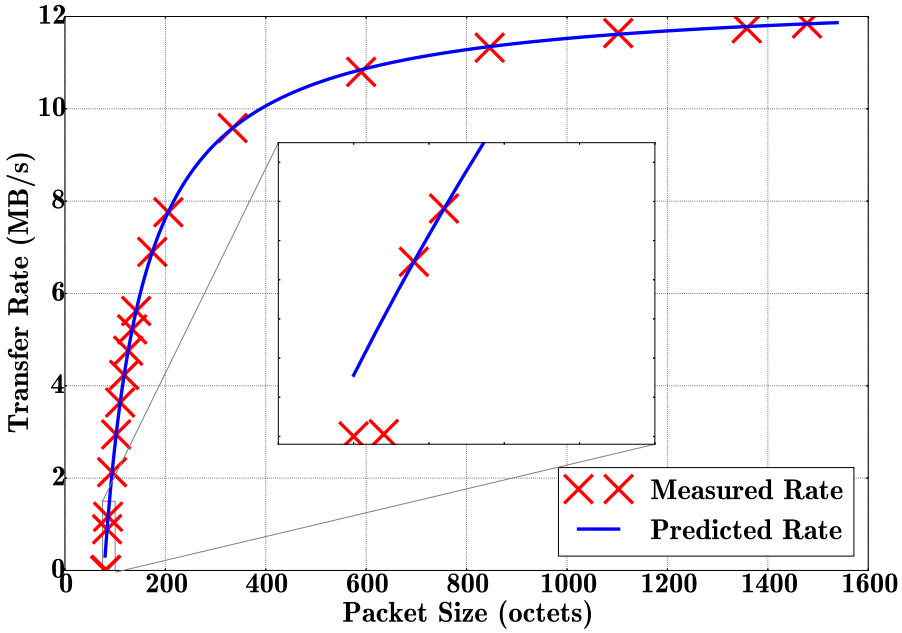

Figure 6: Measured values of data transfer rates (red crosses) and the predicted value (blue curve) plotted against the total size of the packets for a connection speed of 100 Mb/s. The zoomed in inset contains measurements for 2, 4, 6 and 8 octets payload, highlighting the significantly lower transfer rate for payloads smaller than 6 octets compared to the predicted value.

The speeds for very small payloads being much smaller than the predicted values leads to the conclusion that Fakernet fails to communicate these packets at the expected rate.

5.2 Round-Trip Time Measurements Using Ping

Using Ping, it is possible to test the round-trip time (RTT) between the test PC and the FPGA. Furthermore, Ping has the option of varying the size of the packets that are sent in both directions. Tests were performed, measuring the lowest RTT for five different setups, named I through V, using different sizes of packets (see figure 7).

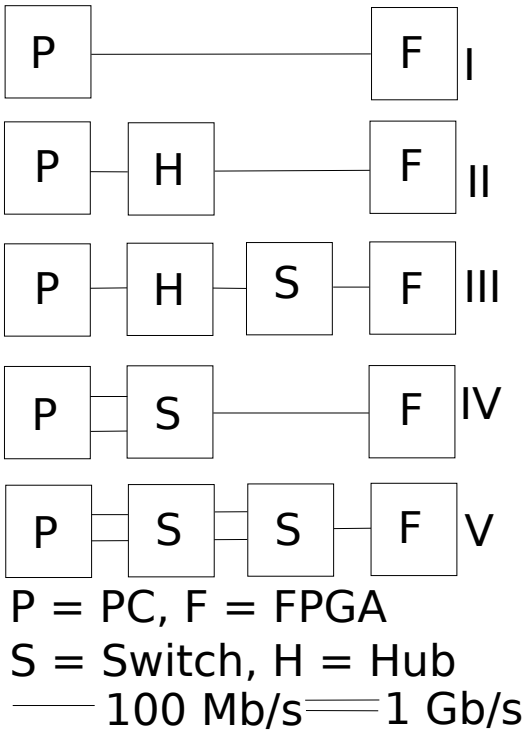

Figure 7: Schematic view of the five different setups for measuring the round-trip time. Note the distinction between connections of 100 Mb/s and 1 Gb/s transfer rate.

This way, the added time to the RTT by changing the setup, for example adding a network switch, was measured for different packet sizes. The switches in the setups are using *store-and-forward* mode, meaning that they receive the entire packet before forwarding it [11]. The FPGA can also be regarded the same way, needing to receive an entire echo request before beginning to send its echo reply. This means that these units add a minimum of $\Delta t = R_c D$, where R_c is the data transfer rate and D is the length of the packet. For a 100 Mb/s interface, this means 0.01 μs/b and 0.001 μs/b for 1 Gb/s. The times added by network hubs are uncertain, since they start forwarding data as soon as they start receiving a packet [10][9]. Furthermore, it should be noted that the RTT also depends on the PC, since the time that is measured is at the application level, meaning the packets have to go through the kernel and not only the network card, adding some time that will always be needed, regardless of the setup. Finally, the cables used

are short enough to not affect the RTT any significant amount (1m of cable amounts to a delay of ≈ 5 ns and the total cable length was never larger than a few meters, see section 4.1.2).

With these assumptions, the round trip time, T_{RT}, is an affine function that depends on a constant, C_0 (with dimension time), and a coefficient, C_1 (with dimension time per data), multiplied by the packet size, D:

$$T_{\mathrm{RT}} = C_0 + C_1 D \quad . \tag{2}$$

By measuring this for different values of D, a system of linear equations is produced. From these, one can fit linear curves. Writing this in matrix notation, $Ax = b$, it is solved with minimal error using the method of least squares: [14]

$$A^T A x = A^T b \leftrightarrow x = (A^T A)^{-1} A^T b \quad . \tag{3}$$

Superscript T denoting transpose of matrix ($A^T_{ij} = A_{ji}$). Testing these setups for some packet sizes and using this method, the measurements resulted in affine functions, with a slope that corresponds to time added per bit and their value at $D = 0$ corresponding to the time constant from equation (2) (see figure 8 for these measurements and fits for three different setups).

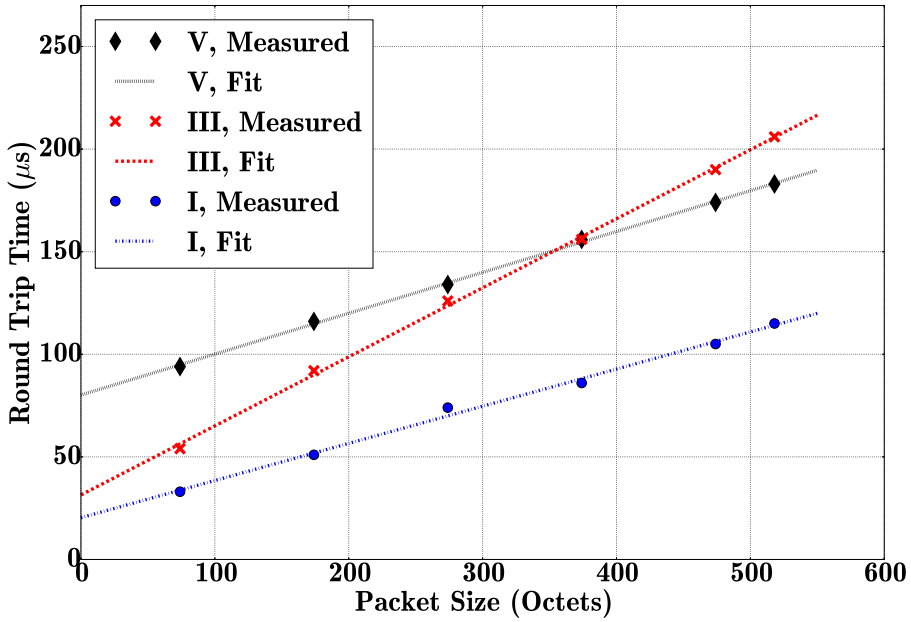

Figure 8: Measured RTTs for three different setups for different packet sizes (markers) and the corresponding fits (non solid lines). See figure 7 for schematics of the setups used.

See table 1 for a comparison of the measured time addition and the predicted minimum between the setups. Comparing these slopes to the predicted minima of time added per bit of the five setups, it can be concluded that none of the measurements showed a smaller time increase than expected and that every fit's slope was close to the minimum time that the network switches should add.

5.3 Effect of Window Size Restrictions

Variation of the TCP receive window changes how much data can be sent before acknowledgement, i.e. be in-flight on the network at the same time. If this value is smaller than the BDP, it will be a bottleneck, limiting the data transfer rate because the sender would be able to send data but is not allowed to do so. By limiting the receive window in the Fakernet source code, it is possible to test this effect (it would be possible, but more complicated

Table 1: Measured fit and predicted minimum time added per bit for different setups. See figure 7 for schematic view of the setups.

Setup	**Measured,** C_1	**Predicted minimum**
	ns/b	**ns/b**
I	11.317	10
II	11.254	10
III	21.035	20
IV	12.157	11
V	12.449	12

to change window size on the test PC side). We have for a window size, W, a bandwidth, R, with maximum, $R_{\max}$, for an otherwise unconstricted connection and a RTT, T_{RT}:

$$R = \begin{cases} R_{\max} & \text{if } W \geq B_{\max} T_{\mathrm{RT}} \\ \frac{W}{T_{\mathrm{RT}}} & \text{if } W < B_{\max} T_{\mathrm{RT}} \end{cases} . \tag{4}$$

For a constant packet size, the RTT is constant and $B_{\max}$ is constant and known. This means that by measuring the bandwidth decrease with decreasing window size, it is possible to find at what window size the bandwidth, R is limited, i.e. the point $W = B_{\max} T$. Finding this window size yields the round-trip time. This was measured for four different packet sizes for a TCP stream, using Netcat to measure bandwidth (correcting data transfer rate for overhead). The setup was identical to setup III from the Ping RTT measurements (see figure 7 for schematic view of the setup). To approximately find the point $W = B_{\max} T$, the method used was to take the window size for the measurement where the bandwidth had decreased to 95 per cent of its maximum (see figure 9).

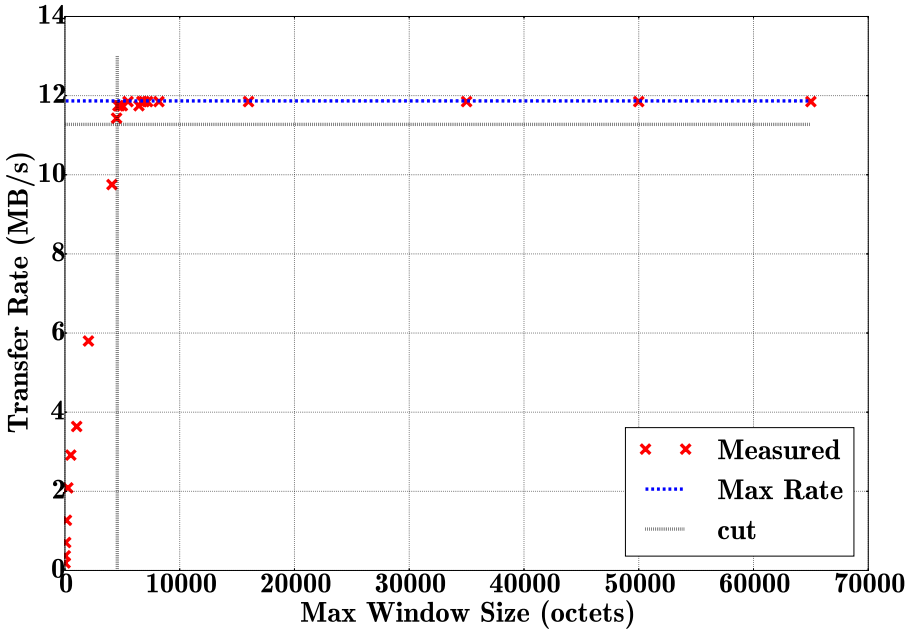

Figure 9: Data transfer rates (directly proportional to bandwidth) measurements (red crosses) for constant packet size (1538 octets and 64 octets acknowledgment) varying the maximally allowed receive window size along with the maximum data transfer rate (blue dashed line, calculated from known bandwidth and overhead-to-data ratio) and the cut (black dotted line) to find $W = B_{\max}T$.

For these measurements, it can be concluded that the RTTs are so small that the window scaling option is not needed for full usage of bandwidth in TCP communication within this small network. To test the validity of this method of measuring the RTT, it can be compared to the RTTs that were measured using ping for the same setup. See figure 10.

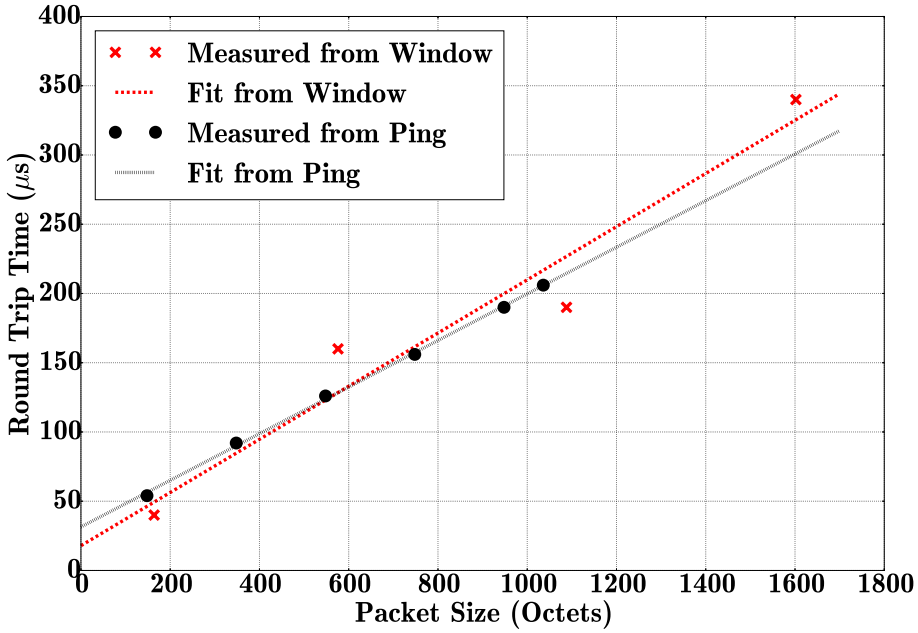

Figure 10: Round trip times determined from receiver window constraints (red crosses) for different packet sizes along with a least squares fit to a line (red dashed line) compared to the Ping RTT measurements (black circles) with their fitted line (black dotted line) for the same setup.

5.4 Bandwidth Decrease in a Saturated Network

Because of the way retransmissions are implemented in Fakernet, data transfer over TCP is more limited when facing a saturated situation on a network than a standard TCP implementation. Standatd TCP implementations tend to stabilize their transfer rates such that they share the bandwidth roughly equally. This is optimal in the sense that it maximizes the lowest transfer rate on the network. This situation was tested by using netcat, having a PC on the local network send one or more streams of TCP, with standard implementation, at the same time as the FPGA transmits data to the test PC, where the transfer rate was measured. To create a bottleneck, the network card of the PC was set to only negotiate 100 Mb/s with the network switch that all this data went through. By measuring the data transfer rate, the percentage of the maximum data transfer rate was calculated. These measurements

show that Fakernet is severely limited when faced with a bottleneck. Fakernet behaves a little bit better on a saturated network if it competes with other identical boards that also run fakernet. This was measured in a similar fashion, connecting two and three boards to the same switch and sending data to the PC. See figure 11. All these measurements were conducted ten times each, measuring an average over 100 seconds. None of the variances were larger than one per cent of the maximum bandwidth.

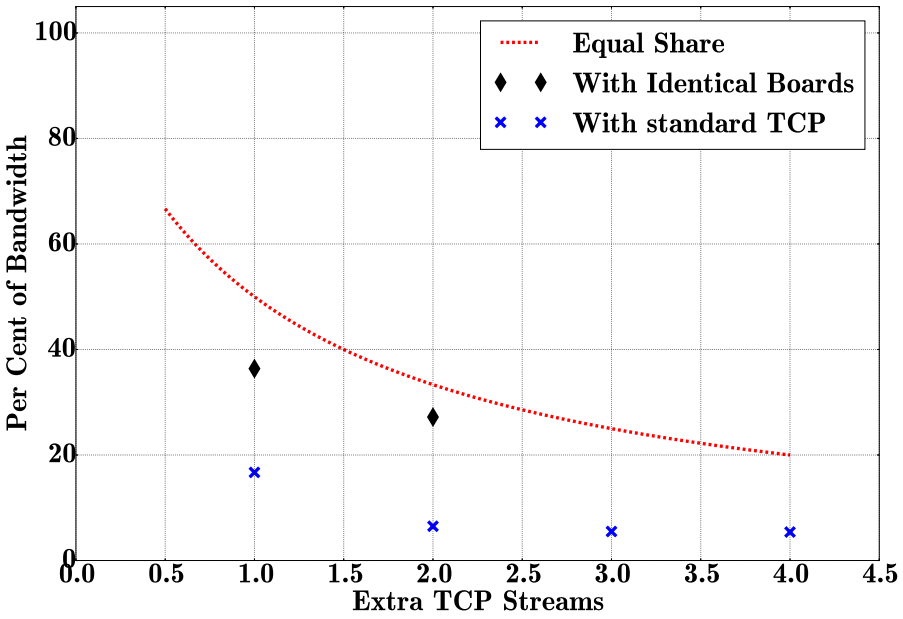

Figure 11: The measured bandwidth percentage of the TCP traffic from an FPGA board with the Fakernet implementation. In the case of different number of extra TCP streams from PCs (standard TCP implementations), it is shown in blue crosses. The black diamonds are for the measurement with other, identically configured boards running Fakernet. Also depicted is the equal share (that is the function $\frac{100}{1+x}$, red dashed curve) that represents the value if the FPGA had shared the bandwidth equally with the other TCP streams.

6 Discussion

6.1 Window Scaling for a Gigabit Interface

One point of interest is how this design would behave with a 1 Gb/s interface instead of one of 100 Mb/s. The design does not implement TCP options and therefore there is no window scaling. This means the maximum allowed receive window is 64 kiB. Remembering equation (4), the maximum allowed RTT for a bandwidth of 1 Gb/s is 524 μs if the window size is not to be a limiting factor. The question that should be asked is under what circumstances this would be achievable.

Measuring the RTT between two computers on the department's local network, connected with 1 Gb/s bandwidth and with one network switch between, Ping reports a time of around 130 μs. Testing also for a longer distance, pinging www.chalmers.se, multiple switches and routers away, gives a time of around 290 μs. This means that a receive window size of 64 kiB will not limit the data transfer rate in these situations. Comparing this to a reasonable setup in which this design would be used, consisting of a small network with only one or two network switches between the FPGA and the PC [3], it can be concluded that the window scaling option is not needed. Therefore, Fakernet's inability to handle window scaling is not a disadvantage, justifying the choice of not implementing it.

6.2 Comparison With Situation at R^3B

The experimental setup at R^3B at GSI in Germany is what originally sparked interest in the idea of transferring the ADC data directly over Ethernet. Upstream from the experimental setup, a stable beam hits the production target of the GSI fragment separator (FRS), producing different nuclides. These are separated in a set of four dipole magnets, causing only certain nuclides to pass, before reaching the R^3B setup. Recent experiments at this setup have an average data output in the order of a few 10 of MB/s, recording a few kHz of events [3]. Implementing the design of this project would mean to use it on around 300 boards [2], each capable of more than 10 MB/s (using only 100 Mb/s). This results in a capability of about 3 GB/s, well above the requirement for average bandwidth, assuming an equal data rate from each front-end board. If, instead of 100 Mb/s, boards with 1 Gb/s interface are

used, the total capability is higher by a factor ten. Using gigabit ethernet requires a more careful design of the VHDL interface to the PHY chip, but the internal processing speeds required are already fully supported in the Fakernet component.

A more interesting comparison would be to the detectors with the highest output rates. These are in the central focal plane of the FRS, before the magnets have separated most ions. If we assume an event rate at these detectors of around 100 MHz during experiment and that every event produces ten bytes (including some overhead) from the ADC, it would require a bandwidth of 1000 MB/s, above the capabilities of this design. There is, however, a solution for this, in that most of these events are uninteresting since they represent nuclides that will not fully pass through the FRS magnets. If a detector after the magnet (which has fewer events, in the order of 100 kHz [3]) detects a hit, it can send a trigger signal to the earlier detector, validating hits in certain time periods, while the rest is discarded. That way, the amount of data to send can be reduced significantly, allowing for this design to be used with one 1 Gb/s interface. Note that detection rates of 100 MHz would require the involved detectors to be segmented, such that the rate is spread over many strips. This would likely imply the use of multiple front-end boards, so that even 100 Mb/s interfaces may suffice.

An alternative trigger scheme, the free-running method of data acquisition, is used by the CALIFA detector at R^3B. This means that the electronics is not put in dead time for each accepted trigger, allowing for more statistics to possibly be collected in a shorter time, thus making better use of beam time. This is in contrast to the currently used trigger scheme that forces the electronics into dead time during conversion each single event. When the PC has received the data of the previous event, the next trigger can be accepted. One event can be converted during readout of previous event, allowing some parallelism but the method still means less efficient use of beam time. Formerly trigger-based schemes were used due to their more efficient use of bandwidth and disk space, but using current network technology, that is not a concern in the experiments of today. Triggered single-event readout requires the readout to have low latencies, for fast dead time release. Fakernet is not suited for this because of the relatively long RTTs. It is, however, well suited for the streaming nature of the data transfer used in free-running schemes.

6.3 Comparison With Other FPGA Network Stack Implementations

It has been mentioned several times in this report that the network stack should be limited so that it saves resources for other tasks on the FPGA. It can be compared to other designs that have been implemented, which are more complete network stacks with functionality that allows for more TCP connections, more standard implemented bandwidth back-off and retransmission, and TCP options such as window scaling. While these designs have much more functionality than Fakernet, the question is at what cost? Compared to more complete network stack implementations in FPGAs, Fakernet uses less FPGA resources. Table 2 compares Fakernet with several other finished designs that all would have the capability to be used instead of Fakernet. It illustrates the considerable amounts of FPGA resources which can be saved by not implementing a full network stack but rather one that is limited to what is necessary for the purpose.

Table 2: Comparison of different designs that implement networks stacks with TCP functionality, showing their FPGA resource utilization and speed and supported number of connections. ♯The resource usage is for the full design, the Fakernet component itself only occupies 1 614 LUTs, 944 flip-flops and 12 BRAM units. *Fakernet is only tested for 100 Mb/s but the code allows for 1 Gb/s. †Data is not provided in the respective references.

Design	**LUTs**	**Flip-flops**	**BRAM**	**Speed**	**Connections**
	Units	**Units**	**Units**	**Mb/s**	
Fakernet♯	1 966	1 241	13	1000*	1
Alt 1 [15]	67 938	83 829	323	2 000	10 000
Alt 2 [16]	≈16 300	≈16 300	134	800	>3
Alt 3 [17]	22 468	24 744	353	10 000	10 000
Alt 4 [18]	–†	–†	–†	425	1
Alt 5 [19]	–†	–†	–†	25 000	4096

6.4 Outlook

Ethernet is a promising technology for several reasons (see section 1). The implementation and testing of a limited network stack on an FPGA shows

that it is possible to achieve a functioning unit that is capable of connecting to other nodes on a network and transfer data using TCP. Furthermore, the results show that it, despite its limitations (see section 3.1), manages to transfer data at expected rates and that it also handles retransmission of packets in a way that makes sending data to a PC work, even under saturated conditions. Finally, testing reveals that the RTT for a small network is sufficiently low for BDP to not exceed the 64 kiB window limit, even at an increased bandwidth of 1 Gb/s. With these results in mind, it is reasonable to believe that this implementation could be successfully used in a large-scale experimental setup such as R^3B.

An experimental setup that uses this design would most likely consist of many FPGAs that are programmed to control ADCs and do on-board data processing, as well as having a Fakernet implementation. On the particular FPGA used in this project, (see section 4.1.1) the Fakernet implementation uses less than a fourth of any of the available components (lookup tables (9.45 %), flip-flops (2.98), block RAM (24 %), global buffers (3 %) and phase-locked loops (20 %)), leaving plenty of room for the other tasks. Comparing it instead to the kind of FPGAs used in R^3B, ECP3-150, these FPGAs have more than seven times as many LUTs, so the design leaves even more room. These FPGAs would be connected to a PC with one or two network switches in between, allowing them to send data from the experiment to the PC using TCP over Ethernet.

Fakernet is under development, with bugs that need fixing, see next section. Håkan T. Johansson wrote the Fakernet code at the same time as this project, and at the same department. The version used for testing has continuously been updated. Even though it is not yet entirely bug free, the current version of Fakernet has the functionality to act as the desired network stack.

Viewing the design of this project as a form of proof of concept, its feasability as a replacement for the methods of today is proven. It will be interesting to see the first results from its use in actual experiments.

6.4.1 Further Development

There are several ways of improving and testing this design that implements Fakernet that have not been done:

- The design has never been tested in a situation where there is actual data from a real ADC to be sent to the PC. It is not clear if this would pose any additional difficulties.
- Fakernet has a problem with sending very small packets (see section 5.1), severely limiting its data transfer rate. This is not that big of an issue because packets that small are rare.
- Fakernet requires the FPGA to be reprogrammed every time it needs to open a new TCP connection, that is having a bitstream file loaded onto it. The reprogramming would preferably be done by a remote user, using a control register over UDP.
- The control register, intended for configuring Fakernet, is untested.
- Retransmission of packets is done in a way that makes packet losses due to, for example, bottlenecks cause more slowing-down than needed. See section 5.4.
- The design is not tested at all for 1 Gb/s bandwidth.

6.5 Conclusion

This project started out with a goal of implementing a limited network stack on an FPGA in order to communicate over Ethernet, to eventually transfer data to a PC using TCP. The purpose for that is to replace current systems for data transfer from ADCs to PCs in nuclear physics experiments, using the advantages of Ethernet.

In the end there is a completed design that implements this network stack. It uses an FPGA together with an on-board PHY chip to interpret incoming Ethernet packets, finding their start and building 16-bit words of the data, the incoming network traffic is forwarded to the Fakernet component. This component functions as a network stack that handles necessary protocols to establish connections and transfer data. When Fakernet has generated a packet, be it as an *echo reply* or a TCP packet, its output is sent through the PHY chip onto the network. Thus the original goal has been achieved.

The results from testing this design shows that it has capabilities that make it suitable as a replacement for the data transport systems of today at real

nuclear physics experimental setups. The TCP implementation works and utilizes the entire bandwidth, the round trip time is low on the kinds of networks it is intended to be used on, the design uses few of the FPGA components and there are no issues with timing. Considering the R^3B setup, this design has the necessary capabilities to be implemented.

References

[1] A. Heinz, private communication, Feb. 2019.

[2] Alexandre Charpy, Andreas Heinz, Håkan T. Johansson et al., "Technical Report for the Design of the NUSTAR Data AcQuisition System," 2018, https://edms.cern.ch/ui/file/2024803/1/TDR_NUSTAR_DAQ_public.pdf.

[3] H. T. Johansson, private communication, Feb. 2019.

[4] A. Rushton, *VHDL for Logic Synthesis*, Third ed. Wiley.

[5] S. Sjöholm and L. Lindh, *VHDL för konstruktion*, Fifth ed. Studentlitteratur.

[6] IEEE Computer Society, "IEEE standard for ethernet," 2015, https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7428776.

[7] *Reference manual*, Arty A7 35T, https://reference.digilentinc.com/_media/reference/programmable-logic/arty/arty_rm.pdf, Digilent, Jun. 2017.

[8] *Pysical layer chip datasheet*, dp83848j, http://www.ti.com/lit/ds/symlink/dp83848c.pdf, Texas Instruments, Mar. 2015.

[9] *Network hub manual*, EN104, http://www.downloads.netgear.com/files/en104ig.pdf, Netgear, 2000.

[10] *Network hub manual*, DS104, http://www.downloads.netgear.com/files/GDC/DS116/DS100series_datasheet.pdf, Netgear, 2000.

[11] *Network switch datasheet*, GS105, https://www.netgear.com/images/datasheet/switches/GS105v5_GS108v4_GS116v2.pdf, Netgear, Jul. 2017.

[12] B. Kernighan and D. Ritchie, *The C Programming Language*, Second ed. Prentice Hall.

[13] https://www.wireshark.org/, Feb. 2019.

[14] I. Gustafsson and K. Holmåker, *Linjär algebra och numerisk analys*. 2013.

[15] David Sidler, Gustavo Alonso, "Scalable 10 G TCP/IP Stack Architecture for Reconfigurable Hardware," 2015, http://www.fccm.org/past/2015/pdfs/M2_P1.pdf.

[16] Shen-Ming Chung, Chun-Yi Li, Hsiao-Hui Lee, Jeng-Han Li, Yau-Chung Tsai, Chi-Chun, "Design and implementation of the high speed TCP/IP Offload Engine," 2007, https://ieeexplore.ieee.org/abstract/document/4392084/metrics.

[17] David Sidler, Zsolt Istvan, Gustavo Alonso, "Low-Latency TCP/IP Stack for Data Center Applications," 2016, http://davidsidler.ch/files/fpl16-lowlatencytcpip.pdf.

[18] Satish Narayanaswamy, "High Performance TCP/IP on Xilinx FPGA Devices Using the Treck Embedded TCP/IP Stack," 2004, https://www.xilinx.com/support/documentation/application_notes/xapp546.pdf.

[19] Enyx, "40G/25G/10G/1G TCP/IP + MAC IP Cores for FPGAs and SoCs," 2018, http://www.enyx.com/files/Enyx%20FPGA%20IP%20Cores%20-%20TCP%20UDP%20MAC.pdf.

A Code Examples

These code examples are meant to show the general idea of writing programs in VHDL. One thing to keep in mind is that the code is executed simultaneously. Sometimes it seems as though signals are assigned two times simultaneously, which is not possible, but that is because VHDL treat *if* statements differently. A signal may be assigned in a process (giving it a default value), then reassigned in a subsequent *if* statement without ambiguity. Note also the use of buffered variables. These are used to compare a signal to its value the previous cycle by each cycle setting it to the new value. Comments are written with double dashes, '−−', and assignments are done with arrows, '<=', instead of equality signs, '=', as in C. Logic signals, such as those used in these examples, have two values, '1' and '0', also referred to as high and low or true and false, respectively. These are sometimes grouped in arrays called logic vectors, commonly used to represent integers in a binary representation. Both sections of code are used in the final design of the project. They are slightly changed to make them more readable in isolation from the rest of the program, and commented to make the intentions of each step clear.

A.1 Process Example

This code example shows a VHDL process that saves a trace of Fakernet output (16-bit words) into a DPRAM. To begin with, the interface for the DPRAM component has input signals 'data', 'write_to_mem' and 'address'. These represent the data to be written, a signal to the memory, saying it is time to write, and an address to write to. The information to write is in the interface for Fakernet component, 'word_ready' and 'word', which represent if an output word is ready and its contents, respectively. Using these signals, the process checks if a word is ready, and if that is the case, it writes it to memory and changes the address for the next word. In addition, we need to check if the memory is full.

```
1
2 -- Code Example: Trace Saving Process
3
4 -- Save the output of Fakernet into DPRAMs
5 -- Fakernet gives a signal, 'word ready', when it is
```

```
6  -- ready to send a word
7  -- called 'word'.
8  -- This should trigger the memory to write it
9  -- and advance the address.
10
11 -- To write to memory, the 'write_to_mem' signal
12 -- needs to be high.
13 -- It needs data to be written in the signal 'data'.
14 -- It needs to have the signal 'address' to know where
15 -- in memory to write.
16 -- These three are inputs to the DPRAM component.
17
18 -- signals:
19 -- 16 bit address represented as a logic vector
20 -- initialized as zeros:
21 signal address_counter : std_logic_vector(15 downto 0)
      := (others => '0');
22 -- constant logic vector that represent the max address
23 -- of the DPRAM:
24 constant signal max_address : std_logic_vector(15
      downto 0) := (others => '1');
25 -- signal that represents if the memory is full
26 -- initialized as zero:
27 signal memory_full : std_logic := '0';
28
29 trace_saving_process : process (CLK)
30 -- trigger process each time the clock signal changes
31 begin
32   -- only execute if clock signal went from low to high
33   if (rising_edge(CLK)) then
34     -- in general, we dont want to write to memory
35     write_to_mem <= '0';
36     -- set write address to our counter value
37     address <= address_counter;
38     -- We want to write and advance address counter iff
39     -- we have word ready
40     -- and the memory is not yet full.
41     if (word_ready = '1' and memory_full /= '1') then
```

```
42        if (address_counter < max_address) then
43        -- advance the address counter if address
44        -- is not max yet
45          address_counter <= address_counter + 1;
46        else
47        -- otherwise, set memory filled signal
48          memory_full <= '1';
49        end if;
50        -- set write data to Fakernet output
51        data <= word;
52        -- set the memory to write
53        write_to_mem <= '1';
54      end if;
55    end if;
56  end process;
```

A.2 Component Example

This code example shows how a whole component can look. This component takes input from the top level component (like a main function in C) in the form of a clock signal as well as a signal that represents the clock of the PHY chip that runs at 25 MHz (0.25 times the speed of the board clock). It also takes input in form of data from the PHY chip that represents the nibble and the data valid pin. These inputs are used to interpret incoming data, find the start of a packet (the SFD) and from there build 16-bit words to later be used in the Fakernet component. These words are used as output, along with ancillary information: when the packet starts, ends and when a new valid word is ready. Note the use of keywords *in* and *out* in the interface port map of the component, signaling if the signals are inputs or outputs.

```
1 -- Code example
2 -- Word building component used for building the
3 -- 16 bit words as well as finding the packet start
4 -- for use in Fakernet implementation.
5
6
7 entity word_builder is
```

```
  -- define the interface port map
  port (
    -- clock from board
    clk : in std_logic;
    -- rising edge of the PHY chip clock which
    -- is slower than the board by a factor 4
    rx_rising_edge : in std_logic;
    -- data valid pin from PHY chip
    valid : in std_logic;
    -- data nibble from PHY chip
    data : in std_logic_vector(3 downto 0);

    -- 16 bit word output
    word : out std_logic_vector(15 downto 0);
    -- signal that is high if word is ready
    word_ready : out std_logic;
    -- high if first word in packet
    packet_start : out std_logic;
    -- high for one cycle if packet ended
    packet_end : out std_logic;
    );
end word_builder;

architecture behavioral of word_builder is
  -- here the code that is run is written

  -- variables
  -- the start frame delimiter
  constant signal SFD : std_logic_vector(15 downto 0)
     := "0101010101011101";
  -- signal telling if start is found
  signal start_found : std_logic := '0';
  -- buffered signal from last cycle
  signal last_start_found : std_logic;
  -- used to keep track of which nibble we
  -- are at, initialized at 1
  signal word_counter : std_logic_vector(1 downto 0) :=
      "01";
```

```
44    -- maximum counter value is 4
45    constant signal word_counter_max : std_logic_vector(1
          downto 0) := "11";
46
47    -- buffered value for comparison to last cycle
48    signal last_valid : std_logic;
49
50  begin
51
52    word_building_process : process (clk)
53      -- process to build 16 bit words of 4 bit nibbles
54      -- and find start of frame
55    begin
56      if (rising_edge(clk)) then
57        -- default values
58        word_ready <= '0';
59        packet_start <= '0';
60        -- buffer variable
61        last_start_found <= start_found;
62
63        -- first we find the start
64        if (valid = '1' and rx_rising_edge = '1') then
65          -- this advances the current 16 bit words
66          -- and takes away the oldest nibble and
67          -- appends the new nibble
68          word <= word(11 downto 0) & data;
69          if (word = SFD) then
70            -- SFD found
71            start_found <= '1';
72            if (last_start_found = '0') then
73              -- make sure it is not an SFD
74              -- inside a packet
75              packet_start <= '1';
76            end if;
77          end if;
78
79          -- next we want to advance word counter
80          if (start_found = '1') then
```

```
81            if (word_counter = word_counter_max) then
82              -- a word is finished
83              -- reset counter
84              word_counter <= (others => '0');
85              -- we have a word ready
86              word_ready <= '1';
87            else
88              -- advance counter
89              word_counter <= word_counter + 1;
90            end if;
91          else
92            -- start not found, reset counter to initial
93            -- value, 1
94            word_counter <= "01";
95          end if;
96        elsif (valid /= '1') then
97          -- invalid data, reset all by saying start
98          -- not found
99          start_found <= '0';
100       end if;
101     end if;
102   end process;
103
104   end_finding_process : process (clk)
105     -- process to find end of packet
106   begin
107     if (rising_edge(clk)) then
108       -- default value
109       packet_end <= '0';
110       -- buffer variable
111       last_valid <= valid;
112       if (last_valid /= valid and valid = '0') then
113         -- we have data invalid and it was
114         -- valid the cycle before meaning
115         -- we have found the end of a packet
116         packet_end <= '1';
117       end if;
118     end if;
```

```
119     end process;
120 
121 end behavioral;
```