DIPLOMA THESIS

Optimised Data Handling Strategy for High Throughput

Submitted at the Faculty of Electrical Engineering and Information Technology, Vienna University of Technology
in partial fulfillment of the requirements for the degree of
Diplom-Ingenieur (equals Master of Sciences)

under supervision of

Ao.-Prof. Dl. Dr. Thilo Sauter

Institute of Computer Technology (E384)
Technischen Universität Wien by

Simon Schwingenschuh
Matr.Nr. 1126174
Theresiengasse 30, 1180 Wien

Vienna, June 2018

Kurzfassung

Die steigende Datenübertragungsgeschwindigkeit heutiger und zukünftiger Kommunikationsnetze erfordert Optimierungen von Datenverarbeitungsstrategien für einen leistungsfähigen Datenaustausch. Es wird eine hohe Auslastung des Peripheriebussystems und die Reduzierung der Prozessorzeit für die Datenverarbeitung gefordert. Diese Masterarbeit beschreibt notwendige Komponenten zur Übertragung von Daten zwischen einem Hostsystem und Peripheriegeräten und konzentriert sich auf die Datenverarbeitungsstrategien von Netzwerkadaptern. Zwei Datenverarbeitungsstrategien werden analysiert und ein optimiertes Konzept erstellt und implementiert. Messungen zeigen die Reduzierung der CPU-Verarbeitungszeit und die Verbesserung der Übertragungsbandbreite einer Netzwerkkarte.

Abstract

The increasing data link speed of today's and coming communication networks demands on optimisations of data handling strategies for high-performance data exchange. High utilisation of the peripheral bus system and the reduction of CPU time used for data handling are requested. This diploma thesis describes needed components for transferring data between a host system and peripheral components and focuses on the data handling concept of network interface controllers. Two data handling strategies are analysed and an optimized concept is created and implemented. Measurements prove the reduction of CPU processing time and the improvement of the bandwidth of a network interface controller.

Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Wien, Juni 2018

Table of Contents

Abbreviations

AHB	Advanced High-Performance Bus
ASIC	Application Specific Integrated Circuit
Avalon-MM	Avalon Memory Mapped
Avalon-ST	Avalon Streaming
AXI	Advanced eXtensible Interface
BD	Buffer Descriptor
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
DRAM	Dynamic Random Access Memory
DMA	Direct Memory Access
EDA	Electronic Design Automation
EOF	End Of packet
FIFO	First-In-First-Out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
GB	Gigabyte
GB/s	Gigabyte Per Second
GMII	Gigabit Media Independent Interface
GP-GPU	General Purpose - Graphic Processing Unit
GPU	Graphic Processing Unit
HDL	Hardware Description Level
IC	Integrated Circuit
I/O	Input/ Output
IEEE	Institute of Electrical and Electronics Engineers
IP	Intellectual Property
MAC	Media Access Control
Mbps	Megabit Per Second
MII	Media Independent Interface
MM	Memory Mapped
NoC	Network on Chip
NIC	Network Interface Card
MB	Megabyte
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PCIe	PCI Express
PE	Processing Element
PHY	Physical layer
PLD	Programmable Logic Device
PS	Processing System
PTP	Precision Time Protocol

RTL	Register Transfer Level
RX	Receive
SI	Sink
SoC	System on Chip
SOP	Start Of Packet
SR	Source
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol/ Internet Protocol
TX	Transmit
UART	Universal Asynchronous Receiver Transmitter
UDP	User Datagram Protocol
USB	Universal Serial Bus
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

1 Introduction

Data transfer between peripheral components highly influences the performance of computer systems. Processing units always need enough data for processing their tasks. The performance of the memory, the central processing unit and the bus system are defining the processing speed of a computer system. Additionally, other peripheral components, as for example graphic cards, network cards or hardware accelerators, are used to add special functionality to a system or speed it up.

The demand of optimising computational intensive or time critical functions available on platforms with reduced computational resources opens a growing area for specialised peripheral components. The peripheral unit moves data processing from the Central Processing Unit (CPU) to specialised hardware components. Optimisation in terms of processing time and power consumption is also a demand for these hardware components.

Moving computational intensive functions like floating point calculations from the CPU to the General Purpose-Graphic Processing Unit (GP-GPU) adds complexity to the development but improves the performance greatly. Initially, GPUs were used to calculate the graphical outputs. In the last decades, the GPUs were upgraded to be usable in general-purpose computing [VN14].

The highest complexity in developing, but with a great boost in computational power, power efficiency and configurability, is reached, if Field Programmable Gate Arrays (FPGA) are used. While GP-GPUs have fixed computational modules, FPGAs are programmable digital logic chips, which can be fully configured to achieve any goal. The high configurability gives the developer a great freedom in developing solutions, but also increases the development time and complexity. FPGAs are not intentionally built for accelerating CPUs. They were built to get reconfigurable hardware chips for replacing digital circuits. They are also a cheap alternative to Application Specific Integrated Circuits (ASICs), which are costly in development and fabrication [Rei09].

Connecting devices in a network enables the exchange of data for different areas of applications. The highly increasing amount of application data and the usable link speed of communication networks is the reason for demanding improvements in data handling strategies. The utilisation of resources available on computer systems and the efficiency of communication channels between components have to be increased.

1.1 Problem Description

Communication channels in present-day computer systems must be prepared for transferring a huge amount of data. The network speed of up to 100 gigabits per seconds (Gbps) is leading to a problem for computer systems. Data handling methods have to be optimised to fully utilize the link speed of network interfaces.

The challenging thing is to serve enough data to take advantage of the high data rate of communication interfaces. However, the processor should not be highly involved in preparing and transmitting data from the memory to the network interface - other components should take care of that.

The limited availability of on-chip memory of a processing chip adds complexity to data handling methods. Low cost network interfaces are implemented without external memory. The complexity in developing a memory controller for temporarily storing data is too high. Also, the costs of the memory chips and the development of the data management units are too big for consumer network interfaces.

The following points have to be considered for creating a data handling strategy for network interface cards:

- The communication bus between the CPU, the memory and the peripheral units is used by multiple devices.
- The data transfer is not predictable, because multiple components can access the memory.
- The communication bus can add communication overhead and can have some parameters, which influences the design of the data exchange strategy.
- Limited availability of on-chip memory on the network interface controller.

A data handling method must have an efficient way to transfer data between the processing system and the network interface without using much processing time. The processor should just be notified, if packet data was sent or received by the network interface. It also has to copy data from memory accessible by the operating system to memory accessible by the network interface. To achieve performant data handling between a host system and a network interface controller, an architecture has to be designed, which utilises the given resources efficiently.

This thesis presents an optimised data handling method for a network controller. The focus is on highly utilising the available resources. The design can be easily adapted for different applications and hardware environments.

1.2 Structure of the Thesis

Chapter 2 gives an overview of the state of the art technologies for developing a Network Interface Card. Basics of system development and describing methods are introduced. Different types of devices in a computer system, their addressing scheme and data exchange methods triggered by the system are described in Sections 2.2 to 2.3. Transferring data needs the usage of bus systems, which are presented in Sections 2.4 and 2.5.

Chapter 3 describes state of the art intellectual properties for handling data between host systems and peripheral units. It also includes the analysis of two data handling methods used in two network interface cards.

Chapter 4 includes the concept of the created data handling method and its hardware and software description.

The performances of the old and the optimised data handling strategy is, presented in Chapter 4, are compared by the measurements described in Chapter 5. The CPU usage for processing packets and the bandwidth of the NIC are measured.

Finally, Chapter 6 summarises the achieved goals of the thesis and provides an outlook on future works.

2 Technology Overview

This section gives an overview about available and used technologies solving the problem of handling data between components. The first section describes the development process of digital circuits and the other sections describe bus protocols and data handling methods.

2.1 System Design

Digital integrated circuits (ICs) packed into chips started to be used in the 1962s, when the Transistor-Transistor-Logic (TTL) based on bipolar transistor technology was introduced by Texas Instruments and Fairchild. Chips with less than 100 transistors packed into one die were produced to improve computers from IBM and DEC. These systems consist of multiple printed circuit boards packed with ICs to achieve the wanted functionality.

New fabrication processes and technology improvements were developed to increase the integration intensity of transistors in chips greatly. Figure 2.1 illustrates the technology improvements in terms of on die place-able transistors and development productivity of a digital circuit designer. The exponential increase of transistor density was observed by Gordon Moore in the year 1965. He predicted, that every 18 months, the integration density of transistors will be doubled (Moore's law) [Moo06]. Multiple techniques were developed to increase the productivity of design engineers.

This section gives a short introduction to system description techniques used in today's chip development.

2.1.1 Design Flow

The design flow of a system uses different abstraction levels to describe the functionality. Modelling systems at different abstraction layers leads to an accelerated development. The lower the abstraction gets the more details are implemented in the model. The verification of the abstracted model is done by simulation. These steps help the circuit designers to find design faults in an early development state. The rising design complexity and the improved fabrication processes, which enables developers to put billions of transistors on a chip, require models with different abstraction levels. Figure 2.2 illustrates the design flow of a system.

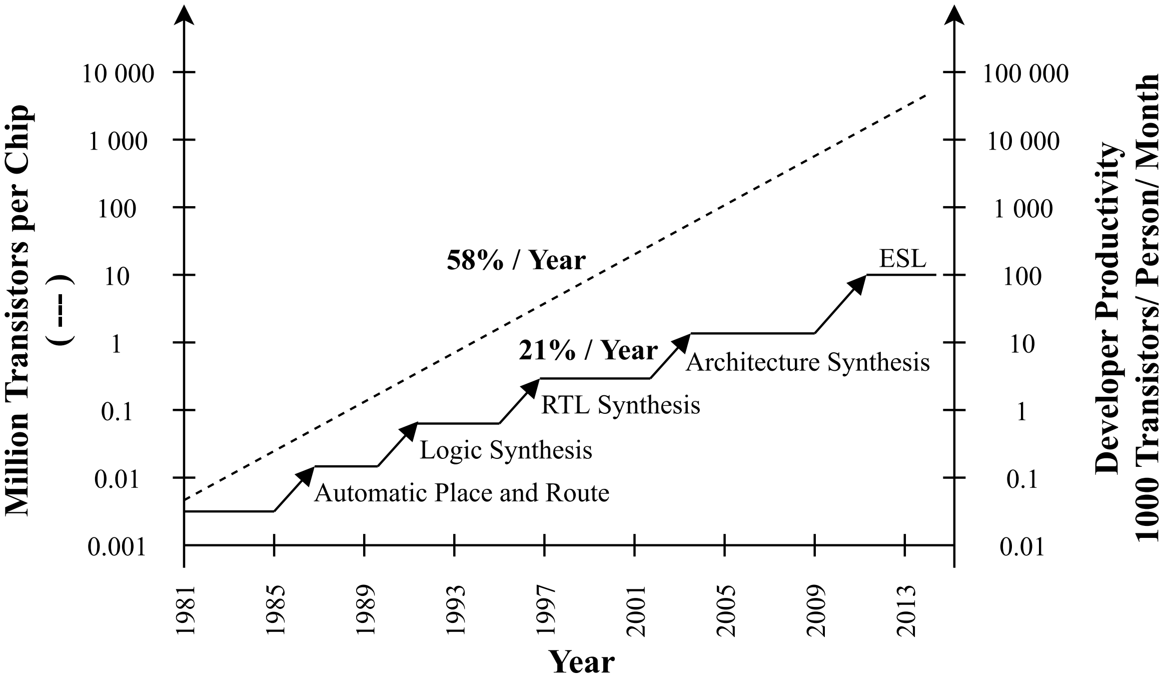


Figure 2.1: Progress in integration intensity of transistors in chips vs design productivity of digital circuit developers [Rei09, chapter 1]

Written System Level Specification

The start of every design is an idea which is written down on paper. The written form characterises the behaviour very abstract, because the wanted functionality is described without defining the realisation in any way. The final version of the written specification is converted into an executable one. The interpretation of some functionality can be misunderstood by the system designers, so the executable version can help developers to find misinterpretation of the specification.

Executable System Level Specification

The Executable System Level Specification is used to get an overview of the needed system components. The executable specification is written in a higher abstract language like C++ or SystemC. The available libraries to implement functionality enables a fast development of this model. The component splitting is done during the model design.

The system designer has to decide which components should be designed in hardware and which in software. Also, the interfaces or bus protocols between the components have to be specified. These steps are necessary to accelerate the development in Register Transfer Level (RTL) level, because multiple developers can build well defined components and verify their functionality.

Existing Electronic Design Automation (EDA) tools support designers in a way, that high level representation of functionality can directly synthesised into RTL. The number of synthesisable application is limited to only a few. The major part of the system has to be rewritten in a Hardware Description Language (HDL) like VHDL or Verilog.

One very important task is to create a simulation environment. The functionality of the model is checked and a Golden Model of the system is created. The Golden Model stores the output values

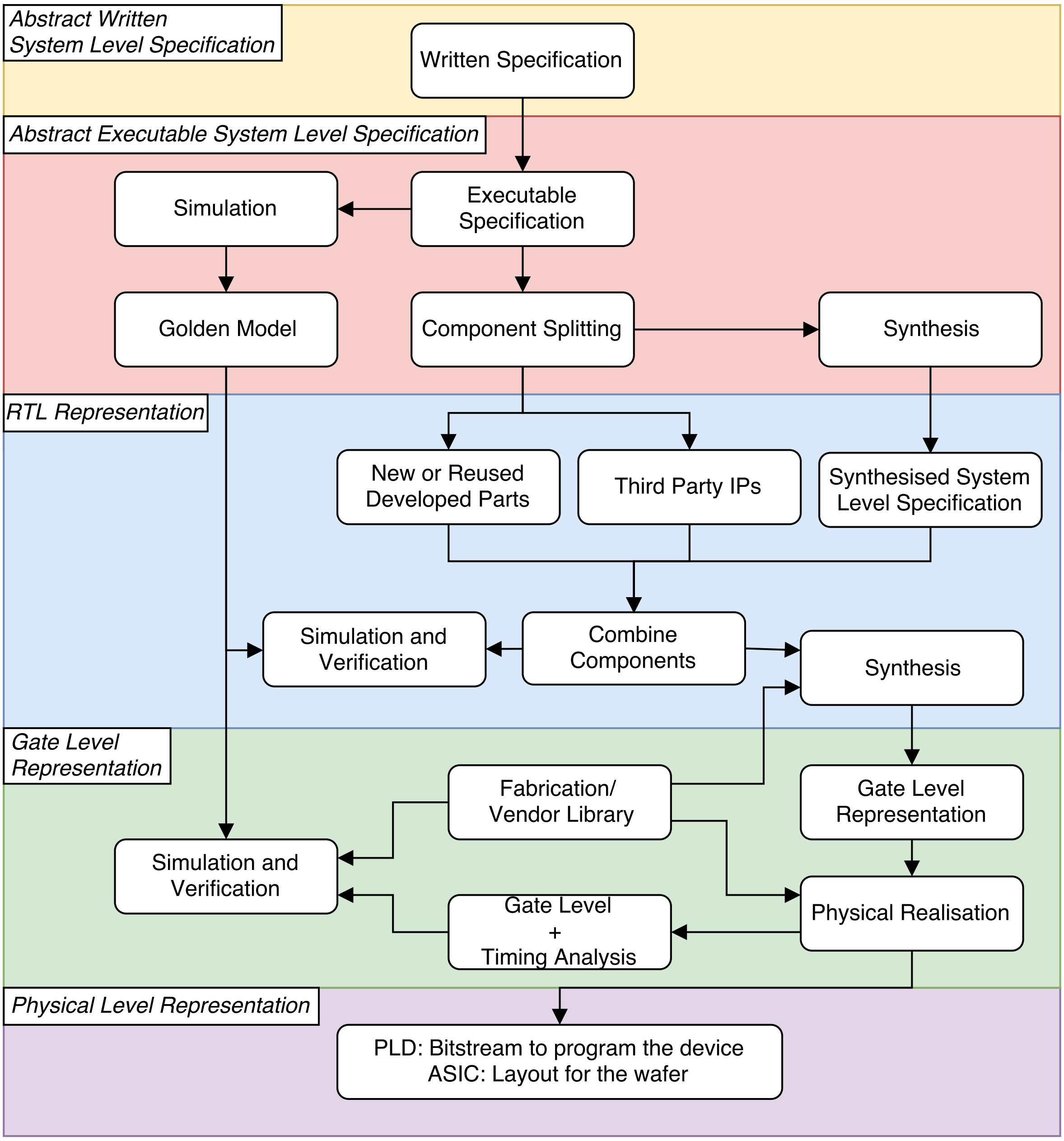

Abstract Written System Level Specification
Written Specification
Abstract Executable System Level Specification
Simulation
Executable Specification
Golden Model
Component Splitting
Synthesis
RTL Representation
New or Reused Developed Parts
Third Party IPs
Synthesised System Level Specification
Simulation and Verification
Combine Components
Synthesis
Gate Level Representation
Fabrication/ Vendor Library
Gate Level Representation
Simulation and Verification
Gate Level + Timing Analysis
Physical Realisation
Physical Level Representation
PLD: Bitstream to program the device
ASIC: Layout for the wafer

Figure 2.2: Design flow of a digital circuit [KB09, chapter 1]

of given input vectors. It is used in the other abstraction levels to check the correct functionality of the design.

RTL Representation

Only a few parts of the Executable System Level Specification can be automatically synthesised in RTL and the other parts have to be designed with an HDL. To reduce the development costs and time of a system, the use of third party Intellectual Properties (IPs) from other vendors or the reuse of already developed parts is necessary.

Third party IPs can be seen as very well tested and verified components. Multiple components like communication cores, data compression cores, cryptography cores or Central Processing Unit (CPU) cores can be bought from vendors. The well-defined and understood interfaces of the components enables fast usage of it.

The built RTL representation of the design defines a bit and cycle accurate model of the design. The interface specification process, which is done during the Executable System Level Specification, already defines the interfaces in a bit accurate way. The cycle accurate representation is modelled with the help of state machines. The states define different phases of the computation in different clock cycles. This cycle accurate and bit accurate representation is one reason, why HDLs are the only way to describe functionality in an efficient way.

The simulation and verification is not only done for the whole system. The component designers also have their own small simulation environment to verify the correct behaviour of their component. The system simulation environment of the Executable System Level Specification is redesigned in a way, that it can be used by the RTL model. The data of the Golden Model are used to check the correct behaviour. The new built model has to be simulated and verified against the data in the Golden Model.

Gate Level Representation

At this level of abstraction, the behaviour of the digital circuit is converted into gates. It describes the design bit right, clock right and delay right. Synthesis tool uses the vendor specific library to convert the functionality into this representation. The target information, served by the vendor libraries, enables the synthesis tool to do some optimisation. FPGAs have built in digital signal processing units and special memory blocks which can be used by the synthesis tool for getting an optimised gate level representation of the RTL model.

The physical realisation of the gate level model adds an important parameter to it: the signal time delay. The timing analysis is important for the correct behaviour of the system. The signal propagation and the time constraints of the gate inputs can be proven. The timing analysis also enables the tool to determine the maximum clock frequency of the design.

The gate level model representation is also verified by simulating the behaviour of the new abstraction level. Time delay data of the signals are added to the simulation and the result is compared with the Golden Model.

Physical Level Representation

The last step in the design flow is done automatically by the design tool. The design is represented in the target needed way. The fabrication labs need the layout of the chip, to build the wafer. Programmable Logic Devices (PLD) like Complex PLDs (CPLDs) or FPGAs need a bit stream, which is used to configure the digital circuits of the board.

2.1.2 Hardware Description Languages

In the 80th of the 20th century, the industry was faced with the problem of a higher getting design productivity gap of digital circuits. The improvements in fabrication made it possible, to put more and more digital circuits on a chip. The rising numbers of transistors, led to more expensive development processes. The major problems where that different software for simulation and no standardised way to document digital circuits was available. The not compatible and vendor specific tools for designing circuits had to be replaced by standardised ones.

The main difference between a software programming language and a HDL is, that the HDL has to be able to describe concurrent processes. The ability of parallel processing blocks marks the information processing strength of digital circuits. The HDLs Verilog and VHDL are the majors languages, which are used in the industry [Hop06, chapter 2].

Very High Speed Integrated Circuit Hardware Description Language

The Department of Defence of the United States initiated the program Very High Speed Integrated Circuit (VHSIC), which had the aim to decrease the development time of digital circuits. Digital circuit companies used different tools to describe their functionality. The lack of documentation and the incompatible design tools made the replacement of existing digital circuits difficult. This was the reason why the companies Intermetrics, IBM and Text Instruments were hired to create the HDL Very High Speed Integrated Circuit Hardware Description Language (VHDL). Companies, like EDA Tool developers, got interested in this topic too, which accelerates the development. In 1987 the American Institute of Electrical and Electronics Engineers (IEEE) standardised the HDL (IEEE 1076-1987) [KB09, chapter 2].

The development focus of VHDL was to create a behaviour description languages for circuits and systems, as well as to get a language for describing the simulation environment. The idea of creating an automatic converter from RTL to gate level came in the end of the 80th. The company Synopsis built a synthesis tool, which was able to build a gate level representation of the RTL design. Because not every function of VHDL code is synthesizable, the standard IEEE 1076.6 was created in 1999. Multiple standards, like IEE 1164, were created to help developers by implementing functions like many-level logic data types and arithmetic operators [KB09, chapter 2]. The VHDL standard was lastly updated in 2008 (IEEE 1076-2008).

Verilog Hardware Description Language

The company Automated Integrated Design Systems started 1985 to build a logic simulator with special hardware acceleration to find a countermeasure against the costly reengineering tasks at broken digital circuits. The company was renamed to Gateway Design Automation and published their first simulator Verilog in the year 1986. The simulator was able to simulate logical circuits and abstract behaviour description, which was written in a C like language. The Verilog simulator gathers the structure of a digital circuit in net lists. The ability to describe simultaneously running task makes the difference between HDL and sequential programming languages like C [Hop06, chapter 2].

The learning curve of Verilog is due to the similarity to C lower than it is for VHDL. VHDL uses a more complex substructure than Verilog. Apart of this, Verilog and VHDL can be used to describe any digital circuit. Because Verilog was built by Gateway Design Automation (later Cadence), it was a commercial product and so other companies were not able to build simulators or synthesis tools. Between 1997 and 2001, the IEEE standardised it as standard 1394 [Hop06, chapter 1].

2.2 Communication Basics

Many Different ways exist to communicate with components of a system. The standardisation of components, their classification and their addressing schemes is very important for the exchangeability of components. All Computer keyboards, printers or hard drives should use the same way to communicate. This section describes classification of components and addressing schemes.

2.2.1 Hardware Device Classification

A not perfect classification method tries to categories I/O devices into two categories: block devices and character devices [TB16, section 5.1]. A block device stores data in fixed block sizes, where each block has a unique address. The block sizes start from 512 bytes and end at 64 kilo bytes. An important property of a block device is, that each block can be read and written independently. Hard disks, USB memories or Blue-ray Discs are examples for block oriented devices.

Character devices use character streams for communication. The internal memory is not addressable and search operation can't be executed. Devices which are not similar to disks like printers, network interface cards (NIC) or computer mice are character devices.

There are many devices which don't fit in these models. Computer displays, touch screens or interrupt clocks can't be categorised. The generalisation of block and character devices helps to define device independent software.

The classification is independent of the system's bus protocols. Off-chip bus protocols like USB can be used to communicate with storage devices like USB memories, but can also be used to receive data from computer mice or keyboards. The software has to control the data handling and also the different data rates of the devices and bus protocols.

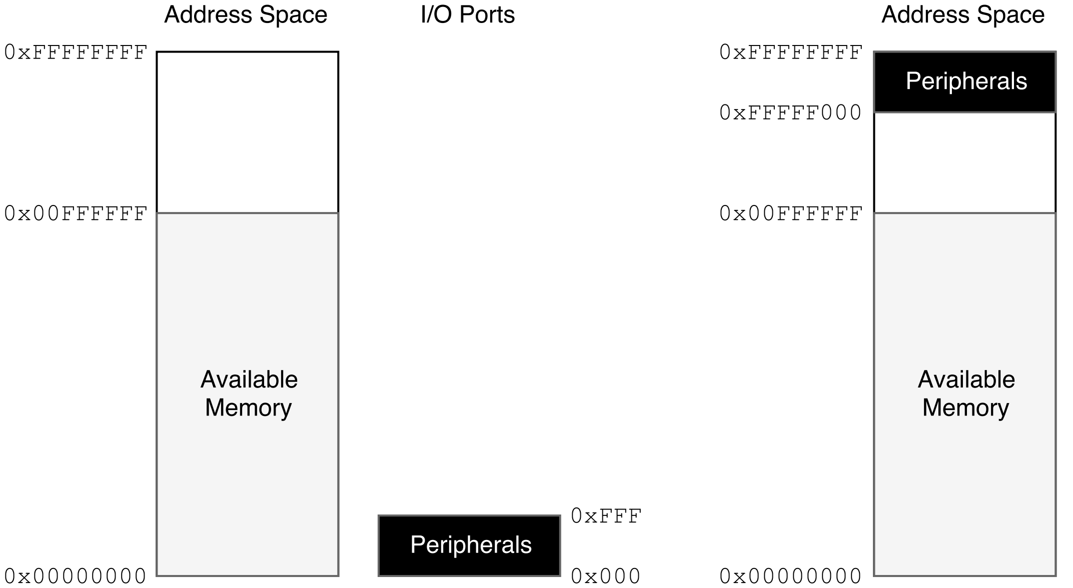
2.2.2 Device Addressing

Devices use registers for storing basic information or configurations. The communication of devices can be done by different bus protocols (see 2.4 and 2.5). The configuration of the devices is loaded during the boot process or during the activation of the device driver. Hardware accelerators, storing devices or output devices additionally have data buffers for storing data before processing them.

Multiple devices can be connected to a bus. A device gets activated, if the address of the bus corresponds to an address in the address space of the device. Processors uses different methods to access the peripherals of their buses. Figure 2.3(a) and Figure 2.3(b) shows two different concepts.

I/O Port

The simplest addressing scheme is I/O Port addressing. The peripherals get an 8-bit or a 16-bit address range, which enables the processor to address the control registers and the data buffers. The address space is protected from being accessed by user programs. Only the operating system is allowed to access the device.



(a) Separate address space for memory and peripherals

(b) One address space for memory and peripherals

Figure 2.3: Device addressing concepts [TB16, Chapter 5.1]

The processor uses special I/O commands to read from and write to devices. The special I/O commands add some complexity for the programmers to use devices. Application programmers have to develop assembler code to communicate with the device. The instruction set of processors can be different, so the assembler code has to be adapted, if the program is executed on different machines [TB16, Chapter 5.1].

Memory Mapped I/O

This addressing technique uses the same address space as the memory. The lower addresses of the space are used for addressing the main memory of the system and the higher are used for accessing peripheral devices. Each device has a unique address to avoid conflicts.

Application developer don't have to distinguish between a read or a write operation on system memory or a device. The processor uses the same operations for accessing the data. The address of the device is put on the bus address lines and the corresponding device is going to answer. This enables the programmer to write code for different machines without changing the assembler code for device access [TB16, Chapter 5.1].

2.3 Input/ Output Operations

Handling data between devices include different things to define like which bus does the device use, how does the device handle read and write requests, who is allowed to access the device or does the device handle the data block or character wise. This section is focusing only on the pure data exchange between the processor and the device. It describes how the software can be developed to transfer data to or receive data from a device.

Reading and Writing data from a processor to peripheral units have to be done in a proper way. The performance of a system depends greatly on efficient data handling between a processing unit and a device. Multiple concepts for handling data between peripherals and the processor exists. The performance of the system, the speed and size of the processor as well as the application helps to find the right data handling method.

The available I/O operation techniques of a processor depends on the available components of the system. This section describes 3 different ways to program the data exchange between the processor and peripherals. Modern processors have all these methods available, while microcontrollers often have only the first two implemented.

2.3.1 Programmed Input/ Output

This technique describes a synchronous one for data exchange. The application, which performs an I/O operation, sets an I/O request and then sets the appropriate I/O status bits. The processor is busy reading the status bit of the I/O module to know, when the I/O operation is completed. The resources of the process are used just for reading the status bits.

The Programmed I/O technique uses the processor as long as the I/O module has done the operation or is ready to get more data. The program waits, until the I/O operation has been executed. This technique degrades the performance level of a system heavily, if much data has to be transferred from or to the device.

The Programmed I/O operation is the simplest one and therefore often used in small microcontrollers or for transmission of few data. Small systems, which only have to do one task, are using also the Programmed I/O technique [SM15].

2.3.2 Interrupt Driven Input/ Output

The Programmed I/O operation wastes much time in waiting for the I/O module to set the status bit to ready. The processor could do some other tasks while the I/O module is processing the operation. The idea of Interrupt Driven I/O is to have a notification mechanism, where the processor gets informed, if the I/O module is ready for new data. During waiting for the ready bit, the processor could be used to do some other tasks, which do not depend on the I/O module operation or could go to a sleep mode, where it saves energy.

Nearly every processor is supporting interrupts. Interrupts help the processor to get informed, that some asynchronous event has taken place. This asynchronous event could be, that a specific time has passed (timer interrupt), an arithmetic execution has caused a problem (program interrupt), a power failure or memory parity error has taken place (hardware failure interrupt) or an I/O module operation has been finished. The interrupt can halt the execution of a normal program to inform the process that a specific action has taken place.

The Interrupt Driven I/O can issue an I/O command to a module and then can switch to another program. The program is executed as long as the I/O module is not issuing an interrupt to the processor. After an I/O interrupt, the processor switches back to the I/O program and then can do some other operations.

The described technique rises the efficiency of the processor usage compared to the Programmed I/O. The processor only gets activated, if the I/O module is ready for sending or getting new data. The processor still has to trigger the next I/O operation.

The two described methods have two major drawbacks [SM15]:

- The limitation of the transfer rate is caused by the speed, which the processor can test and service the device.
- The processor has to set up and manage each I/O transfer.

If a large amount of data has to be transferred, a separate controller should handle the transfer. The data transfer could be done by just knowing the memory address of the data and the destination of it. The processor could then be used for more important tasks.

2.3.3 Direct Memory Access

This I/O transfer technique uses a separate module, a Direct Memory Access Controller, to perform the data exchange. The DMA controller is one module, which has access to the memory and the peripherals. The processor has to setup the DMA controller for transferring data from the memory to the device or vice versa. The processor gets an interrupt, if the transfer has been finished.

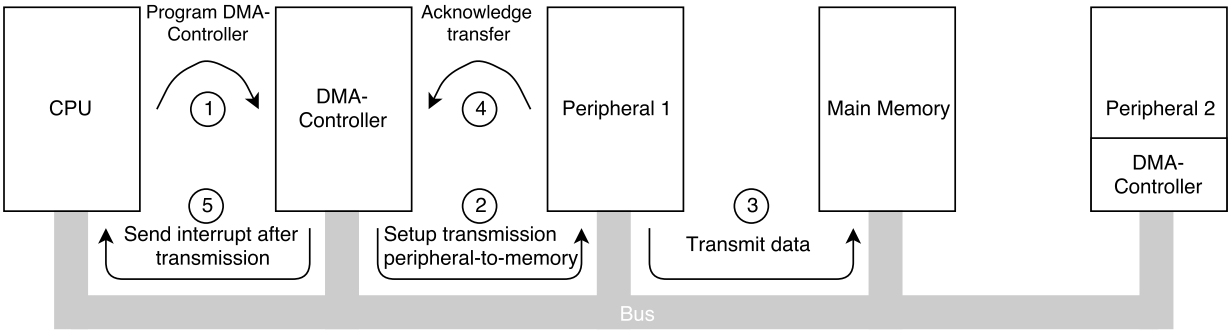


Figure 2.4: DMA transfer flow [TB16, Chapter 5.1]

Figure 2.4 illustrates a DMA transfer flow. At first the processor has to setup the DMA controller for the data exchange. The DMA controller needs the storage address of the data, the address of the peripheral, the amount of data, which should be transferred, and the direction of the transfer (memory-to-peripheral or peripheral-to-memory). After setting up the DMA controller, the processor can process other tasks. The second step has to be done by the DMA controller. It has to setup the data exchange for every word transfer between the memory and the peripheral. The third step transfers the data from the peripheral to the memory. The peripheral sends an acknowledge to the DMA controller, after each transfer. This notifies the DMA controller that the transaction has happened (fourth step). The steps 2 to 4 are repeated, until all data is transferred. After that, the fifth step is done by the DMA controller, where it informs the processor, that the dara transfer has finished, by triggering an interrupt.

The concept of using a DMA controller enables an efficient data transfer between the memory and the peripherals. It also enables new ways of data exchange. As shown in Figure 2.4, a system can have peripheral units, which have DMA controllers implemented. These controllers can be used, to trigger memory-to-peripheral communication without the help of the processors. The communication between the processor and the peripheral could be triggered by the content of a specific memory location, like a processing descriptor. The peripheral unit could periodically read from a specified location of the memory to get informed, that new data is available for processing.

2.4 On-Chip Bus Protocols

Different chip parts use standardised on-chip bus protocols to communicate. This enables the developer to easily reuse IP cores or to easily exchange the functionality of a chip with a better version. FPGA and IP vendors have defined their own interfaces but have made the documentation of it public.

This section describes a streaming and a memory mapped communication interface developed by Altera. The concept of streaming and memory mapped bus protocol can also be found in the area of ARM processing systems. The name of the communication interfaces are then AXI-Stream and AXI-Memory Mapped interfaces. The bigger getting systems and the higher getting complexity in interconnecting devices have led to a network based communication, which can be implemented on a chip. The basics of it are described in Section 2.4.3.

2.4.1 Avalon Streaming Interface

High bandwidth, low latency and unidirectional data flow can be built with the Avalon Streaming (Avalon-ST) interface. It can transport streams or packet oriented data from a source (SR) component to a sink (SI) component. The Avalon Stream interface can implement channels for separating logical paths between two ports [noa18]. Table 2.1 describes the signals of an Avalon-ST interface.

Table 2.1: Avalon-ST Interface signal description [noa18]

Signal Role	Width	Direction	Description
channel	1 - 128	SR → SI	Defines the logical channel
data	1 - 4096	SR → SI	Data bus signal.
error	1	SR → SI	Bit mask for marking bytes with errors.
ready	1	SI → SR	If the sink is ready to receive data, the ready signal is asserted.
valid	1	SR → SI	The signal gets asserted by the source, if valid data is available on the data bus
empty	1-5	SR → SI	Represents the empty symbols in the data bus. The master doesn't have to utilize the whole signals of the data bus. The not used symbols are marked with the empty signal.
endofpacket	1	SR → SI	Signals the sink component the end of a packet.
startofpacket	1	SR → SI	Signals the sink component the start of a packet.

The Avalon-ST interface has different transport models defined. The difference between a continuous data stream transfer and a packet oriented data transfer is, that the startofpacket and the endofpacket signal is set for the packet oriented one. The start of a transfer can also be configured in different ways. One configuration would be, that the sink has to be ready before the valid signal of the source is asserted. Backpressure is the name of the other configuration, where the source asserts the valid signal before the sink asserts the ready signal. Figure 2.5 illustrates a transfer with backpressure.

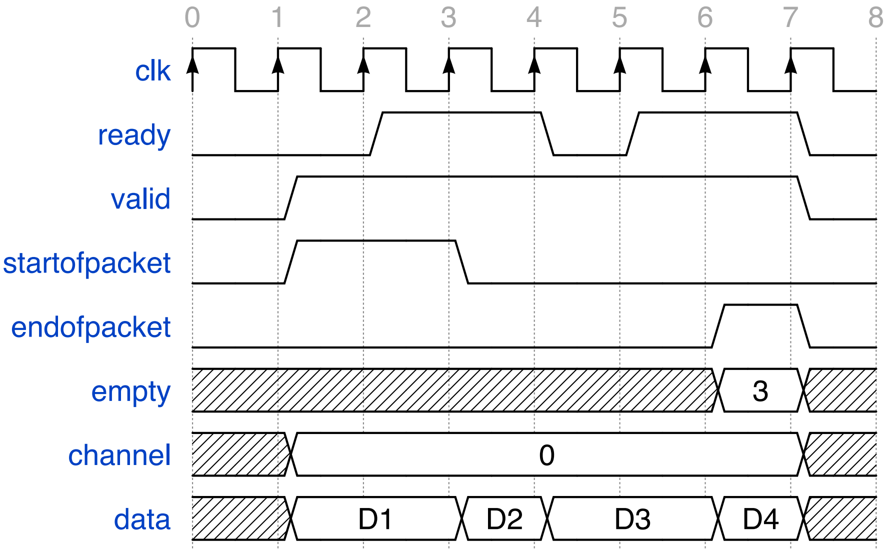


Figure 2.5: Avalon Streaming data transfer with backpressure [noa18]

1. The source sets the valid and the startofpacket signal to indicate, that it is has new data for transmitting (backpressure).
2. The sink asserts the ready signal. The data transfer is started.
3. The first data byte is captured by the sink. The source clears the startofpacket signal and asserts new data to the data signal.
4. The sink is capturing the second data byte. Then it clears the ready signal to inform the source, that it is not ready for receiving data. The source asserts new data to the data signal.
5. The source waits for an active ready signal. The sink is asserting the ready signal.
6. The next data byte is transmitted and the last data packet is set to the data signal. The endofpacket is asserted and the empty signal is set, to inform the sink, that not every symbol on the data bus is used.
7. The signals are cleared and the transmission is finished

2.4.2 Avalon Memory Mapped Interface

The address-based read/write interface is used to communicate between master and slave components. Microprocessors, memories, UARTs, DMAs and timers use the memory-mapped interfaces to address devices and to write data to memory located on these devices.

Table 2.2 gives an overview of the used signals for communication:

Table 2.2: Avalon Memory Mapped Interface signal description [noa18]

Signal Role	**Width**	**Direction**	**Description**
address	1 - 64	M → S	Represents the byte address of memory.
bytenable	2, 4 ,8, 16, 32, 64, 128	M → S	Each signal represents the location of a byte transferred during the transaction. If the <n> bit is low, the byte <n> is not used for transmitting data. The symbol doesn't have valid data and it should not be used by the communication partner.
read	1	M → S	A read request triggered by the master.
readdata	8, 16, 32, 64, 128, 256, 512, 1024	S → M	Data, which was requested by the master.
response	2	S → M	Optional signal for transferring the status of the response.
write	1	M → S	A write triggered by the master
	8, 16, 32, 64, 128, 256, 512, 1024	M → S	Represents the data which should be written to the address specified in the address signals.
lock	1	S → M	If the master gets access to teh bus, the lock signal is set to notify the slave, that the master can do multiple transactions.
waitrequest	1	S → M	The slave can assert this signal to notify, that the read or write request is processed and that it is not ready for the next request.
readdatavalid	1	S → M	Is asserted if the slave has valid data on the read data signal. It is used for variable-latency and pipelined read transfers.
writeresponsevalid	1	M → S	Optional signal which is asserted if the value of the write response is valid
burstcount	1 - 11	M → S	It is used to define the length of a burst transfer.
beginbursttransfer	1	M → S	Indicates the begin of a burst transfer.

The Avalon-MM interface has five different transfer modes for communicating with slaves. The transfer modes use different signals for configuring the transfer.

Typical Read and Write Transfer

This transfer mode supports read and write transfers, where the slave can control the flow with the waitrequest signal. The slave can stall the transaction by asserting the waitrequest signal.

Figure 2.6 shows the sequence and the used signals for transferring data with this mode.

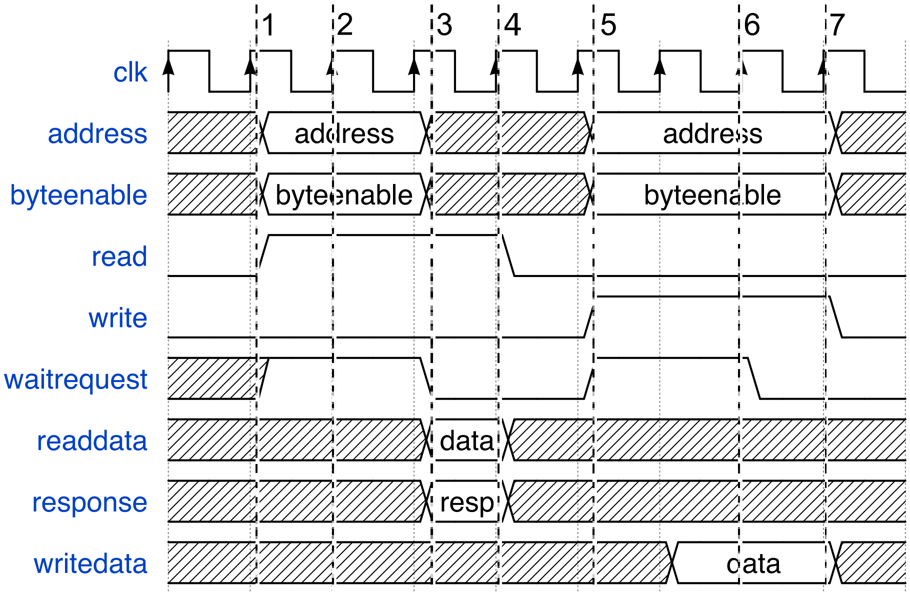


Figure 2.6: Typical read and write transfer [noa18]

1. The address, the byteenable and the read signal are asserted by the master, to start a read transfer. The waitrequest signal is immediately asserted by the slave, to inform the master, that the next readdata signal don't represent valid data.
2. The master samples the active waitrequest, which means, that the request is stalled. The asserted values of the master remain constant.
3. The waitrequest signal is cleared by the slave. This indicates that the data transferred with the readdata signals are valid. Additionally the response signal is used to send the status of the readdata.
4. The master is triggering a write request by setting the correct data to the writedata signals and asserting the write signal. The slave asserts the waitrequest signal to stall the transfer.
5. The waitrequest signal is cleared, which indicates that the transfer will finish with the next rising edge.
6. The master clears the write signal and the transfer finishes.

Transfer Using the waitrequestAllowance Property

The property waitrequestAllowance is used for specifing the number of write transfers a slave must accept after it has asserted the waitrequest. The slave must have a buffer to store the values sent with the signal writedata. If the waitrequestAllowance is 0 a typical write transfer is triggered.

Read and Write Transfer with Fixed Wait-States

The Avalon-MM interface has two properties, the readWaitTime and the writeWaitTime, to specify the time, a master has to wait until data is written to the slave or valid data are asserted on the readdata signals. This mode doesn't need the waitrequest signal, because of the fixed wait state configured by the two properties

Pipelined Transfer

The pipelined transfer enables the master to trigger multiple read request to the slave. The slave asserts the waitrequest signal, if it can't store more requests. The readdatavalid signal is used to indicate valid data on the readdata signal. Figure 2.7 illustrates a pipelined read transfer with variable latency:

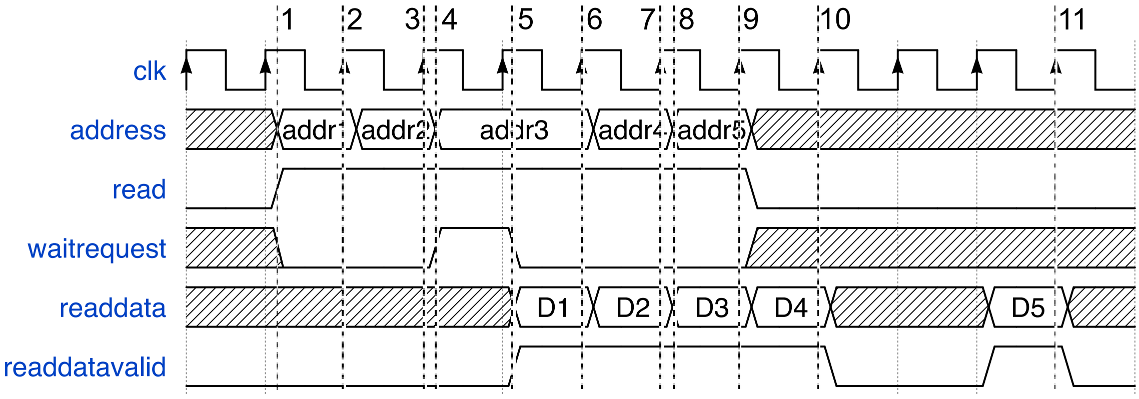


Figure 2.7: Pipelined read transfer with variable latency [noa18]

1. The master starts a read request by setting the address signals and asserting the read signal.
2. The slave captures the address. The master changes the address and doesn't clear the read signal, to request a second read transfer.
3. The slave captures the address. The master changes the address and doesn't clear the read signal, to request a second read transfer.
4. The slave can only handle 2 read request, so the waitrequest signal is asserted.
5. The slave sets the readdatavalid signal after asserting the wanted data on the readdata signals. The waitrequest is cleared, to indicate, that new requests can be processed.
6. The master receives the data and can request a new read transaction. The slave changes the readdata signals and leaves the readdatavalid assigned. The slave stores another read request of the master.
7. The slave asserts new data on the readdata signal and the master requests a new read action.
8. The slave asserts readdatavalid and sets the readdata value.
9. The slave gets the 5th read request of the master. The read signal is cleared, because the master doesn't have more read requests.
10. The master captures data from the readdata signals
11. The master gets the data of the fifth read request. After that the readdatavalid signal is cleared and the pipelined read transfer is finished.

Burst Transfer

Burst transfers are used to transfer multiple data words by just requesting one burst transfer. The address specified while requesting the transaction is increased automatically by the slave. This transfer mode increases the efficiency when handling multiple data words at a time, like it is done when data is read or written from a memory.

Figure 2.8 illustrates the steps for a write transaction

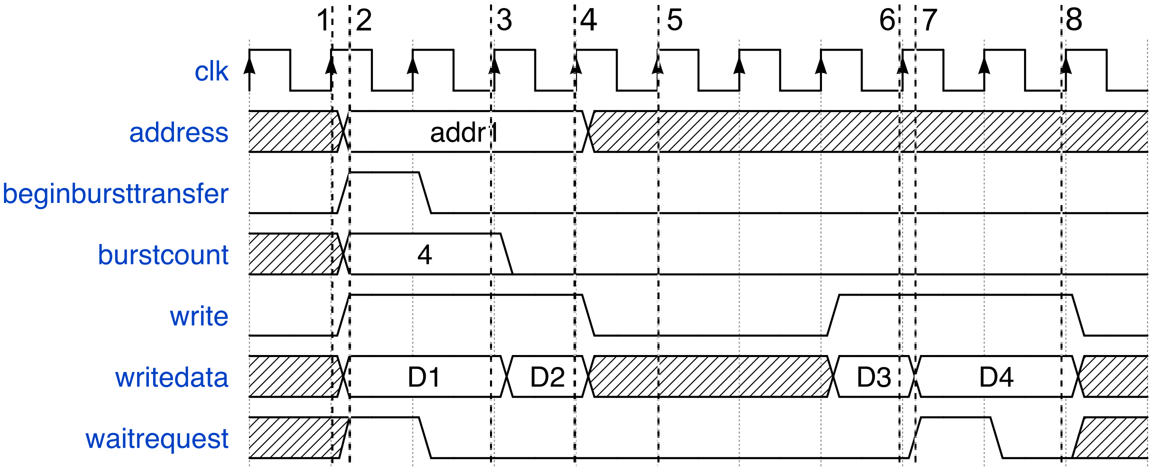


Figure 2.8: Write burst transfer transitions [noa18]

1. The Master asserts the address, the beginbursttransfer, the burstcount, the write and the writedata signal to initiate a write burst transaction. The beginbursttransfer signal is only asserted for one cycle.
2. The slaves sets the waitrequest signal to stall the write request.
3. The waitrequest signal was cleared by the slave, so the first data is written.
4. The second data is written to the slave. The master has no more data ready so the write signal is cleared and the burst is paused.
5. The burst is paused
6. The write signal is asserted again and the slave reads the data.
7. The waitrequest is asserted so the burst is paused again
8. The last data are written and the burst transaction is finished by clearing the write signal

A read burst transaction sequence is a bit different to the write transaction. The slave signals valid data on the readdata bus with assigning the readdata valid signal. Figure 2.9 illustrates a read burst transaction for 3 data words.

1. The Master asserts the address, the beginbursttransfer, the burstcount, and the read signal to initiate a read burst transaction. The beginbursttransfer signal is only asserted for one cycle. The slave sets the waitrequest signal to stall the read request.

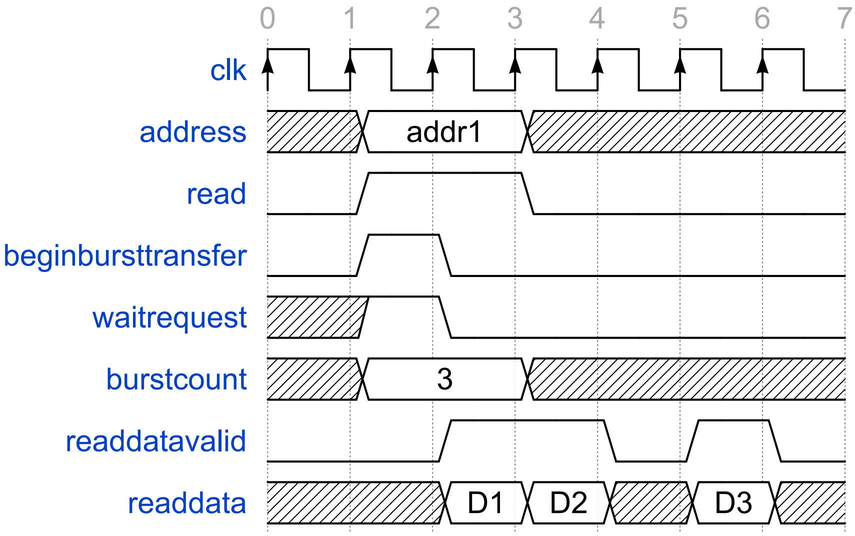


Figure 2.9: Read burst transfer transitions [noa18]

2. The waitrequest signal is cleared by the slave and the readdatavalid signal is assigned, to inform the master, that valid data is assigned on the readdata signal
3. The master clears the read signal because the waitrequest signal is cleared. He also receives the first data word, because the readdatavalid signal is set.
4. The master receives the second data word. The slave stalls the transaction by clearing the readdatavalue
5. The slave assigns the readdatavalid to signal valid data on the readdata bus
6. The last data is received by the master

2.4.3 Network On Chip

Communication between multiple processing elements (PEs), processors and storage elements increase the requirements on scalable communication systems. The interconnect structure is moving away from bus-based to network-based solutions. Network on Chip (NoC) connects functional units of a chip via a packet-switching communication network. The scheme can be compared with the internet, where multiple computers are connected. With the help of routing information, packets are sent through the network. Figure 2.10 illustrates a typical NoC topology. Each PE has implemented a packet switch for forwarding packets [BA13, chapter 4].

The NoC has the following features:

- Asynchronous data transfer through the network.
- Transmission of packets instead of words.
- No dedicated address lines are needed.
- High bandwidth, because of the distributed propagation delay across multiple switches and effective pipelined packet transmission.

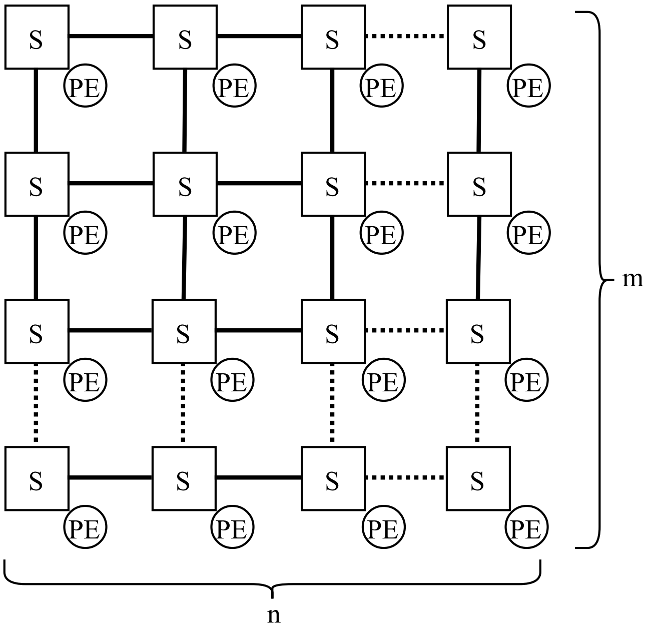


Figure 2.10: Typical standard N x M mesh topology [BA13, chapter 4]

- Transmission can be done in parallel, if the sender and receiver can handle more transmission channels.
- Theoretical infinity of scalability.
- Simplify reuse of IP cores.
- Enables a higher level of parallelism.

2.5 Off-Chip Bus Protocols

Multiple ways exist to communicate between chips and components, which are externally connected. The performance of the component, the application and the environment where the component is used are helping to find the right protocol.

This section describes to common off chip bus protocols: The Peripheral Component Interconnect (PCI) Express bus and the Universal Serial Bus (USB).

2.5.1 PCI Express Bus

The state-of-the-art way to connect components of a computer system is the PCI Express bus. It was built for high-speed, high-performance, serial and point-to-point communication between devices. A packed based protocol is used for transmitting data [Sol14].

PCI Express Link

The PCI Express bus uses links for defining the communication channel. Every link can have multiple of two, low-voltage, differentially driven signal pairs. One pair is for receiving and the other pair for sending data between the components. The link doesn't have a dedicated wire for transferring a clock signal. It uses an encoding scheme to recover the transfer clock out of the data. The PCI Express standard version 1 to 2 used an 8b/10b encoding scheme and starting from version 3.0 they used a 128/130b encoding scheme. The upgrade of the encoding scheme reduced the bandwidth overhead from 2/10 = 20% to 2/130 = 1.54%.

The PCI Express supports different number of lanes for a link. A link consists at least of 1 lane, while a lane consists of one sending and one receiving signal pair. The standard specification describes x1, x2, x4, x8, x12, x16 and x32 lane widths. The higher the lane width, the higher also the bandwidth of the system. The PCI Express components identify the number of available lanes and the operating frequency during the initialisation process.

The bandwidth of the different PCI Express versions is shown in Table 2.3.

Table 2.3: Bandwidth of different PCI Express versions [Sol14]

Bandwidth (GB/s)	**Link Width**				
	x1	**x2**	**x4**	**x8**	**x16**
PCIe 1.x ”2.5 GT/s”	0.25	0.5	1	2	4
PCIe 2.x ”5 GT/s”	0.5	1	2	4	8
PCIe 3.0 ”8 GT/s”	1	2	4	8	16
PCIe 4.0 ”16GT/s”	2	4	8	16	32

PCI Express Fabric Topology

Figure 2.11 illustrates a PCI Express fabric. The figure illustrates a hierarchical structure with the following components:

- **Root Complex:** Is the top of the hierarchy of a PCI Express bus. It connects the CPU and the memory with the PCI express endpoints. Every interface of the component defines a separate hierarchy domain, which can include multiple switches and endpoints.
- **Switch:** A PCI Express switch forwards packet data from a PCI Express device to the location specified in the header of the packet. It uses a round robin or a weighted round robin arbitration to decide, which packet is forwarded.
- **PCI Express Endports:** A Requester or Completer of a PCI Express transaction is defined as endpoint. Graphic cards, NICs or USB host controller, which are attached to the PCI Express bus, are examples for endpoints.
- **PCI Express to PCI/PCI-X Bridge:** Works as bridge to the older bus systems PCI and PCI-X.

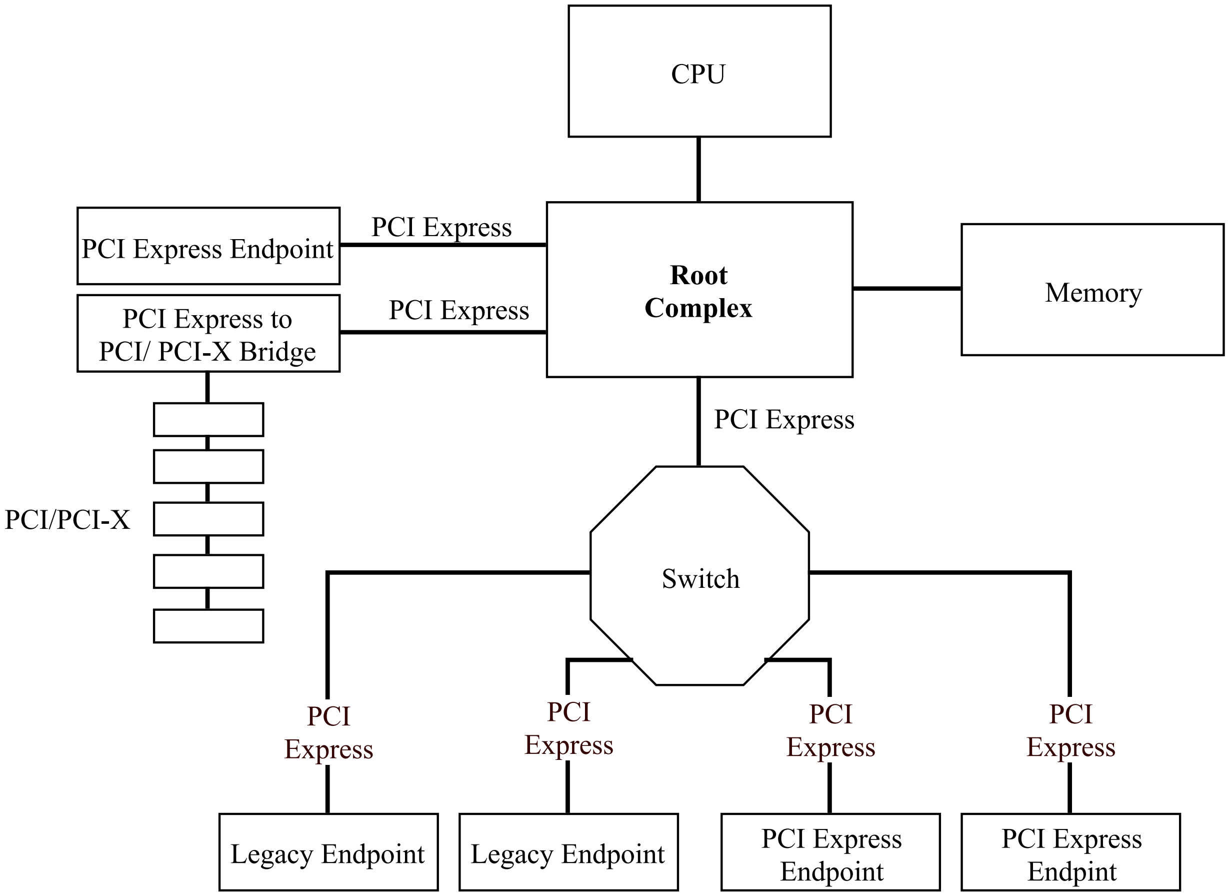


Figure 2.11: PCI Express Fabric Topology [noa10]

2.5.2 USB

The Universal Serial Bus (USB) is a peripheral bus for low- and high-speed devices like keyboards, printers or hard drives. A personal computer can connect up to 127 peripheral devices, which will automatically be configured as the device is attached. The data rate of the bus changed from USB 1.0 width 1.5Mbps to 10Gbps with USB 3.1 [Ela18, chapter 6.9].

Architecture

Figure 2.12 illustrates the basic architecture of the USB bus. Each USB device is directly communicating with the host controller. The USB hubs are only forwarding the data from the devices to the host controller.

- **Host Controller:** Every USB bus has only one host controller which initiates all data transfer on the bus. Connected and disconnected devices get detected with a polling action triggered by the controller.
- **Root Hub:** The power distribution of the devices and the activation of ports are controlled by the root hub. It provides also the connection between the host controller and the USB ports.

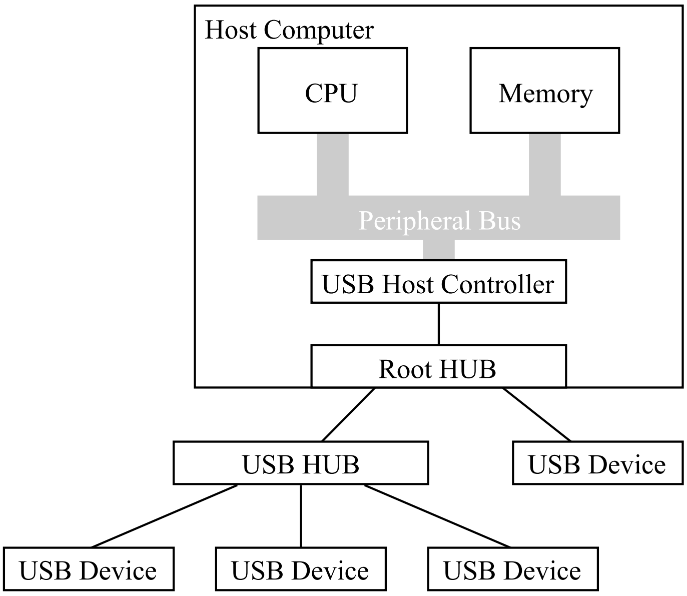


Figure 2.12: Architecture of a USB bus [Ela18, chapter 6.9]

- **Hub:** This component helps to expand the number of connected devices. If data is sent from the host controller to the USB device, the hub transmits it to every device. The recipient device of the data accepts the packet and the others discard it.
- **USB Cable:** Four Wires are used for data transmission: A ground wire, a V wire for powering the device and two signals, called D+ and D-, for data transmission.
- **USB Device:** Different types of devices can use the USB bus to transmit information. The basic information of the type, the manufacture ID and the data rate can be easily read by the host controller. This enables the system to establish a basic connection and to transfer basic data.

3 Data Handling Techniques between Host System and Peripheral Units

Multiple bus protocols can be used to communicate with peripheral units. Systems use different bus protocols depending on their hardware and system requirements. The protocols only define the data transport to the hardware. The data fetching strategy, which has to be organised by the host system and the hardware, has to be defined.

This chapter describes Intellectual Properties (IPs) for transferring data from the host system to the hardware as well as data handling implementations for Network Interface Cards (NICs).

3.1 Available Intellectual Properties For Data Handling

Each hardware accelerator or peripheral unit needs a way, to communicate with its host system. The data transfer should be implemented in an efficient and standardised form. On-chip bus protocols give the developer standardised interfaces, which can be used to implement the data transfer.

Data handling is a task, where designers have to develop the software (device driver) and the hardware part of it. To minimize its development time of new systems, IPs are used to implement functionality. This section discusses the open-source high-performance PCI Express Version 3 Streaming Library JetStream and Xillybus, which is an FPGA IP core for easy DMA over PCI Express.

3.1.1 Xillybus

The Xillybus IP core is a commercial DMA-based end-to-end solution for data transport between an FPGA and a host system running Linux or Microsoft Windows. The IP core can communicate with personal computers as well as with embedded systems. The PCI Express bus is used for communicating with personal computers. Xillybus is also available for ARM-based processors, where the AXI bus is used as an interface.

Xillybus provides the developers a well-defined interface to handle data between the host and an FPGA. A user program can easily read and write to different interfaces, which are defined as devices. Linux operation systems access devices by opening a file or by piping data directly into a file. Its communication to the programmable logic is done by FIFOs. If data is available (empty signal is cleared) or the FIFO has space for receiving data (full signal is cleared), the Xillybus can handle data between the FPGA and the host system.

A custom Xillybus IP core can be generated by using an online web interface. With the help of different parameters like number of streams, their bandwidth or their direction, the online tool creates a customised hardware block and its device driver.

Hardware

Figure 3.1 illustrates the block diagram of the Xillybus hardware. The left diagram illustrates the block diagram of an embedded system implementation of the Xillybus, which communicates with an on-chip ARM processor core using the AXI bus interface. The right block diagram uses the PCI Express IP core of Xilinx or Altera.

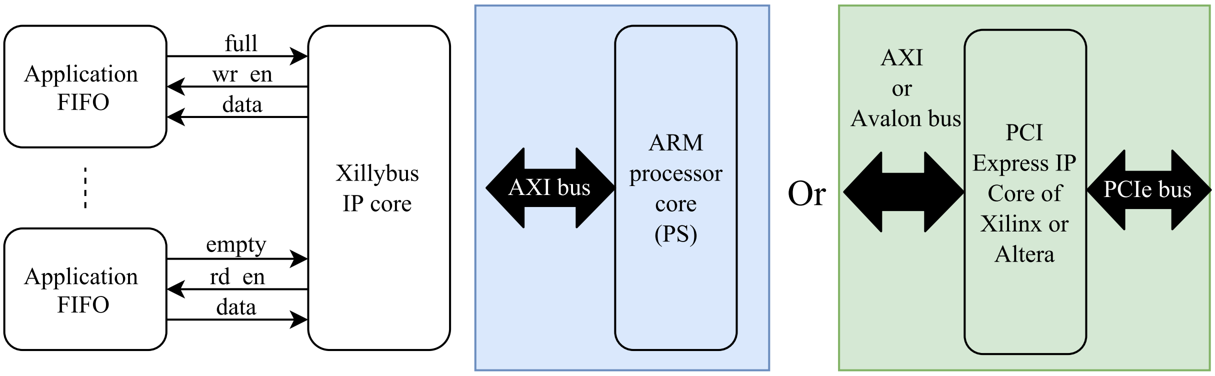


Figure 3.1: Block diagram illustrating Xillibus IP core communication between a host system and programmable logic [Ltd18a]

The communication between the ARM processor and between the Xilinx PCI Express IP core is done using AXI bus interfaces. The Altera PCI Express IP core uses the Avalon bus interface. The Xillybus IP core communicates with the application logic with the help of application FIFOs. The full and empty signals are used to signal the Xillybus IP core, that new data is available or that data can be written to the FIFO.

The Xillybus IP core can be customised with an online tool on the website of Xillybus. The following parameters can be configured:

- **Device file name:** Describing the device name of the Xillybus interface starting with xillybus_{device file name}.
- **Direction:** The direction of the stream can be upstream (FPGA-to-host), downstream (host-to-FPGA) or bidirectional. The bidirectional selection enables the user to configure the up- and downstream. The two streams share a device file, which can be opened for reading and writing separately.
- **Use of device file:** The stream can be used for different purposes like audio or video streams, data exchange with coprocessors or to address data interfaces.

  When it is used for data acquisition or frame grabbing, the buffer sizes and the flow control is chosen to support a continuous data stream.

If "Data exchange with coprocessor" is selected, the stream is optimised for high throughput.

The option "Address/ Data interface" adds address lines to the FPGA and the stream is configured as synchronous.

- **Data width:** Xillybus supports 8-bit, 16-bit and 32-bit interfaces. High performance streams should use 32-bit data width, because the internal data paths of the Xillybus are optimised for 32-bit words.

  The performance of the bus greatly relies on the data width. If, for example, a host-to-FPGA link is 32-bit wide and only 2 bytes of data are written from the user program to the driver, the driver will wait to send the data to the FPGA until two more bytes are written by the application. This could lead to an undesired behaviour of the interface.

- **Expected bandwidth:** The streams are operating with the rate at which data is made available to them. For performance tuning, the IP core generator needs the expected speed to define the DMA buffer size in the host memory. A realistic bandwidth should be set to support the generation and also to display warnings, if the expected bandwidth can't be reached by the bus.

- **Synchronous / Asynchronous stream:** These attributes configure the driver of the Xillybus. A synchronous stream blocks the writing to the file until the data is transported. Asynchronous writes are the preferred choice in almost all cases, in particular, if the communication must be performant.

  Synchronous streams are used for sending commands to the FPGA. A continuous flow of data is nearly impossible.

- **Number of buffers:** The Xillybus driver has to allocate DMA buffers to guarantee a performant data exchange. The IP core needs the number of buffers to prepare the logic appropriate.

  The possible size of buffers is between 2 and 1024. Slow streams (< 10Mbps) need 4 buffers, while high bandwidth streams should use 16 to 64 buffers.

- **Size of each buffer:** This attribute sets the size of each DMA buffer. The total size of all buffers should not exceed 8MB on Linux systems and 512MB on Windows. The allocation of more memory is refused by the operation system.

  A fully filled buffer sends a hardware interrupt to the host system, which should be kept at a level sane to the processor.

Software

Accessing the FPGA from the host is done by the driver. The Xillybus IP core generator also generates the driver and the device file for every stream. The streams can be handled like files, which enable every practical programming languages to use the FPGA without adding any special module extension or other adaption. Programming languages only need functions to open, read or write files.

The illusion of continuous data stream is done by a handshake protocol between the FPGA and the host. While the driver is loaded, the DMA buffers are allocated and the FPGA is informed about their addresses. The number of DMA buffers and their size is hard coded into the Xillybus IP core.

Example Code

```
 1 int main() {
 2   int fd, rc, numbytes;
 3   unsigned char *buf;
 4 
 5   // open file with low-level function
 6   fd = open("/dev/xillybus_ourdevice", O_WRONLY);
 7   // proof, if device is available
 8   if (fd < 0) {
 9     perror("Failed to open devfile"); exit(1);
10   }
11   // initialise the variables
12   // ..
13   // endless while loop for writing data
14   while (1) {
15     // writing "numbytes" data from the buf array to the FPGA
16     rc = write(fd, buf, numbytes);
17 
18     // error handling
19     if ((rc < 0) && (errno == EINTR)) {
20       continue;
21     }
22 
23     if (rc < 0) {
24       perror("write() failed"); break;
25     }
26     // do something with "rc" bytes of data
27     // prepare buffer for writing data to the FPGA
28   }
29 }
```

Listing 3.1: Example code for transferring data using Xillybus [Ltd18b]

Listing 3.1 illustrates the example code for a write action to an FPGA stream ”ourdevice” [Ltd18b]. The device is opened with the low-level open function. The device is opened for writing in blocking mode. This means, that the `write` function waits until all bytes are written to the FPGA. A successful open is notified, if the file descriptor has a number greater zero assigned. The while loop continuously writes to the FPGA (line 12). Line 16 issues a write of `numbytes` bytes to the FPGA from the buffer `buf`. The `write` function returns the written bytes. The first if-statement checks if the `write` function returned prematurely, because of a system interrupt as result of the process receiving a signal from the operating system. The program was not able to write data, but no error occurred. The second if statement can terminate the while loop, because a real error has occurred.

3.1.2 JetStream

JetStream is an open-source high-performance PCI Express 3, which supports FPGA-to-host and FPGA-to-FPGA communication [VKVF16]. FPGA vendors already have IPs available to support the development of PCIe connected devices. Nevertheless, the need of several additional logic is required to manage the communication between the host system and an FPGA.

JetStream additionally enables the developer to transfer data directly between FPGAs. This enables the developer to separate logic over several FPGAs. A multi-FPGA solution can help the developer to reduce the hardware costs, by using several smaller and cheaper FPGAs. Power consumption of FPGAs could also cause problems for peripheral units, which gain their power only from the bus system. 8-lane PCI Express cards are only allowed to consume 10W for half height and 25W for full height cards in high power mode [noa10].

Hardware

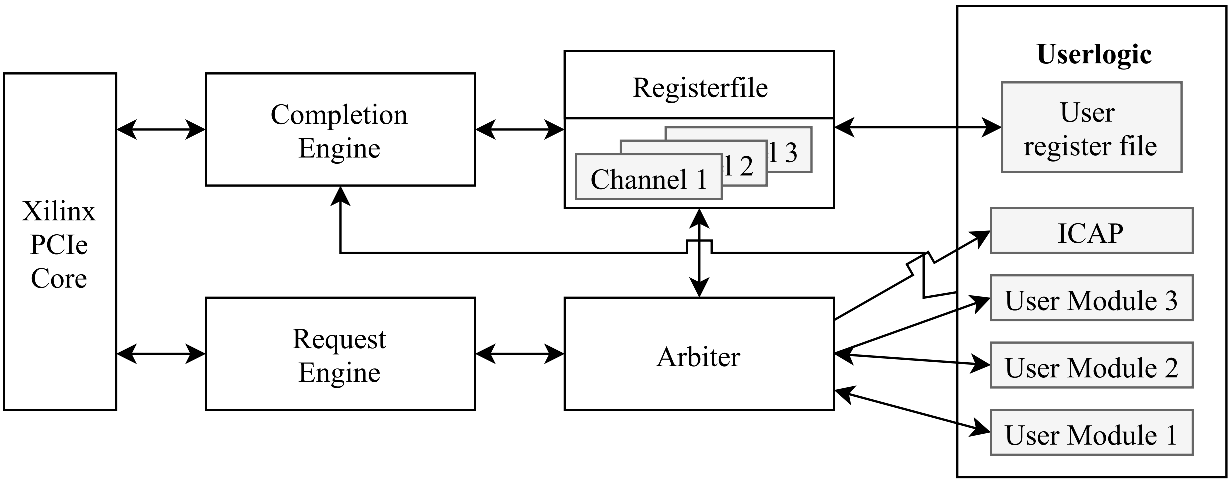


Figure 3.2: System Overview of the JetStream Library [VKVF16]

Figure 3.2 illustrates the hardware overview of the library. JetStream uses the Xilinx PCIe IP core as a start point of the communication library. The IP core has four AXI interfaces for data transfer, where one pair is connected to the Completion Engine and the other pair one to the Request Engine. The Completer Engine handles transactions initiated by the host and the Request Engine handles FPGA-to-host or FPGA-to-FPGA transactions. The engines are used to create/decode the headers and to align the 32-bit data words to 256-bit, which is the size of the other interface.

The Registerfile module is used to program the size of DMA transfers. The number of channels, which can be used, is parameterisable. The driver can automatically configure itself by reading the available channels from the Registerfile module. Each channel contains of a register group and a FIFO, which serves the driver the ability to issue multiple commands on one channel. A completion counter is used to give the host system the ability to know how many commands have been completed. The user cannot add user defined registers to the Registerfile module, but the Registerfile module serves an interface, where the user can define an address to access the user register files located in the user logic module.

The Arbiter module handles valid commands to the corresponding User Module. The arbiter is split into a receiving arbiter and a sending arbiter. Each of them has a command and an arbitration logic. The command logic setups a transfer by looking at the commands stored in the Registerfile and the available data. The module also splits the data into multiple packages, if the data size is greater than the possible payload size of one PCIe request size.

The User Modules can be fully customised by the user. The developer has to make sure, that the module is able to receive or create at least a maximum PCI Express payload sized packet at wire speed. The library provides a parameterisable buffer FIFO for easy implementation.

The host system must have always sufficient memory allocated to be able to receive data from the hardware. This is not always the case for FPGA-to-FPGA communication. To proof the availability of data storage, the sending FPGA is transmitting a request to the FPGA, which contains the amount of data he wants to send. The arbiter of the receiving FPGA ensures, that the data sending is only triggered, if enough memory is available. The line between the Completion Engine and the User Module illustrates a sink channel for direct FPGA-to-FPGA communication.

Software

The software of the JetStream library is split in two parts. First of all, the library has a device driver for the FPGA. The second part is a C++ API for initiating data transfer. The API supports three transfer modes (Figure 3.3). The three modes differ in size, complexity and speed.

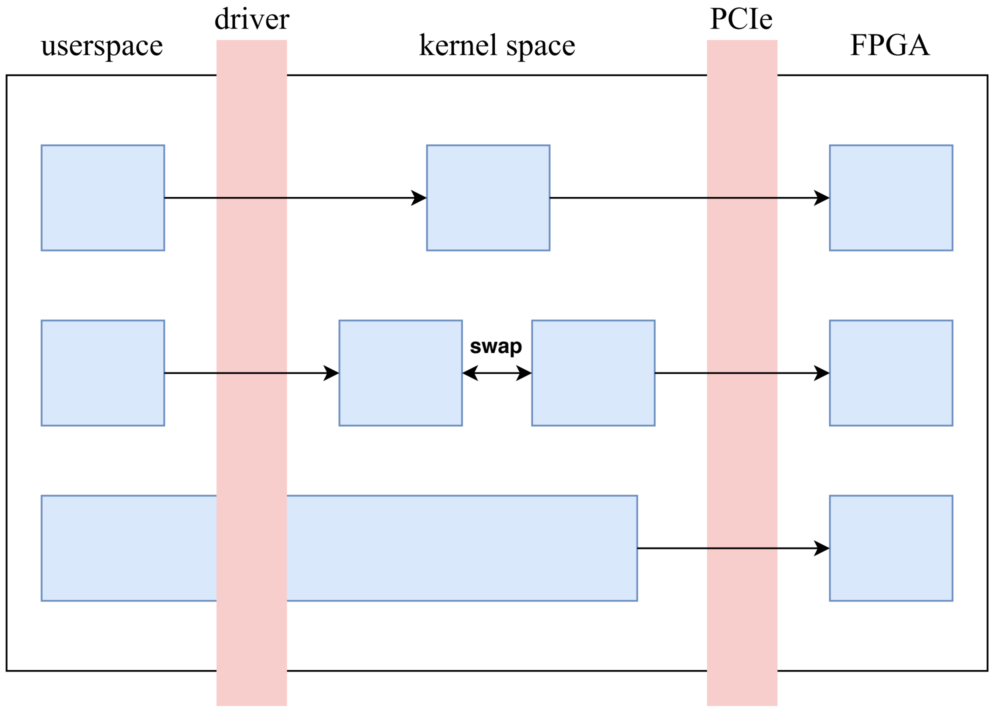


Figure 3.3: Three transfer modes for handling data between host and FPGA [VKVF16]

The single buffering mode (top figure of Figure 3.3) is the simplest mode. The driver gets a pointer to data in the memory space. The driver allocates FPGA accessible memory and copies the data to this memory. After that, the FPGA gets a request, that new data for a specific channel is available. This transfer mode can be configured for both directions. This buffering method needs an additional copy of data between the user and the kernel space. The user space or the FPGA is only able to access data, if all of the data is transferred.

This is not the case for the double buffering transfer mode (middle figure of Figure 3.3). The driver is copying data from memory allocated in the user space to a kernel buffer. Additional to the buffer between the user space and the kernel space, it has a buffer for making data available from the kernel space to the FPGA. This enables the driver to copy data into the first buffer, while the FPGA is reading from the second buffer, and vice versa for FPGA-to-host communication. This transfer mode speeds the data transfer up, but also doubles the memory usage.

The last transfer mode (bottom figure of Figure 3.3) avoids the need of an extra data copy between the data memory and the buffer. The Zero copy DMA mode offers the user space program a memory accessible by the FPGA. This avoids an additional copying mechanism and also achieves the highest data throughput.

Example Code

```
1  int main() {
2    //allocate the next available FPGA
3    FPGA fpga();
4    //allocate a buffer
5    Buffer buffer = fpga.malloc(1024);
6    // fill the buffer with data
7    // ...
8    {
9      //lock the FPGA
10     Transaction fpga.transaction();
11     //non blocking write request command
12     fpga.channel[0].send(buffer);
13     // ...
14     //blocking read request command
15     Operation receive =
16       fpga.channel[1].syncReceive(buffer);
17   }
18   //receive.sync();
19 }
```

Listing 3.2: Example code for transferring data using Zero copy DMA [VKVF16]

Listing 3.2 illustrates the code for transferring data between user space and an FPGA. The creation of an FPGA object, is the starting point of every program. This FPGA object references a physical FPGA. Line 5 allocates memory for the data transfer. The command is used to allocate memory for the Zero copy DMA memory transfer. This means, that the memory is accessible from the user space and also from the FPGA side. After that, the buffer has to be filled with data. In this example, the FPGA is doing a specific task and the outcome is read out afterward. To make sure, that the program is the only program accessing the FPGA, the FPGA get locked (line 9). A write request is sent to the channel 0 of the locked FPGA. The FPGA is now reading the data from the memory asynchronously. The send command is a non-blocking C++ command. The program could now perform some other tasks. For receiving data, a non-blocking and a blocking option is available. The command on line 16 is used to do a blocking read on the channel 1 of the FPGA. If the command `syncReceive` is replaced by `receive`, the receive command would also be asynchronous and the data would be written from the FPGA to the host into the memory, described by the buffer object. Line 18 would then wait for reading the data.

3.2 Data Handling in Network Interface Cards

The growing number of connected devices and the applications, which use internet services, makes it necessary to implement efficient and fast communication interfaces. Many services use battery driven mobile devices, where processing time should be reduced to a minimum. The reduction of processing time is in conflicts with the growing amount of data and the growing link speed of network interfaces.

The rising link speed of Network Interface Cards (NICs) needs developments in the data handling concept for data transmission. The communication between the CPU and the NIC has to be improved, to support higher link speeds. The functionality of a communication channel has a software and a hardware part. The software part can be implemented as application or as device driver of the operating system.

This section describes the basic concept of data handling between a host and a NIC and two different implementations. The SoC board WR-ZEN represents consists illustrates the implementation of a communication interface for SoCs and the Oregano Systems syn1588® PCIe NIC illustrates the data handling in a consumer Ethernet network card.

3.2.1 Basic Concept

Network cards are more complex regarding in data handling compared to other peripheral units. The data transmission of the system is never deterministic, therefore the NIC has to be always ready for sending and receiving data. After a packet has been transmitted or received, the system must be informed to do some ongoing tasks.

The data handling between the host and the NIC can be split into two parts. The first part is named as the buffer descriptor exchange. A buffer descriptor contains the memory location of the packet data and some other information used for exchanging data between the host and a NIC. The host transfers this data at first, to configure the transmission. The second part is the packet data transmission from and to the FPGA.

In Section 2.3, different forms of I/O operations were introduced for communicating with peripheral units. The network interface card can use multiple. Steen Larsen and Ben Lee [LL14] described a possible sequence of operations for sending and receiving a packet using DMA controlled transmission.

Ethernet Packet Transmission

The transmission of a packet from a host to the NIC needs 8 steps:

1. The kernel or the driver of the system creates the outgoing packet and saves it to the system memory. The packet needs header information, sequence number and checksums to support the protocol stack like Transmission Control Protocol (TCP) and Internet Protocol (TCP/IP)
2. The NIC has to be informed, that new data is available for transmission. This is done by a doorbell request on the bus. This is the last thing the host does for the packet transmission. It will wait to release the memory buffers until the NIC notifies the transmission of the packet.
3. A DMA request for reading the physical address of the sending payload is triggered.

4. The read request returns with multiple descriptors, containing the physical addresses of the header and the payload. After that, the NIC requests the header information of the packet.

5. Next step is to transmit the payload from the memory to the transmit buffer of the NIC.

6. If the payload data and the header are available, the NIC combines it to an Ethernet frame with the correct ordering

7. At last, the bitstream is passed to the physical layer (PHY), which converts it into the proper signal condition of the medium

8. After transmitting the packet, the host gets informed by an interrupt. The system can now free the allocated memory.

Ethernet Packet Receiving

The receiving of data and the transmission to the host needs the following 6 steps:

1. The NIC needs a descriptor for storing a packet in the system memory. The host has allocated memory for storing data and has stored the location of the memory in the buffer descriptor. The NIC fetches the descriptor from the system memory

2. The NIC receives a packet from the physical layer (PHY) asynchronously.

3. A DMA write transfer is triggered to store the received packet to the memory location which is stored in the descriptor.

4. The NIC interrupts the host after transmitting the complete packet to the memory.

5. The host is processing the interrupt and writes the next location of a descriptor to the NIC. This action also confirms the packet receiving by the host

6. The host system can know process the received packet.

3.2.2 WR-ZEN Board

The first System on Chip (SoC) implementation of a White Rabbit sub-nanosecond synchronised communication is built in the WR-ZEN board [SGLAJLD17]. The board consists of a Zynq-7000 SoC, which has a dual-core ARM Cortex-A9 Processor System (PS) and a Programmable Logic (PL) part with Artix-FPGA logic. The original developed board uses the NIC modules with regular Ethernet data traffic and White Rabbit timing distribution on the same link. This simplifies the network infrastructure, the deployment and the maintenance of the system.

The link speed of the NIC supports 1Gbps. Due to not optimised data handling between the PS and the PL, the board was not able to utilise the full link speed. The paper [SGLAJLD17] describes the changes for optimising the data throughput.

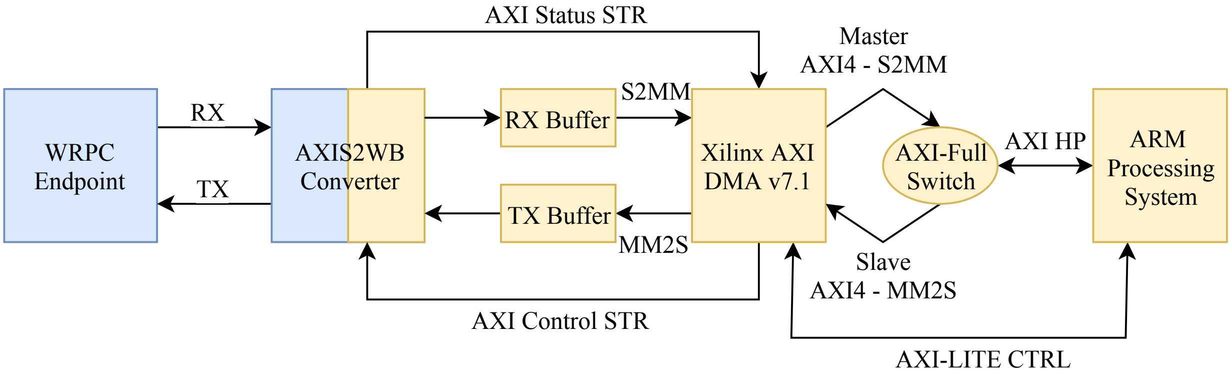


Figure 3.4: Block diagram of the WR-ZEN board hardware [SGLAJLD17]

Hardware

Figure 3.4 illustrates the optimised hardware components of the WR-ZEN board. The components for transferring the data from the PS, where a Linux is running, are the following:

- **Xilinx AXI DMA IP Core:** The Advanced eXtensible Interface (AXI) DMA core of Xilinx provides high-bandwidth access directly to the memory. The AXI4 Stream to Memory Mapped (S2MM) slave interface is used for writing data to the memory and the AXI4 Memory Mapped to Stream (MM2S) master interface interfaces is used for reading data from the memory. The DMA and the Scatter/Gather engine for fetching buffer descriptors reduce the CPU usage significantly. The DMA IP core provides high-speed data rates and can be seen as major component for improving the data rate.

- **AXI-Streaming to Wishbone Fabric Converter (AXIS2WB Converter):** The White Rabbit PTP Core (WRPC) uses the open source bus Wishbone Pipeline Fabric for data exchange. This block converts the AXI signals communing from the Xilinx DMA controller into the Wishbone Fabric bus. This converter is also responsible for communicating management configuration options for each transmission. It also ensures, that each Ethernet packet reaches the minimum frame length.

- **White Rabbit PTP Core (WRPC):** The real time White Rabbit stack is implemented in this component. The Wishbone Pipelined Fabric bus is used at one side and on the other side the Ethernet PHY of the WR-ZEN board is connected through its Ethernet MAC implementation. The architecture allows using a generic NIC with White Rabbit synchronisation.

- **Additional AMBA AXI Bus Fabric:** This bus connection is used for accessing the main memory of the PS. The high speed AXI4 connection enables fast data movements and therefore maximises the data transfer throughput.

- **AXI-Streaming Buffering:** Two FIFOs, one for receiving and one for sending data are used to buffer transmission. The 16k-word FIFOs enable the system to transfer Ethernet jumbo packets and avoid data losses, while operating at high data rates.

Software

The hardened ARM PS core of the SoC is used for running an embedded Linux. The software, for communicating with the NIC, is built as a network device driver of the Linux system. The NIC device driver has to be updated, to support the new DMA-based hardware. Figure 3.5 illustrates the components of the driver.

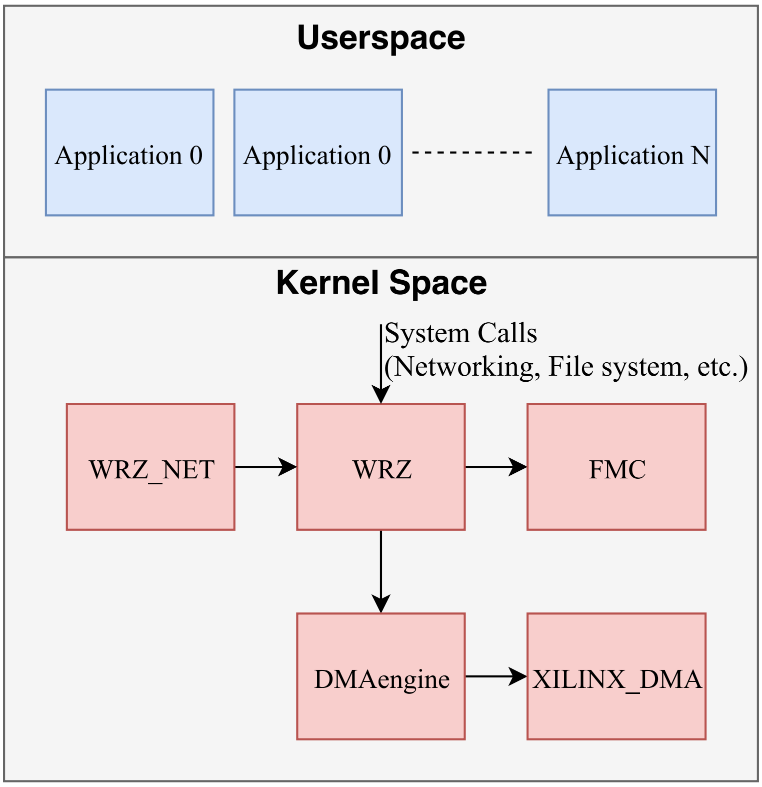


Figure 3.5: Linux Network Driver for DMA-based hardware. [SGLAJLD17]

- **WRZ (Main Module):** It is used to configure the WR-ZEN board, setups the network interface and manages the user-space communication.
- **WRZ_NET:** Communicates with the main network and enables network functionality
- **FPGA Mezzanine Card (FMC):** Different FMC extension cards can be used by the WR-ZEN board. This part enables the usage of them.
- **DMAengine:** Abstraction layer for accessing DMA controllers.
- **XILINX_DMA:** Module for setting up data transfer. The configuration and the access of the Xilinx DMA is done through the generic interfaces of the DMAengine

The driver uses two descriptor pools for data handling, one for receiving and one for transmission. After the network interface has received a packet, the data allocation for a free receive buffer descriptor is done. After that, the data is transferred from the board to the main memory of the system. The end of the transmission is signalled by an interrupt triggered by the DMA module to the PS. The driver will then make the received packet available to the operating system.

The data handling for sending packets is also done by buffer descriptors. The hardware loads the data specified in the buffer descriptor and sends it to the network interface. An interrupt is triggered by the DMA module, if the packet is sent. The driver can then free the buffer descriptor and the packet data memory.

The interrupt handling, for notifying the driver that data has been received or transmitted, was designed two times. The architecture of the first design had to send an interrupt for every sent and received packet. Because the first approach led to a bandwidth inefficient, the interrupt triggering was bundled. The second approach let the hardware module wait until N packets were sent or received. This approach works great for high bandwidth utilisation, but has to be completed by a timeout mechanism, if only a few packets were transferred. If N packets were sent or a specific counter value has been exceeded, the hardware has to send an interrupt to the PS.

3.2.3 Oregano Systems syn1588® PCIe NIC

The syn1588® PCIe NIC is a 1-lane PCI Express Ethernet card with the supported link speeds of 10Mbps, 100Mbps and 1Gbps. The special feature of the card is a patented technology for on-the-fly timestamping of sent and received packets. Additionally, the card is fully compliant to the time synchronisation standards IEEE1588-2002 and IEEE1588-2008. It includes a patented hardware clock for serving highly accurate timestamps.

The functionality of the data handling is specified in a not publicly available document. The specification document and the source code of the hardware and software implementation were provided by Oregano Systems.

Buffer Descriptor

The packet transfer between the host system and the NIC is split in 2 parts. Before the payload of the packet can be transmitted, the NIC needs a buffer descriptor. The buffer descriptor for receiving and transmitting packets with the syn1588® NIC includes the 32-bit entries described in Table 3.1 and 3.2.

Address	Description
0	Receive Buffer Descriptor Control/ Status Word
1	RX BD Address Word
2	Reserved
3	Reserved

Table 3.1: Entries of a Receive Buffer Descriptor

Address	Description
0	Transmit Buffer Descriptor Control/ Status Word
1	TX BD Address Word
2	Timestamp Nanoseconds
3	Timestamp Seconds

Table 3.2: Entries of a Transmit Buffer Descriptor

The Receive Buffer Descriptor (RX BD) consists of the RX BD Control/ Status Word and the Memory Pointer, of the data. The RX BD Control/Status Word includes several transmission information described in Table 3.3. The important information for data handling is stored in bits 31:16 and bit 15. The NIC needs a free RX BD to move the data from the NIC to the host. A free BD is marked with bit 15. The empty bit is cleared by the host and must be set by the NIC, if the received data of a packet is stored in the RX BD Address Word entry pointed memory. The normal packet length of an Ethernet packet can be between 42 and 1500 bytes. Therefore, the host has to allocate at least 1500 bytes for the packet data in the memory. Data centers often use bigger Ethernet packet sizes, so called jumbo packets, to improve the throughput. Therefore, the packet length information, which is stored in bits 31 to 16, can store greater values than 1500. Bit 10 informs the NIC, if an interrupt has to be sent to the host to inform, that a packet was successfully sent. If the packet includes a timestamp, the 6th bit of the RX BD Control/Status Word is set. Bits 0 to 5 and bits 7 to 9 are storing transmission information of the packet.

Bit	**#**	**Description**
31:16	RW	Number of the received bytes associated, with this BD.
15	RW	Empty 0 = BD has valid data 1 = BD data is empty
14:11	RW	reserved
10	RW	Interrupt Request Enable, 0 = No interrupt is generated after the reception. 1 = When data is received (or error occurs), an RXF interrupt will be asserted
9	RW	VLAN RX packet: 0 = no VLAN packet; 1 = VLAN packet
8	RW	Control Frame: 0 = Normal data frame; 1 = Control frame
7	RW	Miss (promiscuous mode only): 0 = Address recognition hit; 1 = Promiscuous mode
6	RW	Packet with timestamp: 0 = no timestamp; 1 = 64-bit timestamp appended
5	RW	Invalid Symbol (promiscuous mode only): Bit is set when the reception of an invalid symbol is detected by the PHY
4	RW	Dribble Nibble (promiscuous mode only): Bit is set when a received frame cannot be divided by eight
3	RW	Long Frame Error (promiscuous mode only): Bit is set when a frame larger than the maximum length is received
2	RW	Short Frame (promiscuous mode only): Bit is set when a frame smaller than the minimum length is received
1	RW	Rx CRC Error (promiscuous mode only): Bit is set when a received frame contains a CRC error.
0	RW	Late Collision (promiscuous mode only): Bit is set when a late collision occurred while receiving a frame

Table 3.3: Receive Buffer Descriptor Control/Status Word bit definition

The Transmit Buffer Descriptor (TX BD) consists of the TX BD Control/Status Word, the TX BD Address Word and the seconds and nanoseconds timestamp word. The entries are stored as 32-bit words. The TX BD Control/Status Word entries are illustrated in Table 3.4. The TX BD Ready information, located at the 15th bit of the TX BD Control/Status Word signals the NIC, that the packet is ready for transmission. If the TX BD is processed by the NIC, the 11th bit is set. Bits 0 to 7 are storing information about the packet transmission.

Hardware

Figure 3.6 illustrates the functional module blocks of the NIC. The main logical functionality of the NIC is implemented in an FPGA. The translation of the Ethernet signals is done by

Bit	**#**	**Description**
31:27	RW	Reserved for internal usage: BD RAM Address [7:3]
26:16	RW	Number of the received bytes associated, with this BD.
15	RW	TX BD Ready 0 = BD is not ready and payload data can manipulated 1 = data buffer is ready for transmission
14:12	RW	Reserved for internal usage: BD RAM Address [2:0]
11	RW	BD used
10	RW	Interrupt Request Enable 0 = No interrupt after transmission 1 = interrupt after transmission
9	R	reserved for internal usage
8	R	TX packet has received a timestamp
7:4	RW	TX Retry Count Indicate the number of retries before successfully sending.
3	RW	Retransmission Limit This bit is set when the transmission of a packets fails due to repeated collisions on the medium
2	RW	Late Collision Late collision is any collision after the 64th data byte.
1	RW	Defer Indication The frame was deferred before being sent successfully.
0	RW	Carrier Sense Lost Carrier Sense was lost during a frame transmission

Table 3.4: Transmit Buffer Descriptor Control/Status Word bit definition

the Physical Layer Interface. The syn1588® Clock_M Module is connected to a highly accurate oscillator, which is used as clock source.

The communication between the host system, the main memory and the NIC is done by a 1-lane PCI Express version 2.0 bus. The data handling functionality between the host and the NIC is implemented by the 10/100/1000Mbps MAC IP core. The other modules don't affect the implementation of the data transmission between the host and the NIC.

Figure 3.7 gives a more detailed view of the model used for data transmission. The data handling procedure needs the PCI Express IP Core, a module called AHB Slave, a memory for storing the buffer descriptors, two modules which converts the Avalon-MM bus into a AHB bus, a module called AHB Master and two modules for sending (TX MAC) and receiving (RX MAC) the data from the network.

The PCIe IP core has an Avalon-MM Slave and an Avalon-MM Master interface implemented. The master interface is used by the CPU to access registers of the ABH Slave module and to access the Buffer Descriptor Memory. The host system can read and write buffer descriptors for receiving and sending packets. The AHB Master uses the Avalon-MM Master interface to trigger read or write actions on the PCIe bus.

The ABH Slave module is used to setup the NIC controller, to read data registers and to access the buffer descriptor memory. The configuration registers of the NIC are located in the AHB Slave module. The Clock IP as well as the AHB Master have access to the registers of the AHB

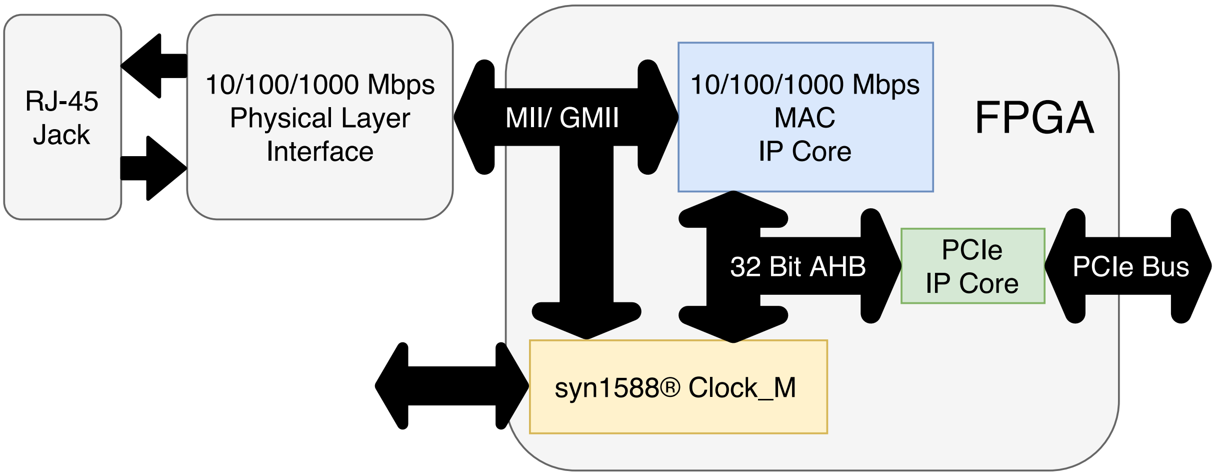


Figure 3.6: Block diagram of Oregano Systems syn1588® PCIe NIC [Cad17]

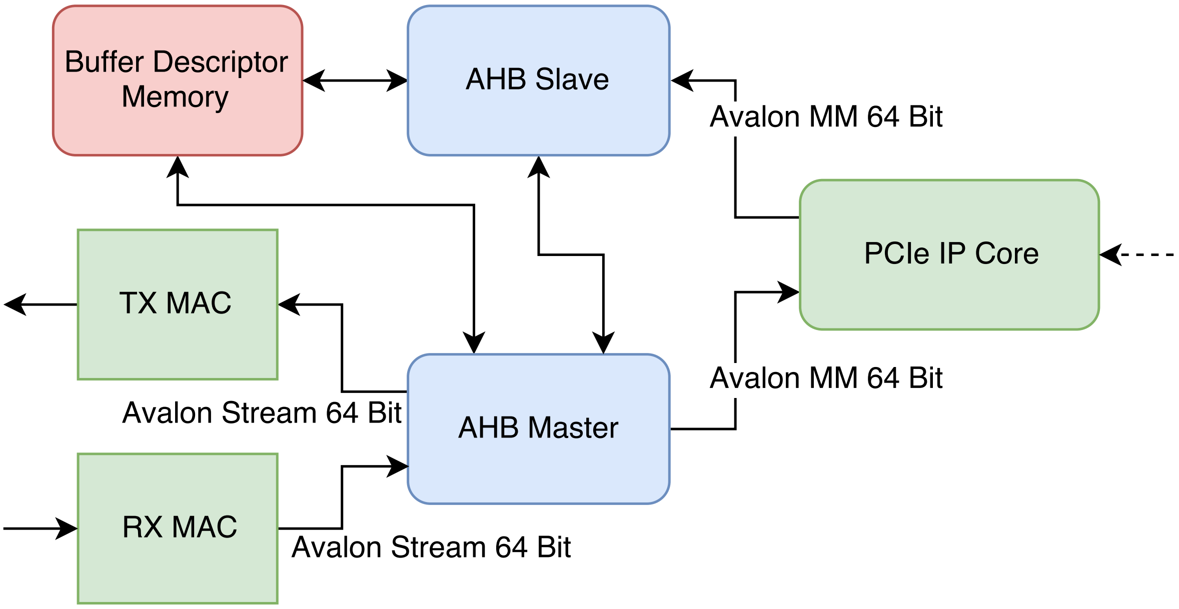


Figure 3.7: Components necessary for data handling

Slave. The host system is able to access the buffer descriptor memory over the AHB Slave to read and write buffer descriptors.

The Buffer Descriptor Memory stores the receive and transmit buffer descriptors. The memory is implemented as a 9 x 32-bit dual ported ram with a 9-bit address signal and 32-bit data lines for input and output. The AHB Slave is connected to port A and the AHB Master is connected to port B. Each receive and transmit BD needs 128 bits or 4 x 32-bit. The memory can store 64 RX BD and 64 TX BD. The buffer descriptors are separated in two sections. Address $0x000$ to $0x0FF$ $(0 \text{ to } 64 * 4 - 1)$ are used to save the entries of TX BD and the Addresses $0x100$ to $0x1FF$ $(64 * 4 \text{ to } 128 * 4 - 1)$ are used to store the RX BD entries. Bits 1:0 are describing, which entry of the BD gets accessed (see Table 3.1 and 3.2).

The AHB Master controls the data transmission between the memory and the NIC as well as the signalling for new received or sent packets. The AHB Master is connected to port B of the Buffer

Descriptor Memory and to the registers of the AHB Slave. The RX MAC and the TX MAC are connected with an Avalon Streaming Bus interfaces. The AHB Master is connected to the RX MAC and the TX MAC, as well as to the PCI Express IP Core.

The functionality of the AHB Master is separated in two processes. The first process handles the BD. It continuously searches for new BD and stores ready (TX BD) or empty (RX BD) buffer descriptors in the local register. The process also updates buffer descriptors of sent or received packets. The BD Control/Status Word for receiving and transmitting packets include a bit, which triggers the AHB Master to send an interrupt to the host system after updating. The module is accessing the Buffer Descriptor Memory, if the other process does not have an empty or ready buffer descriptor. The Buffer Descriptor Memory is also accessed from the AHB Slave, which has priority over the AHB Master.

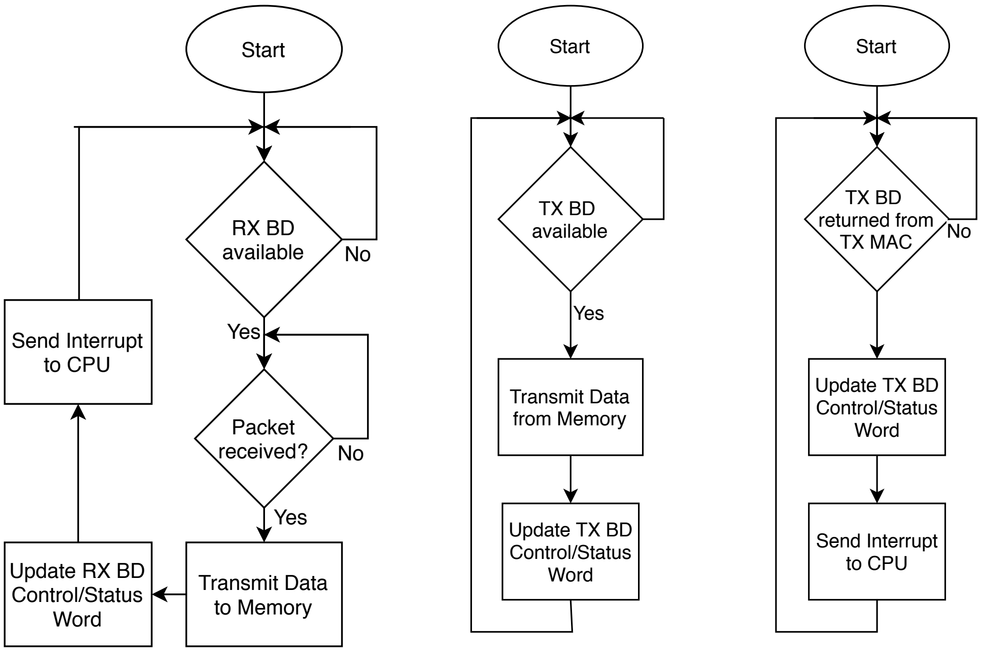
The second process is handling the packet data transfer between the host and the TX MAC and the RX MAC. For receiving data, the process must have an empty BD available and the valid signal of the RX MAC Avalon Stream interface has to be set (backpressure transfer mode). The first 64 bits of the transmission includes the RX BD Control/Status word of the received packet. The process can then initialise a transfer from the RX MAC to the location in the memory, specified in the BD entry 2, the RX BD Address Word. The transmission of data is activated, if the first process finds a ready TX BD in the Buffer Descriptor Memory and the Avalon Stream sink interface of the TX MAC has its ready signal set. The process initialises a transfer from the memory location, specified in the BD entry 2, the TX BD Address Word, to the TX MAC. The first 64 bits, which are transferred, contain the TX BD Control/Status Word of the packet. The address of the TX BD in the Buffer Descriptor Memory is stored in bits 31:27 (address 7:3) and bits 14:12 (address 2:0). After transferring the data to the TX MAC, the 11th bit of the BD Control/Status Word has to be set and the BD entry has to be updated in the Buffer Descriptor Memory. After the TX MAC has sent the packet data over the network, the TX MAC signals the AHB Master, that the transmission was done. Furthermore, the process updates the TX BD entries with the address located in bits 31:27 and 14:12 of the TX BD Control/Status Word (see Table 3.2). The update also includes the setting of the timestamps (TX BD entries 2 and 3).

Figure 3.8 illustrates the sending and receiving control flow of the AHB Master module. The hardware is continuously searching for available BD in the Buffer Descriptor Memory. If a valid empty RX BD is available (Figure 3.8(a)), the data of the RX BD is stored in internal registers and the hardware waits, until a packet is received. The received packet is copied to the memory address specified in the RX BD Address Word. After that the RX BD Control/Status Word is updated and an Interrupt is sent to the CPU.

The sending control flow (Figure 3.8(b)) transmits the data specified in the TX BD into the TX MAC Module. The TX BD is then updated and a new TX BD can be searched. The TX MAC returns the TX BD Control/Status Word after sending a packet to the AHB Master module. The returned TX BD Control/Status Word includes its location in the Buffer Descriptor Memory in bits 31:27 and 14:12 (see Table 3.2). The returned TX BD Control/Status Word is updated in the memory and an interrupt is sent to the CPU.

Software

The software of the NIC is implemented as a device driver of the operating system. The main task of the software is to setup the NIC, allocate memory for packet transmission and handle received data from the NIC.


Start
RX BD available
No
Yes
Packet received?
No
Yes
Transmit Data to Memory
Update RX BD Control/Status Word
Send Interrupt to CPU
Start
TX BD available
Yes
Transmit Data from Memory
Update TX BD Control/Status Word
Start
TX BD returned from TX MAC
No
Update TX BD Control/Status Word
Send Interrupt to CPU

(a) Receiving packets (b) Transmitting packets

Figure 3.8: Data handling control flow of the Oregano Systems syn1588® PCIe NIC

During the setup process, the driver has to initialise the BD for receiving and transmitting. This includes setting up the configuration registers of the NIC, allocating memory for packets, initialising the BD Control/Status Words with default values and storing the addresses of the packet memories in BD Address Words of the BD. The allocation of the memory in the main memory for storing the packet data is done at the beginning, to reduce the time for allocating and saving the addresses to the Buffer Descriptor Memory.

The driver gets activated for transmitting a packet, if the operating system triggers it. The driver uses two pointers for knowing, if a BD is free. The first pointers (write pointer) saves the index of the last written BD and the second pointer (read pointer) stores the last BD, which was sent by the NIC. After incrementing the write pointer, it gets compared with the read pointer. If the values are equal, no BD is available for transmitting packets. Otherwise the packet data can be transferred to the location, which is stored in the TX BD Address Word, and the TX BD Control/Status Word is set. Afterwards, the TX BD Control/Status Word has to be written to the NIC. The TX BD Address Word value of a TX BD is fixed after initialisation, so it doesn't have to be updated.

The NIC is sending an interrupt, if a packet was sent and the interrupt bit (10th bit) of the TX BD Control/Status Word is set. The TX BD Control/Status Word was updated by the NIC and was marked as not ready. The read pointer can be incremented to mark the BD as available for a new packet.

The processing of received data is triggered by the NIC. After transmitting the data from the NIC to the main memory, an interrupt is sent over the PCIe to the host. The driver, which stores the pointer to the next RX BD, reads the RX BD Control/Status Word from the Buffer Descriptor Memory of the NIC. The RX BD Address Word can be calculated out of the starting address of the allocated memory and the RX BD pointer stored. The data is moved from the location stored in the RX BD Address Word to a memory, which is accessible by the operating system for further processing. After that, the RX BD Control/Status Word is set to an initial value and is written to the Buffer Descriptor Memory of the NIC.

Control Flow for Packet Sending

The control flow for sending a packet includes the following steps:

1. The operating system forwards some data to the driver for sending on the network interface.
2. The driver search for a free TX BD. Afterwards, the packet data is copied to the memory location of the TX BD. The address of the location is stored in the TX BD Address Word.
3. The TX BD is then written back to the Buffer Descriptor Memory of the NIC
4. The hardware is continuously searching for new TX BD. If a new TX BD is found, the data of it is stored in local registers.
5. The packet data stored at the location specified in the TX BD Address Word is transferred to the TX MAC.
6. The TX MAC returns the TX BD Control/Status Word of the packet. The TX BD Control/Status word contains the Buffer Descriptor Memory address of the TX BD in the bits 31:27 and bits 14:12 (see Table 3.4). The TX BD gets updated and an interrupt is triggered to the CPU
7. The CPU reads the TX BD and signals the operating system, that the packet was sent successfully.

The software executes the steps 1 to 4 and 8 while the others are processed by the hardware.

Control Flow for Packet Receiving

The control flow for receiving a packet includes the following steps:

1. The hardware is continuously searching for free RX BD. If an empty RX BD is found, the data of it is stored in a local register.
2. The AHB Master is now sensing on the Avalon Streaming bus interface of the RX MAC. The RX MAC signals a received packet by setting the valid and start of packet signal. The first 64 bits of data contains the RX BD Control/Status Word of the received packet. It is used to configure the data transfer to the memory location specified in the RX BD Address Word entry specified in the RX BD.
3. After sending packet data to the memory, the RX BD is saved to the Buffer Descriptor memory and an interrupt to the CPU is sent.
4. The CPU reads the RX BD and signals the operating system, that a packet was received.
5. After that, the RX BD is reset to an initial value.

The steps 1 to 3 are executed by the hardware and step 4 and 5 are handled by the software.

3.3 Performance Analysis of Data Handling Components

Multiple parameters influence the performance of data transfer between multiple components of connected components of a device. Figure 3.9 gives a graphical representation of a system bus with multiple components.

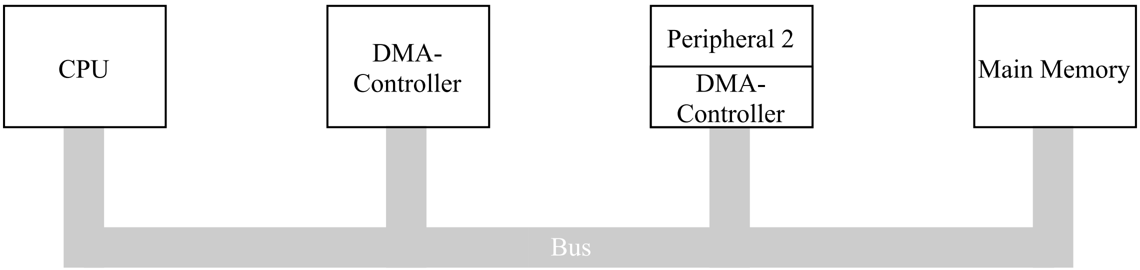


Figure 3.9: Multiple connected components which use the same bus for data exchange

The communication between components is done with the help of a bus system. Multiple components are connected and they share the same bus for data transfer. The CPU and the components are exchanging their data with the help of the main memory of the system. Fast communication and processing should not mean that the processor is heavy utilised or slowed down. This is the reason, why data handling components like DMA controllers can be used to outsource data moving mechanisms from the CPU to another component. Peripherals could also have a DMA controller implemented to speed data communication up.

The following chapter discusses the performance impact of the memory and bus protocols and presents the important points for getting a throughput optimised data handling concept.

3.3.1 Memory

Memory is an expensive part of a component. It is available on-chip and off-chip. On-chip memory of an FPGA or an ASIC is limited to a few megabytes. Therefore, additional memory is added by putting memory chips on the Printed Circuit Board (PCB). The complexity of a memory managing unit integrated on an FPGA or an ASIC is very high and also its development costs are very high. Digital circuit designers try to avoid adding additional memory blocks to hardware by storing needed data on the main memory of the system.

Modern computer or SoC systems have multiple gigabytes of main memory available. It is used to store application data and for transferring data from CPU to peripheral units. The performance of memory intensive applications highly depends on the speed of the main memory.

Figure 3.10 illustrates the 3D - architecture of a Dynamic Random Access Memory (DRAM) chip [RDK$^+$00]. The three dimensions of the memory are represented with banks, rows and columns. A bank has an array of memory cells which accesses an entire row at a time. The data of a row is stored in a cache after if it was accessed. This reduces the latency of subsequent read or write actions on the same row.

Several parameters are influencing the performance of reading or writing data from the memory:

- Clock frequency of the memory.

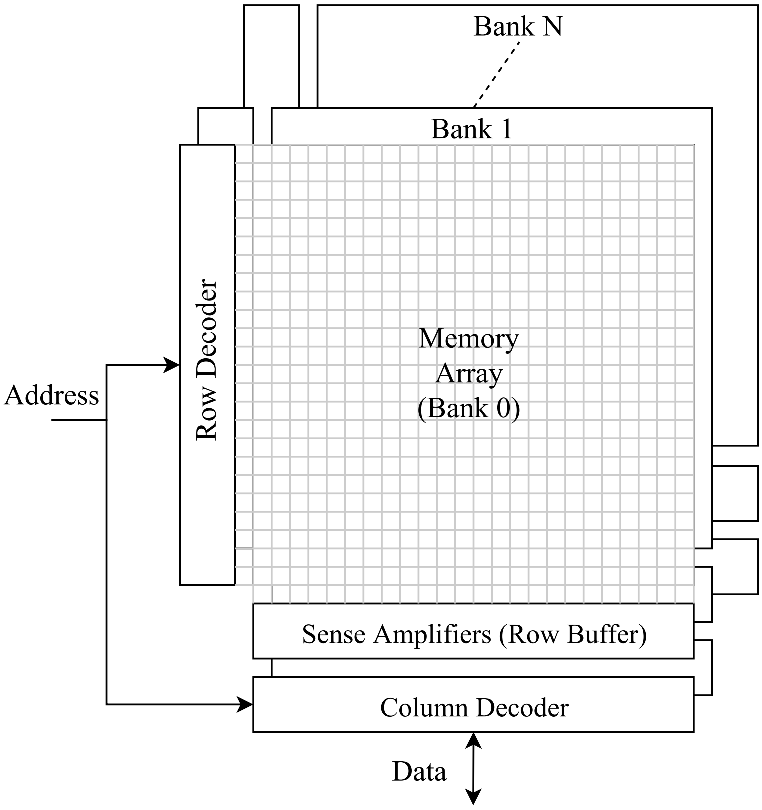


Figure 3.10: 3D - architecture of a Dynamic Random Access Memory chip

- Memory scheduling schemes [RDK$^+$00], which is responsible for reordering data fetching requests, increase the utilisation of the caches and therefore the data exchange gets faster.
- Organisation of stored data.

The last point of the performance influencing parameters is addressed to the application engineer. The data fetching from the memory can be optimised, if the data are consecutively stored in the memory. The memory can be fully utilised for transferring the requested data. If multiple requests to different addresses have to be processed by the memory controller, the performance will be decreased.

3.3.2 Bus Protocols

Different bus protocols can be used to enable communication between components. Multiple bus protocols exist for on-chip and off-chip communication. Memory-Mapped (MM) bus protocols, which are able to address components or memory locations can be used to transfer data from the main memory of a system to the CPU or other peripheral units.

On-Chip Bus Protocols

AMBA AXI-MM and Avalon-MM bus protocol 2.4.2 can be used to transfer data between the main memory of a SoC to peripherals or the processing system. A bus system uses dedicated signals for transferring addressing information and for transferring the data. A MM bus protocol has different methods for transferring data.

- A simple read/write transfer: This action is used for getting only one data word of a memory. The address signal refers to a byte, but will transfer data starting from the address specified to the highest address possible to send over the data signal. Data widths of on chip bus systems can be 8, 16, 32, 64, 128 or 256 bits.

  This transfer mode should not be used to transfer a great amount of data. The utilisation is reduced, because the master has to get access to the bus, before he can request a new transaction. If multiple components use the bus, it has to wait until it gets control over the bus again.
- Pipeline transfer: Multiple requests on different addresses can be triggered from the component or the CPU and the memory responses in-order. This transfer mode can be used, if multiple data, which is not stored consecutively, should be transferred.
- Burst transfer: The master can request a read or write transaction for multiple data words. The starting address is specified by the address signal and a separate burst length is transferred with a signal, which describes the amount of transported data.

The last transfer method is the preferred one for transferring multiple data words. The application developer has to allocate data blocks, so that the burst action can be used. The simple and the pipelined transfer method will reduce the throughput on the bus. The arbiter of the bus system will not give the same component access to the bus multiple times in a row. The CPU and the DMA module of the system will have priority over other components.

Off-Chip Bus Protocols

The PCI Express bus (Section 2.5.1) is the most used off-chip bus protocol used for high speed communication between the main memory and a peripheral unit. The packet based point-to-point transfer supports different maximum payload sizes. Six different maximum payload sizes exist: 128, 256, 512, 1024, 2048 and 4096 bytes.

The bus request is fully utilised, if the maximum packet size can be used for reading or writing requests. The internal buffers of the system should be chosen for supporting at least a payload size of 128 bytes.

3.3.3 Data Handling

Transporting data between a CPU and peripheral units includes several steps and used components. The performance of an application running on a peripheral component highly depends on fast data exchange.

A data handling mechanism, optimised for data throughput, should consider the following points:

- **Use memory blocks**: Write and read actions on the system memory can be optimised by accessing coherent memory blocks. The buffer of the memory can then be utilised efficiently.

- **Use separate DMA controller**: Handling data from one location to another can be done by a separate DMA controller (see Section 2.3.3). It is important to unload simple procedures from the CPU to other controllers. This frees the CPU for more important and more difficult tasks.
- **Consider maximum data size of bus systems**: Application engineers should try to utilise the whole data width of the bus. The read and write action should only be requested, if the size of the request is big enough to utilise the bus width.
- **Use as less bus requests as possible**: Using less bus requests and fetching multiple data points with one burst transaction will increase the data transfer. The arbiter of a bus protocol, where multiple components are accessing the same bus, will give faster access to a component, if the component uses fewer requests.
- **Use as less memory requests as possible**: Less memory requests mean less bus requests, which can be easily guaranteed, if the application memory is allocated in blocks
- **Minimise communication between CPU and peripheral unit**: The CPU will directly write to peripheral unit for configuring it. The DMA controller should transfer greater amount of data. The peripheral must also transfer notification to the CPU for signalling process completion. This notification can be sent by interrupts. After the CPU receives the interrupt, an action gets triggered to do some ongoing tasks. If these interrupts can be bundled to a few, so that the CPU needs less context switches, the performance of the system will increase.

3.3.4 Analysis of State of the Art Solutions

The two analysed methods for handling data to a network interface for transmitting data over the network use the same link speed for data transmission. The data handling solutions are working in different areas. The WR-ZEN Board is a SoC board, which is capable to process White Rabbit synchronisation. The NIC of Oregano Systems represents an interface which can be used in every host system with a PCI Express bus.

WR-ZEN Board

A working group (Jorge Sánchez-Garrido et al. [SGLAJLD17]) have already found problems, referring the data communication speed. The optimisation of the data handling is described in Section 3.2.2. The paper focuses on the impact of high data throughput with the synchronisation mechanism, but does not describe the maximum bandwidth of the system.

The possibilities for creating a complete new data handling strategy in a SoC are higher then for NICs. Jorge Sánchez-Garrido et al. use two bus systems to transfer data. The packet data are transferred with an AXI High Performance bus and the meta information is transmitted with the help of an AXI LITE interface.

The paper describes, that every time a packet arrives, memory has to be allocated in the main memory to store the data. Each received packet needs processor and memory actions for getting memory and for transferring the allocated address back to the memory. The memory's allocation can fail, because no memory is available. This can lead to packet drops. Also, the load of the memory is higher, because the allocation of memory for every packet needs memory and processing time. The allocation of memory and the meta data transfer, which is triggered by the processor, can be avoided by defining the memory for packet transfer during the activation of the network interface.

Oregano Systems syn1588® PCIe NIC

The Oregano Systems PCI Express NIC is connected to a host system with a PCI Express bus. Every time the CPU has to access the NIC, the PCI Express bus, which can be used by multiple peripherals, has to be used.

The most time-consuming process of the NIC is the writing and reading of meta information to and from the NIC. The meta information of a packet is represented with the help of buffer descriptors (BDs). The NIC has a memory block on the hardware, which is used to store the BDs. Every time the operating system has to send data, the device driver has to do the following actions:

- Copying the data from the application storage to a memory location, which is accessible by the NIC
- Writing the TX BD to the Buffer Descriptor Memory on the NIC
- Wait, until the NIC sends the packet and triggers an interrupt
- Reading the interrupt source from the NIC
- Reading the TX BD of the sent packet from the NIC

Sending a packet, needs one write and 2 read actions on the PCI Express bus. The transfer of a received packet needs the following actions:

- Wait, until the NIC triggers an interrupt
- Reading the interrupt source from the NIC
- Reading the RX BD of the received packet from the NIC
- Writing an initial value to the RX BD, when it was processed by the device driver

This action also needs one write and 2 read actions on the PCI Express bus. The PCI Express bus is not always available, so the time needed for reading and writing data is not deterministic. The amount of data, which is transferred each time, are four bytes. A transaction with only a few bytes does not fully utilize the bus.

4 Optimised Data Handling Architecture for Network Interface Card

The fourth industrial revolution, where every device should be accessible via internet, demands on high speed communication. The costs, energy efficiency and the performance of the used devices have to be optimised for being competitive on the market.

The amount of data, which has to be transferred through the network, grows continuously. This leads to rising link speeds, which are supported by different communication technologies. These rising link speeds needs optimisations in the data handling mechanism of computer systems or bus systems with higher bandwidths to transfer the data.

Network interface cards for wired and wireless communication are used to prepare data for communicating. This includes the handling of data from the CPU to the hardware and the conversion of the data into the electrical representation used by the communication technology.

High speed computer networks use copper or optical interfaces for communicating data. The link speed standard of computer systems is nowadays 1Gbps. This link speed defines only the theoretical data throughput between two network nodes. The challenge of fast communication is to serve enough data for transmission. The rising data link speed needs improvements in the data handling mechanism between the CPU and the NIC.

The architecture for handling data between the CPU and the NIC includes the development of a software part, which is executed by the CPU, and a new design of the hardware, which is represented by a NIC. The tasks of the software are to allocate memory in the system memory, create packets for sending, handle received packets to the destination application and to handle interrupts triggered by the hardware. The hardware has to send and to receive data from the network, store it in the system memory and signal the CPU, that new data is available.

This chapter describes an optimised data handling architecture for NICs. The concept of an optimised architecture as well as the hardware and software realisation are described.

4.1 Concept

A computer system consists of multiple connected components. The peripheral units share a bus system for exchanging data to each other. The host system with a NIC consists of the units illustrated in Figure 4.1. The PCI Express bus is the state of the art communication bus on host systems. The CPU is accesses the NIC over the PCI Express bus and the NIC is accesses the system memory also with this bus. The data handling module has an Avalon-MM bus to communicate with the PCI Express module and uses two Avalon-ST buses to transfer received and transmit packet data to the Gigabit MAC IP Core.

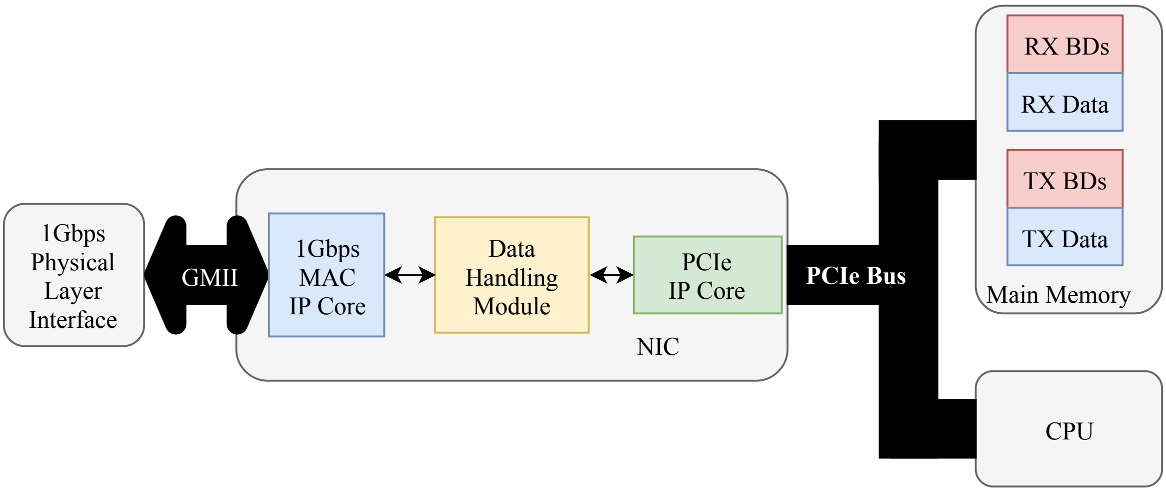


Figure 4.1: Concept of the Data Handling Module

The communication between the CPU and the NIC is divided into two parts: At first the meta information of a packet must be available on the NIC and then the packet payload can be fetched. In order to increase the throughput of data the communication should be optimised. This goal can be reached by realising data exchange with a minimum of bus and memory requests. Besides that, direct communication should only be necessary fir configuring the NIC and for reading interrupt signals.

The minimisation of bus requests is achieved by fetching multiple meta information and packet data with one burst transaction. The burst length can be set to meet the payload sizes of one PCI Express packet. As illustrated in Figure 4.1, the BD and the packet data of all packets are stored in memory blocks. The organisation of the BD (Section 4.1.1) enables the NIC to fetch multiple meta information of packets with one burst transaction. This optimises the usage of the bus and accelerates the data transmission.

4.1.1 Buffer Descriptor Management

The meta data are organised in buffer descriptors, which have a different structure for sent and received packets. Table 4.1 describes the list elements of a Transmit Buffer Descriptor (TX BD) and Table 4.2 describes Receive Buffer Descriptor (RX BD) list elements.

The list elements of both BDs are designed as linked list. If the NIC gets the start address of the first BD element, the hardware does not need any other configuration to get a new BD than that. The BD fetching is optimised by using a data block for storing the BD. This enables the hardware to get multiple BD with only one read burst transaction from the main memory. The last BD list of the data block is linked to the first BD list via the 7th entry (Address of the next RX BD List) of the structures. The BD storage can be regarded therefore as ring buffer of BD. Figure 4.2 illustrates the memory used for two RX BD.

Entry	Description
0	TX BD Control/ Status Word
1	Reserved
2	TX BD Address Word
3	Timestamp Seconds
4	Timestamp Nanoseconds
5	Reserved
6	Reserved
7	Address of the next TX BD List

Table 4.1: Transmit Buffer Descriptor list elements

Entry	Description
0	RX BD Control/ Status Word
1	Reserved
2	RX BD Address Word
3	Reserved
4	Reserved
5	Reserved
6	Reserved
7	Address of the next RX BD List

Table 4.2: Receive Buffer Descriptor list elements

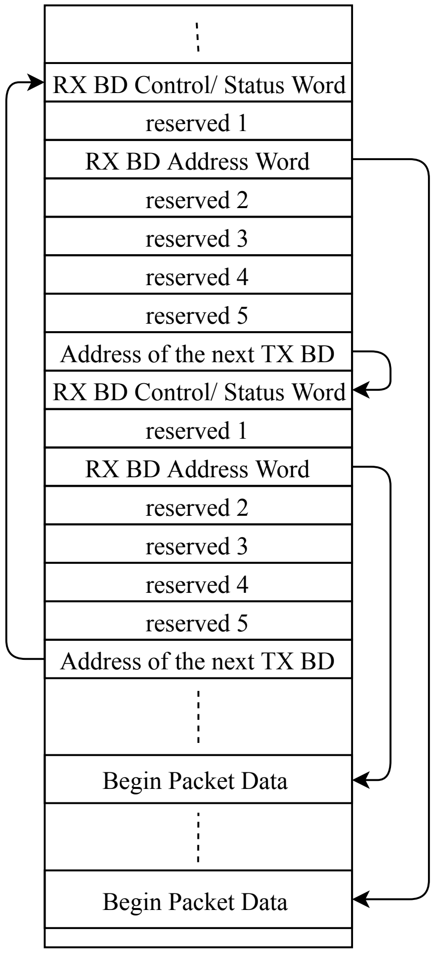


Figure 4.2: Memory organisation of RX BD ring buffer with two RX BD

The first element of the BD list is the TX or RX BD Control/ Status Word, which stores multiple transmission parameters. The data handling procedure uses the 15th bit (empty bit) for storing the availability of a RX BD. The RX can be used by the hardware, if the empty bit is set. A TX BD is marked as filled by setting the 15th bit (TX BD Ready) of the TX BD Control/ Status Word.

The BDs are initialised by the device driver and the address of the first BD is saved to a configuration register. The hardware is polling for new BDs, until it gets valid ones. The communication of meta and payload data of a packet is therefore done by a polling action from the main memory. The CPU communicates the data with the help of the system memory to the NIC. It has just to trigger a memory copy from the application data to the memory location accessible by the NIC. This is a very performant way to handle the data because the CPU is nearly not utilised and the data exchange between CPU and memory is optimised.

4.1.2 Receive Data Sequence

The sequence for transmitting received data from the NIC to the operating system has the following steps (Figure 4.3). The empty blocks represent software/ driver tasks and grey blocks are hardware tasks.

1. The device driver of the NIC has to allocate memory in the system memory for storing RX BDs which have to be initialised as empty RX BDs. The data for storing the payload of a packet has to be allocated and its address saved into the 3rd element of the RX BD list. The 7th element must locate the address of the next RX BD and the last RX BD must point to the first RX BD list address to create the wanted ring buffer.
2. The device driver has to set the start address of the RX BD and some configuration registers on the NIC.
3. The first task of the hardware is to fetch multiple RX BDs to be able to transmit received packets to the system memory. Multiple RX BDs are read from the memory by using a read burst request. The availability of an empty RX BD can be proven by reading the 15th bit of the RX BD Control/ Status Word. If one read RX BD is marked as not free, the hardware stops reading the other RX BD, which are fetched during the burst request. Empty RX BDs are stored in a FIFO.
4. The hardware reads an empty RX BD from the FIFO and is now ready for receiving data.
5. If the memory, which stores the received Ethernet packets, signals, that new data is available, the data handling module starts to upload the data to the main memory.
6. After the packet data was transferred, the RX BD of the packet is transmitted to the main memory and the received packet interrupt signal is set for one clock cycle.
7. The device driver gets activated by the interrupt. The PCI express only supports one interrupt signal, that is why the NIC has to read the interrupt source from the NIC and can then handle the received data over to the application specified in the packet data payload.
8. After processing the RX BD, the value of the BD Control/ Status Word it is set to an initial value.
9. If the FIFO for storing the RX BD is empty, new receive RX BDs are read from the system memory (step 3), otherwise step 4 is executed.

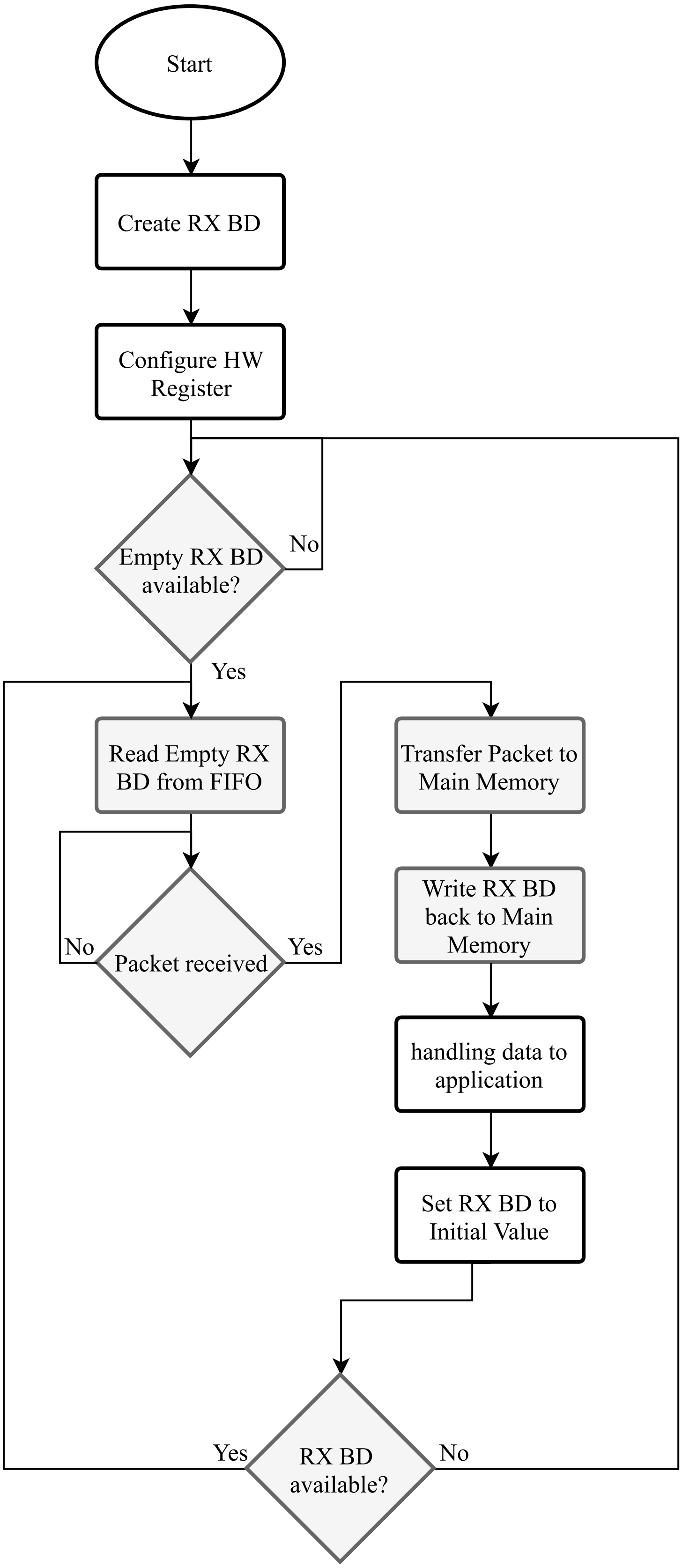

Start
Create RX BD
Configure HW Register
Empty RX BD available?
No
Yes
Read Empty RX BD from FIFO
Packet received
No
Yes
Transfer Packet to Main Memory
Write RX BD back to Main Memory
handling data to application
Set RX BD to Initial Value
RX BD available?
Yes
No

Figure 4.3: Receive Data Sequence

4.1.3 Transmit Data Sequence

The sequence for sending data over the network has the following steps (Figure 4.4). The empty blocks represent software/ driver tasks and grey blocks are hardware tasks.

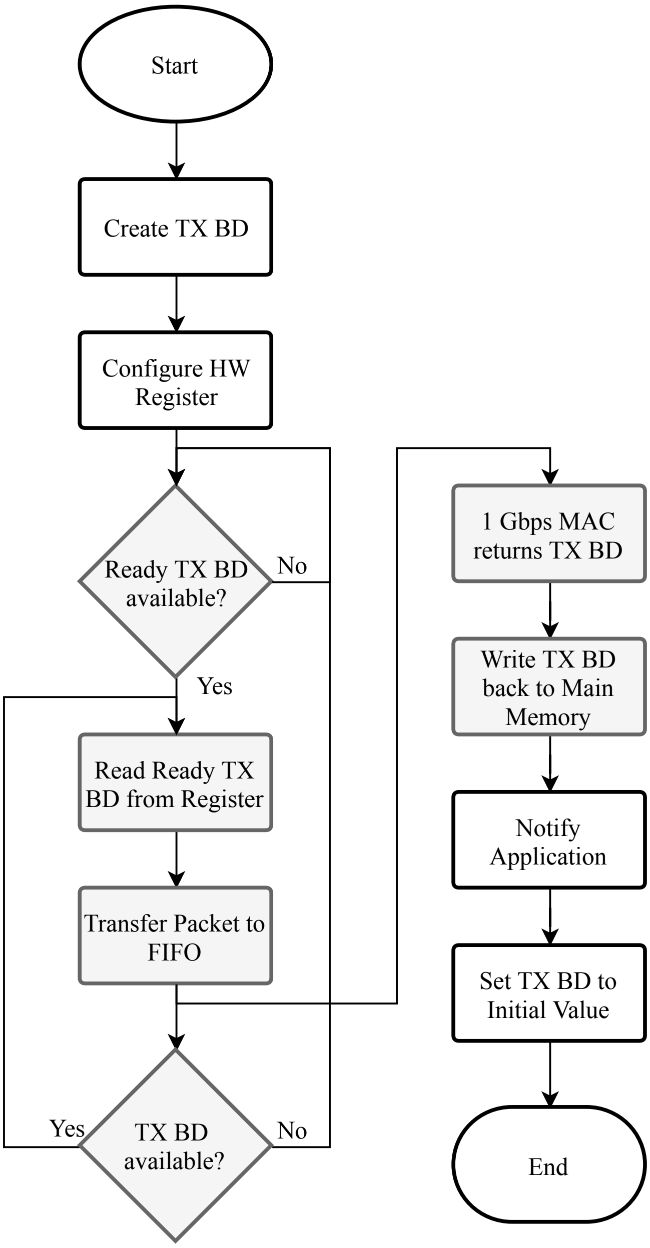


Figure 4.4: Transmit Data Sequence

1. The device driver of the NIC has to allocate memory in the system memory for storing TX BDs, which have to be initialised as not ready TX BDs. The data for storing the payload of a packet has to be allocated and the address of it has to be saved into the 3rd element of the TX BD list. The 7th element must locate the address of the next TX BD and last TX BD must point to the first TX BD list address to create the wanted ring buffer.

2. The device driver has to set the start address of the TX BD and some configuration registers on the NIC.
3. The first task of the hardware is to fetch multiple TX BDs. Multiple TX BDs are read from the memory by using a read burst request. The availability of a ready TX BD can be proven by reading the 15th bit of the TX BD Control/ Status Word. If one received TX BD is marked as not ready, the hardware stops reading the other TX BD, which are fetched during the burst request. If the request contains a ready TX BD, it is stored in a register. The address, at which the TX BD elements are stored in the register, is saved into reserved bits of the BD Control/ Status Word. This address is necessary for acknowledging the transmission of a packet.
4. The hardware reads a ready TX BD from the TX BD register and starts transmitting data to the FIFO, if it space is available for store packet data.
5. The data handling module has to transfer the data from the system memory to the FIFO module, which stores the packet data for sending.
6. After the packet data are transferred, the next TX BD can be loaded from the TX BD register, if available there; If not, new TX BDs are read from the main memory.
7. The TX 1Gbps Mac IP Core of the NIC indicates, that a packet was sent, by setting a signal. The TX 1Gbps MAC IP Core and the data handling module are operating in different clock domains, so synchronisation mechanisms have to be designed. The TX BD Control/ Status Word and the time stamps of the transmitted packet are returned
8. After writing the TX BD elements back to the main memory, a transmit interrupt is triggered.
9. The device driver gets activated by the interrupt signal. The current PCI Express IP Core implementation only supports one interrupt signal, that is why the NIC has to read the interrupt source from the NIC first. Only then it can signal the operating system, that the packet data were sent.
10. After processing the TX BD, the value of the TX BD Control/ Status Word it is set to an initial value.

4.2 Hardware

The environment, in which the optimised architecture is built into, is illustrated in Figure 4.5. A PCI Express IP core is used to implement the PCI Express bus functionality. The Data Handling Module communicates over the Avalon-MM bus with the PCI Express core. The Avalon-MM bus transfers 64 bits of data with each transaction. The MAC core of the NIC uses the Avalon-ST interface to get sent data and to send received data to the Data Handling Module.

The functionality of the NIC is split into multiple modules. The modules consist of different processes for enabling the data handling between the main memory and the NIC. Figure 4.6 illustrates the different modules. The system memory is accessed with the help of the Avalon-MM interface; the Media Access Control modules for receiving and transmitting packets are accessed through two Avalon-ST interfaces. Multiple configuration registers can be read to setup

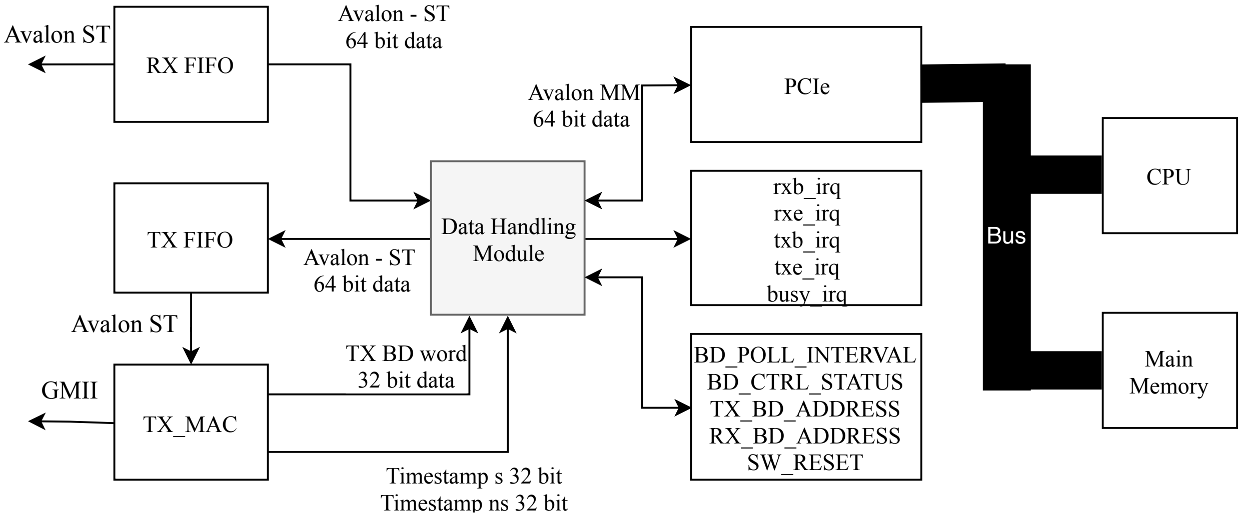


Figure 4.5: Environment of the Data Handling Module

basic configurations of the data handling architecture. Multiple interrupt signals are used to notify the CPU, that packet data were transmitted.

This section will describe the behaviour of the data handling architecture modules.

4.2.1 TX BD Register

The TX BD Register stores the meta information for sending a packet on the network. The input and output signals of the module are illustrated in Figure 4.7. The size of the storable TX BD can be configured by a generic value. The whole data architecture is built vendor and platform independent.

Three entries of the TX BD list have to be stored, to have the necessary information for sending a packet:

- **entry '00':** Address of the TX BD list element
- **entry '01':** TX BD Status/ Control Word
- **entry '10':** TX BD Address Word

The following processes and modules are implemented to enable the storing of TX BD:

Generated Register Memory Core

Storing blocks like registers or First-In-First-Out (FIFO) memories can be integrated with IP cores. This IP cores can be generated with development tools of the destination platform. The TX BD Register needs a register memory element for storing the TX BDs.

The TX BD Register module uses a generic value for getting the size of the storable elements. This value must be the source for calculating the size of the generated register. The register needs the following parameters:

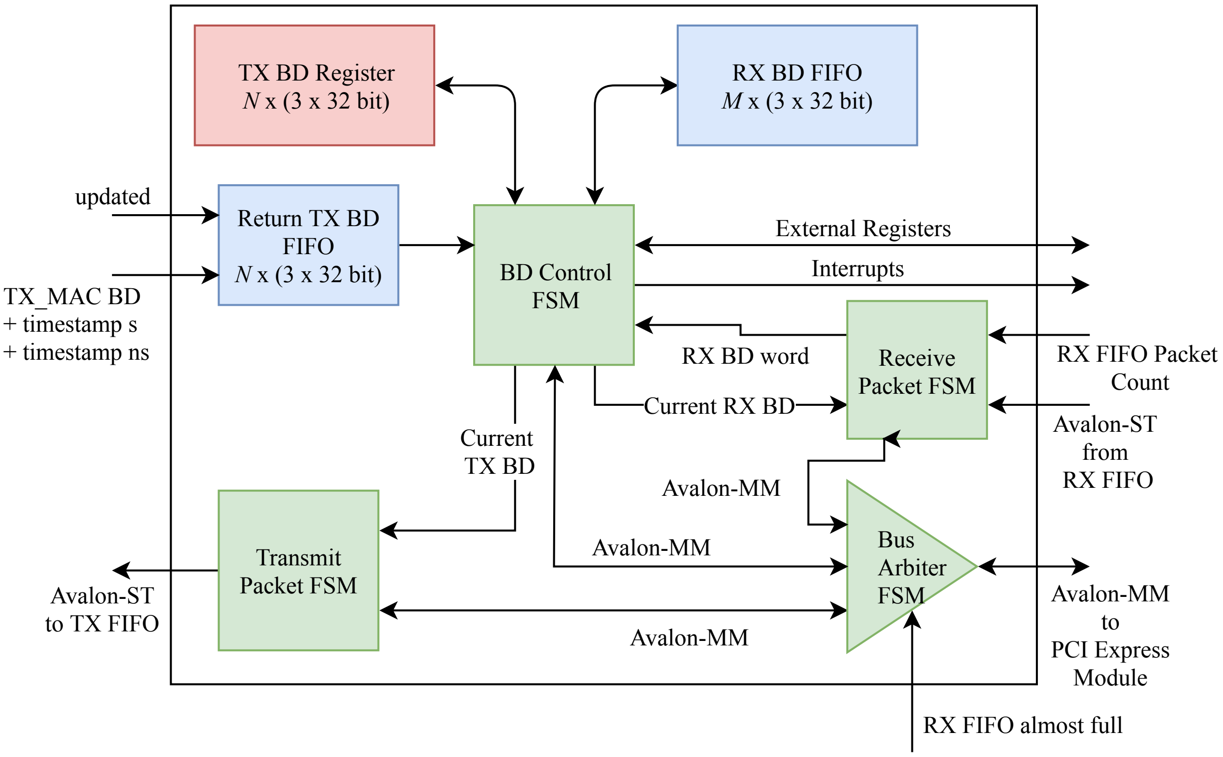


Figure 4.6: Modules of the Data Handling Architecture

- Type: Single Port Ram
- Data With: 32bit. Each BD entry consists of 32 bits.
- Data Depth: 8 to 512 entries. It is calculated out of the entries, which are addressed by the signal *txbd_reg_entry_i* and the number of storable TX BD. For storing 4 TX BD, the register needs a data depth of 16.
- Unregistered Output
- Registered Input

Process for Calculating Available Space

A TX BD is valid and uses storage, if the 15th bit of the TX BD Control/Status Word (entry '01') is set. A counter is reporting the available space to the output signal *txbd_reg_avail_space_o*. The address of a free register is reported with the signal *txbd_reg_free_addr_o* and the *txbd_reg_free_addr_valid_o* validates, that the address is valid and can be used.

Process for Checking the BD Storage

This process checks, if the content of the registers was not corrupted during storage. The TX packet transmission order is also checked during this procedure. The sent order must be the same order, in which the TX BD were written to the register. An error is reported with the signal *txbd_reg_sanity_check_irq_o*.

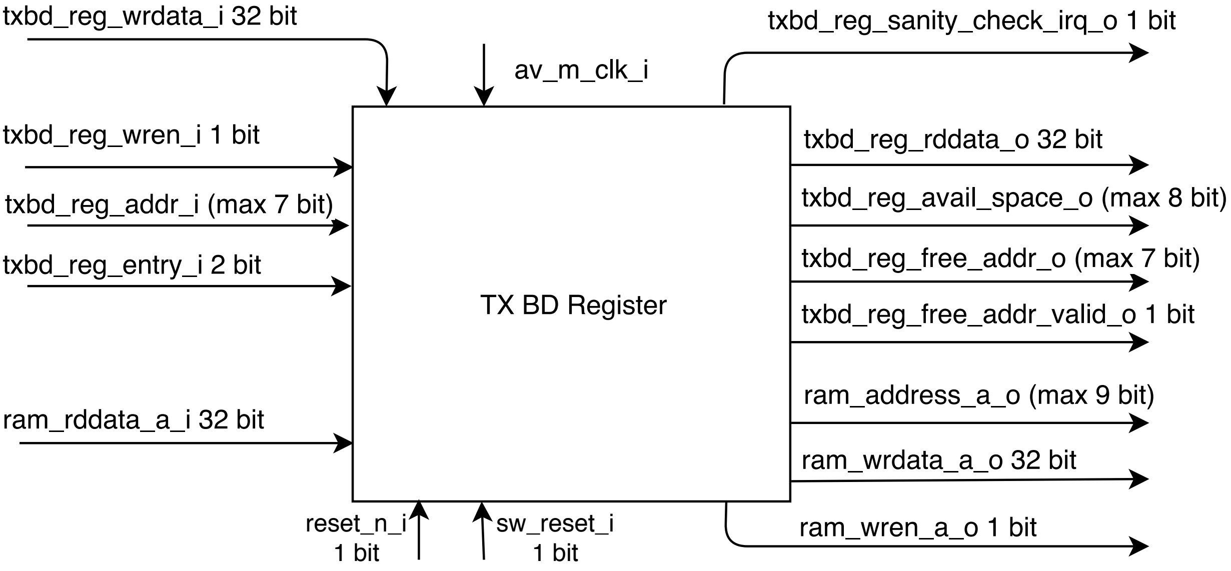


Figure 4.7: Inputs and Outputs of the TX BD Register module

Process for Connecting Signals to the Register Memory Core

The input and output signals used for communicating with the storage are registered. The signal *txbd_reg_entry_i* is directly routed to the first two address signals of the generated register. The other address signals of the generated registers are connected with the signal *txbd_reg_addr_i*.

4.2.2 Return TX BD FIFO

The description of a transmit packet sequence (Section 4.1.3) illustrates, that the TX MAC returns the BD of a packet after sending. Multiple packets can be stored in the TX MAC and can wait there to be sent. The Return TX BD FIFO is used for storing TX BD and the timestamp information, which are returned from the TX MAC. The FIFO is necessary, because new sent data can be returned, before the architecture is able to update the data to the main memory. Figure 4.8 illustrates the input and output signal of this module.

The Return TX BD FIFO module has to save the returned data to a FIFO. The TX MAC and the data handling architecture are working in different clock domains. To ensure correct functionality independent on the clock relation between the two clock domains, the 32-bit input signals from the TX MAC are "frozen" for at least 16 clocks at transmit clock. The availability of new data is indicated by negating the signal *tx_mac_updated_i*. An edge detection (rising, falling) is performed in the data handling architecture clock domain to detect the availability of new data.

The Return TX BD FIFO saves the following data:

- TX BD Control/ Status Word of the sent packet
- Timestamp in seconds
- Timestamp in nanoseconds

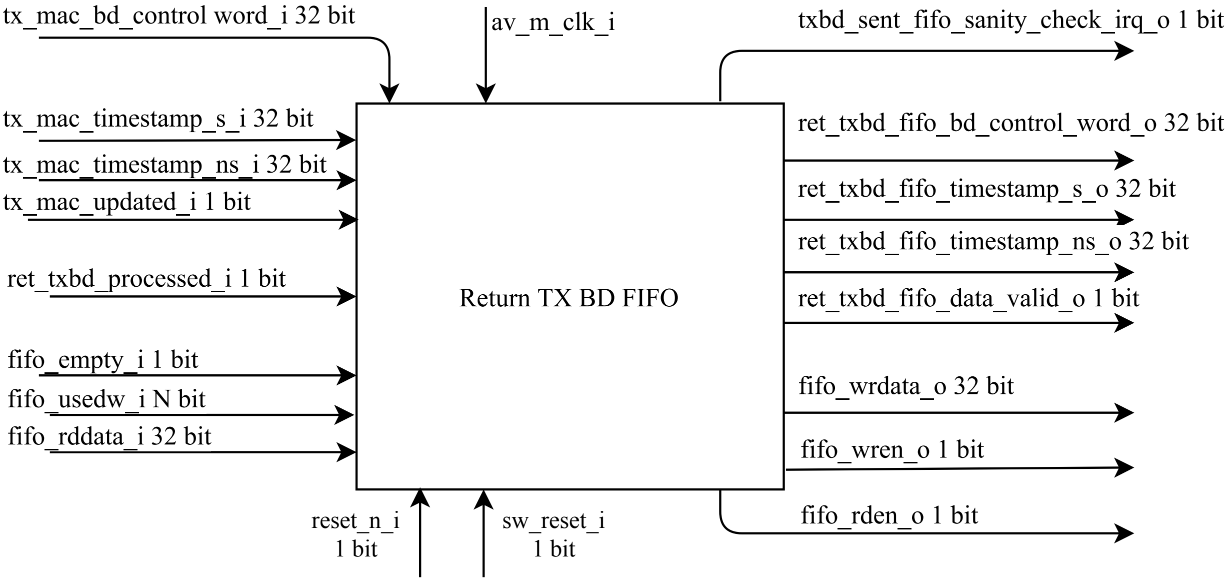

tx_mac_bd_control word_i 32 bit
av_m_clk_i
txbd_sent_fifo_sanity_check_irq_o 1 bit
tx_mac_timestamp_s_i 32 bit
tx_mac_timestamp_ns_i 32 bit
tx_mac_updated_i 1 bit
ret_txbd_processed_i 1 bit
fifo_empty_i 1 bit
fifo_usedw_i N bit
fifo_rddata_i 32 bit
Return TX BD FIFO
ret_txbd_fifo_bd_control_word_o 32 bit
ret_txbd_fifo_timestamp_s_o 32 bit
ret_txbd_fifo_timestamp_ns_o 32 bit
ret_txbd_fifo_data_valid_o 1 bit
fifo_wrdata_o 32 bit
fifo_wren_o 1 bit
fifo_rden_o 1 bit
reset_n_i 1 bit
sw_reset_i 1 bit

Figure 4.8: Inputs and Outputs of the Return TX BD FIFO module

After data is stored in the FIFO, the data should be handed over to the BD Control FSM module, which updates the TX BD list entry data.

The following processes and modules are designed to store the data:

Generated FIFO IP Core

The platform independence of a memory can be guaranteed, if the FIFO for storing the data is generated with a tool supplied by the vendor of the target system. The FIFO has to be generated using the following parameters:

- Type: Native FIFO
- Data With: 32bit. Each entry consists of 32 bits.
- Data Depth: 8 to 512 entries. The data depth depends on the storable TX BD in the TX BD Register module. It must have the same size.
- Asynchronous Reset
- Synchronous Reset
- Empty Signal
- Used Word Signal
- Unregistered Output

Process for Reading Data from the TX MAC

This process reads the TX BD Control/ Status Word and the timestamp signals from the TX MAC and saves it to the FIFO. Due to the fact that the TX MAC and the data handling architecture are operating in different clock domains, it is necessary to add the functionality for synchronisation. The availability of new data is indicated by negating the signal *tx_mac_updated_i*. As already mentioned, an edge detection (rising, falling) is performed to detect the availability of new data. After detecting the edge, the signals from the TX MAC are valid. The data are stored in the signals *tx_mac_bd_control_word_i*, *tx_mac_timestamp_s_i* and *tx_mac_timestamp_ns_i*.

Process for Transferring Data to the Data Handling Master

If the *fifo_emtpy_i* signal of the generated FIFO is cleared, new data is available. The data is loaded to the signals *ret_txbd_fifo_bd_control_word_o*, *ret_txbd_fifo_timestamp_s_o* and *ret_txbd_fifo_timestamp_ns_o*. The signal *ret_txbd_fifo_data_valid_o* is set afterwards to notify the BD Control FSMmodule, that new data is available. The signal *ret_txbd_fifo_data_valid_o* is cleared, after the data were updated and the BD Control FSM module has set the signal *ret_txbd_processed_i*.

Process for Checking the Storage

A storage sanity check is implemented in the module, which uses a counter to record the number of stored BDs. If no read or write request is triggered for 2 cycles and the counter of the stored BDs multiplied by 3 is not equal to the used word signal value, the storage is corrupted and the interrupt signal *txbd_sent_fifo_sanity_check_irq_o* is set.

Process for Connect Signals to Generated FIFO Module

The input and output signals for transferring data to the modules are registered signals. This process manages the connection of the signals.

4.2.3 RX BD FIFO

The RX BD FIFO stores the meta information for receiving a packet on the network. The size of the storable RX BD can be configured by a generic value and the EDA tool of the platform can build the vendor dependent FIFO, which is used in this module. The input and output signals of the module are illustrated in Figure 4.9.

Three entries of the RX BD list must be stored in order to have the necessary information for storing a received packet:

- Address of the RX BD list element
- RX BD Status/ Control Word
- RX BD Address Word

The processes for checking the storage and for connecting the FIFO signals to the input and output signals are the same as described in Section 4.2.2 and the generated FIFO IP Core is also the same as in Section 4.2.2

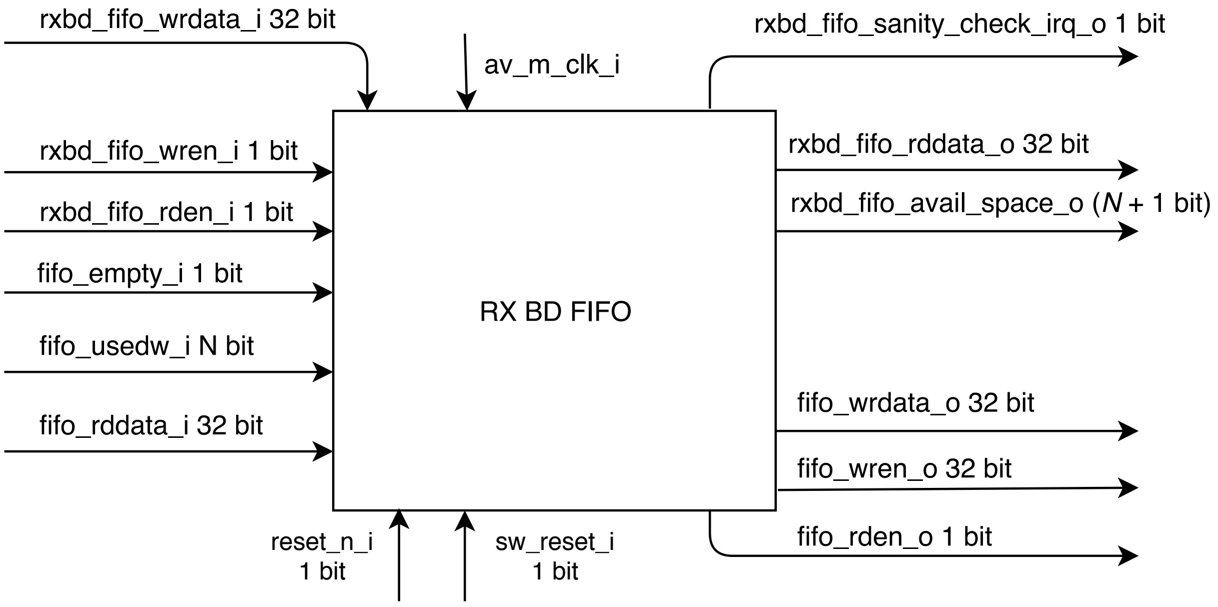


Figure 4.9: Inputs and Outputs of the RX BD FIFO module

4.2.4 Bus Arbiter FSM

The data handling architecture uses the Avalon-MM bus to communicate with the main memory. Three modules called the BD Control Finite State Machine (FSM), the Transmit Packet FSM and the Receive Packet FSM need to have access to the Avalon-MM bus. The Bus Arbiter FSM controls the access to the bus. The BD Control FSM has a 50% chance to get access to the bus. The Transmit Packet FSM and the Receive Packet FSM equally share the other 50%. However, the Receive Packet FSM is prioritised over the Transmit Packet FSM, if the RX FIFO of the RX MAC is almost full.

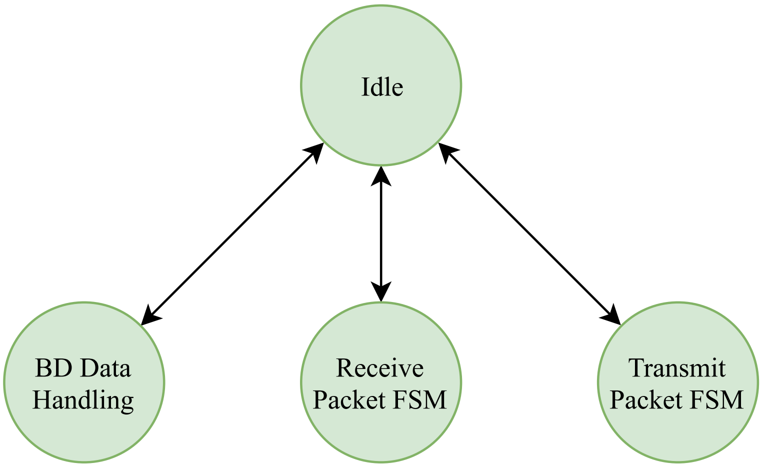


Figure 4.10: Finite State Machine of the Bus Arbiter for the Avalon-MM interfaces

The Bus Arbiter FSM has the following states (Figure 4.10):

- **Idle**: This is the default state of the FSM. Any other state returns back to the idle state, if the Avalon-MM bus is no longer needed. The request of a bus is handled here and all other states are available from here.

- **BD Control FSM**: The BD Control FSM module signals are mapped to the Avalon-MM signals and the FSM returns back to the t_idle state, if it does not need the bus any more.
- **Transmit Packet FSM**: The Transmit Packet FSM signals are mapped to the Avalon-MM signals and the FSM returns back to the t_idle state, if it does not need the bus any more.
- **Receive Packet FSM**: The Receive Packet FSM signals are mapped to the Avalon-MM signals and the FSM returns back to the t_idle state, if it does not need the bus any more.

4.2.5 BD Control FSM

The BD Control FSM module controls the BD exchange between the main memory and the NIC. It polls RX and TX BDs from the memory and updates the used BDs. The Transmit Packet FSM and the Receive Packet FSM module are notified, if valid BDs are available. The BD Control FSM module reads BDs from the memory and serves them to the Transmit Packet FSM and the Receive Packet FSM module.

Polling BD from Memory

Two processes exist for managing the data polling from the main memory to the local memories. The difference between the RX BDs and the TX BDs is, that the RX BDs are stored in a FIFO and the TX BDs are stored in a register.

A BD polling action is triggered if space is available in the local buffers and a timeout got triggered. The timeout mechanism is described in paragraph 4.2.5. The polling action only requests as many BDs as are storable in the local memories. The fetched BDs are only stored, if the 15th bit (RX BD: Empty bit, TX BD: Ready bit) is set.

Updating BD to Memory

After a packet is received or was sent, the meta information has to be refreshed in the main memory. Additionally, the operating system is informed, that new data is available. For the received packet, only the RX BD Control/ Status Word must be updated. The BD of a transmitted packet can include a timestamp; hence, three elements of the TX BD are written back to the main memory: The TX BD Control Status Word, the nanosecond and the second timestamp.

A RX BD is directly updated, after packet data were transferred from the NIC to the main memory. A TX BD is updated, after being returned by the TX MAC and sent to the Data Handling Module. The Return TX BD FIFO stores the returned TX BD Control/ Status Word and the timestamps. After a BD was written back to the main memory, an interrupt is triggered in order to signal, that new data is available.

Serving BD to Transmit Packet FSM and Receive Packet FSM

The BD Control FSM module reads valid BDs from the storage and serves them to the Transmit Packet FSM and the Receive Packet FSM. After the FSMs have used the BD, an acknowledge signal is triggered and new BDs are read from the memory.

BD Poll Timeout

Two counters are used to trigger timeouts for polling new BDs from the memory. The 32 bit register *BD_Poll_Interval* stores the value, where a poll request should be triggered. The first 16 bits of the register stores the microsecond value, at which a RX BD should be requested and the last 16 bits store the value for the TX BD poll timeout. The counter is counting until the triggered value is reached and is cleared, when BDs are polled from the memory.

4.2.6 Transmit Packet FSM

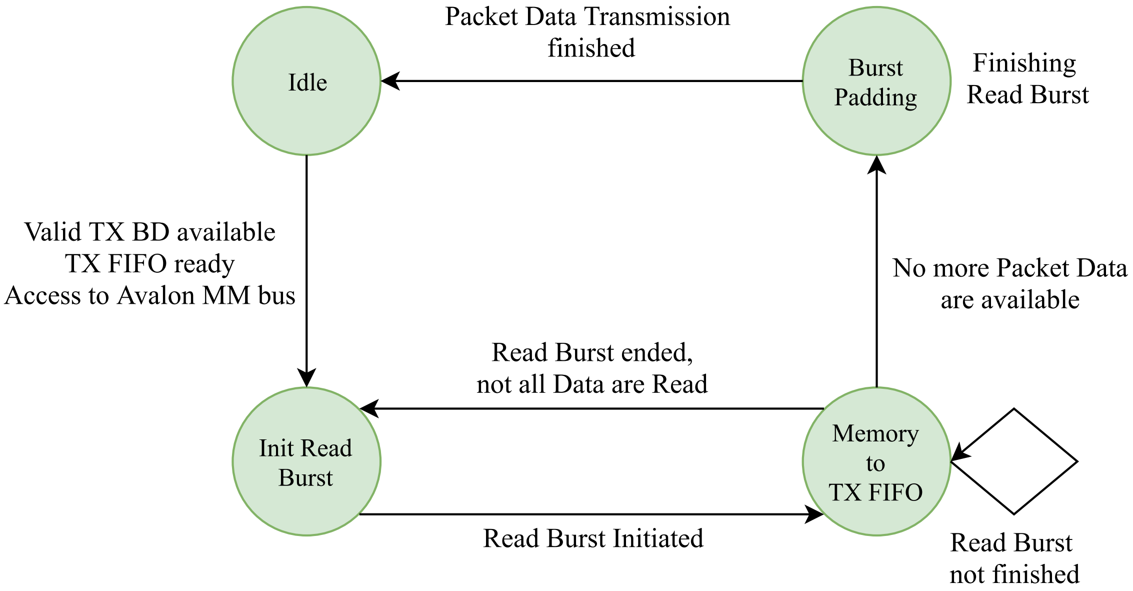


Figure 4.11: Finite State Machine for transferring packets to send from the main memory to the NIC

The transmission of packet data is triggered, if the BD Control FSM module serves a valid TX BD. The state machine initiates an Avalon-MM read burst from the memory location specified in the TX BD Address Word. The read data are transferred with the Avalon-ST interface to the TX FIFO, which stores the packet data for sending.

The Transmit Packet FSM states are illustrated in Figure 4.11

- **Idle**: The idle state waits until three conditions are fulfilled: when a valid TX BD is served by the BD Control FSM module, the TX FIFO has space for new data and the Bus Arbiter modules grants access to the Avalon-MM bus.
- **Init Read Burst**: An Avalon-MM read burst action is triggered by this function. The packet data transfer has to be split in multiple burst transactions. The maximum number of transactions during a burst is located in the register BURSTLENGTH.

- **Memory to TX FIFO**: Packet data are read from the Avalon-MM interface and written to the Avalon-ST interface. If the read burst transaction is finished and more packet data is available, a new burst transaction is initiated by going to the state Init Read Burst. The amount of packet data is stored in the TX BD Control/ Status Word of the packet.
- **Burst Padding**: This state proofs, if the Avalon-MM burst transaction was finished, otherwise it reads data from the interfaces, without storing them. The next state is the Idle state, if the Avalon-MM read burst came to an end.

4.2.7 Receive Packet FSM

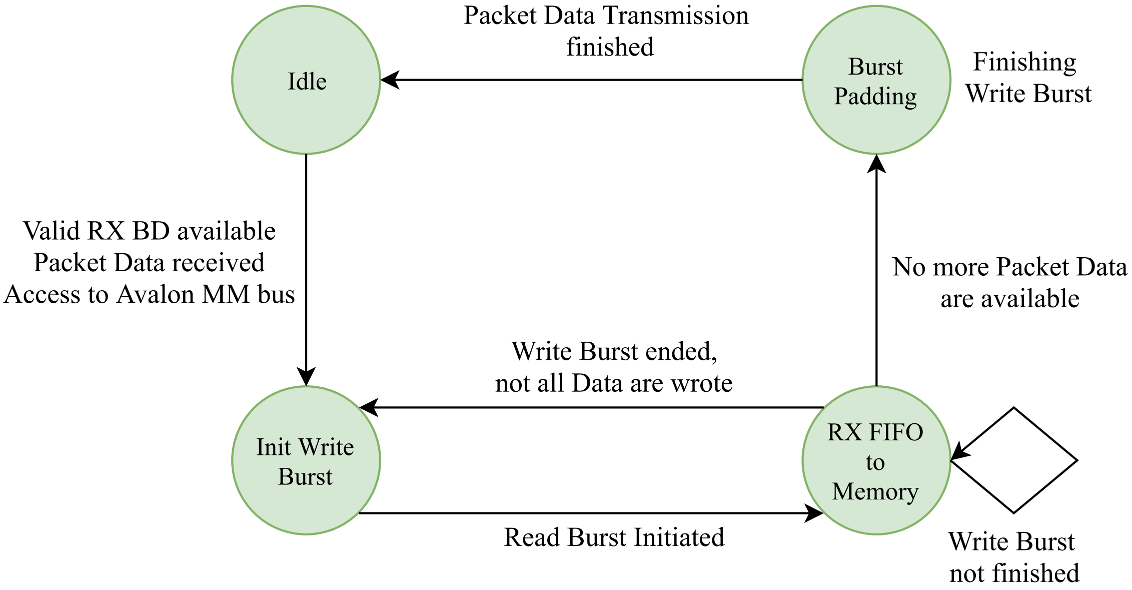


Figure 4.12: Finite State Machine for transferring received packets from the NIC to the main memory

The RX FIFO - connected to the Receive Packet FSM via an Avalon-ST interface - signals received data by backpressure (see Section 2.4.1). The packet can be transferred to the main memory, if the BD Control FSM module serves a valid RX BD to the FSM. An Avalon-MM write burst transaction is issued to the memory address stored in the RX BD Address Word for transferring the data.

The Receive Packet FSM states are illustrated in Figure reffig:ReceivePacketFSM.

- **Idle**: The idle state waits until three conditions are fulfilled: when a valid RX BD is served by the BD Control FSM module, the RX FIFO signals, that new data is available and the Bus Arbiter modules grants access to the Avalon-MM bus.
- **Init Write Burst**: An Avalon-MM write burst action is triggered by this function. The packet data transfer has to be split in multiple burst transactions. The maximum number of transactions during a burst is located in the register BURSTLENGTH.
- **RX FIFO to Memory**: Packet data is read from the Avalon-ST interface and written to the Avalon-MM interface. If the write burst transaction comes to an end and more packet data is available, a new burst transaction is initiated by going to the state Init Write Burst.

The end of a packet is signalled with the Avalon-ST signal End of Packet (EOP). After the EOP is assigned and the last data are sent to the main memory, the next state is the Burst Padding State.

- **Burst Padding**: This state proofs, if the Avalon-MM burst transaction was finished, otherwise it sends dummy data. The next state is the Idle state, if the Avalon-MM write burst came to an end.

4.3 Software

Multiple applications are communicating with other network nodes over a NIC. The operating system manages the data exchange between multiple applications with the NIC. Therefore, the software for handling the data is written as device driver of the operating systems. This device driver must be equipped with an implementation of the standard functionality of a NIC. The standard function of a NIC are initiating the device, sending data, receiving data and getting statistics of received and transmitted packets.

Figure 4.3 and 4.4 illustrate the receive and the transmit packet sequences. The blank boxes illustrate the functionality, which the software has to have implemented. This section describes the basic functionality of the NIC driver.

4.3.1 Setup NIC

During the setup process, multiple registers are configured to enable packet receiving and transmission. The direct writing to and reading from registers should be minimized during the data transmission. The setup of the NIC is only done once; so direct writing does not affect the performance of the NIC.

The memory for sending and transmitting data is allocated during the setup process. The driver defines the used amount of RX and TX BDs. Each Buffer Descriptor has eight entries, where each entry uses four bytes. The setup function must write initial values to the BDs as well as to write the address of the packet data location to the 3rd entry of the list (TX BD Address Word or RX BD Address Word).

The memory for storing all RX BDs should be allocated contiguously. In that case, the memory works as a ring buffer, so that multiple BDs can be fetched by reading multiple of 8 * 4 bytes from a start address, which is the begin of a BD. The same memory allocation mechanism is used for the TX BDs.

After the allocation of the memory, a default value is written to each RX/TX BD Control/ Status Word, the RX/TX BD Address Word is set to a location, where 2048 bytes were allocated for storing the packet data, and the Address of the next TX/RX BD List entry of the BD is written to form a ring buffer. Figure 4.2 illustrates the memory organisation of the BD memory.

4.3.2 Data Transfer

The device driver has default functions for getting data from the operating systems. One function is for sending data the other for receiving data. Both functions have to copy data from the memory accessible from the NIC to a memory location accessible to the operating system. The NIC informs the device driver by triggering an interrupt, that a packet was received or that a packet was successfully sent.

Receive Packet

After the NIC has triggered an interrupt, the device driver reads from the interrupt register of the NIC to get the interrupt source. If a new packet has been arrived, the receive interrupt signal is asserted. The NIC loads then the data of the expected RX BD to get the information for copying the data to a memory location, which is accessible by the application. After the data of the BD are read, the initial value is assigned to the RX BD Control/ Address word.

Transmit Packet

Applications can trigger data transmission with commands served by the operating systems. The operating system uses standard functions served by the device driver to transfer the data to the device driver. After copying the data to the TX BD Address Word location of the next free TX BD, the device driver sets the TX BD Control/ Status Word of the TX BD and the NIC can get the data by polling the TX BD from the memory.

The device driver needs a second function, which is triggered, if the NIC sends a transmit packet interrupt. This function is used to scan the TX BD memory, if a packet was sent to the network and inform the application, that data was successfully sent.

5 Measurements

The performance of the new data handling concept, built into the syn1588 PCI Express NIC of Oregano Systems, is measured in this section. The data throughput of the new concept is compared with the currently implemented strategy and different parameters of the new data handling method are varied to illustrate the best data handling configuration. The reduction of CPU time is measured and illustrated in Section 5.3

5.1 Setup

The performance measurement setup is composed of two directly connected computers as illustrated in Figure 5.1. The computer, with an Oregano Systems NIC installed, uses an Intel Core 2 Duo CPU E6550 with a frequency of 2.33GHz and the second computer is an Intel Pentium G4400 with a CPU frequency of 3.3GHz. It has a Realtek RTL8111/8168/8411 PCI Express Gigabit Ethernet Controller installed. Both computers are running the operating system Ubuntu 14.04 and the kernel version 3.13.0-88. The Oregano Systems NIC is programmed with the old and the new data handling concept.

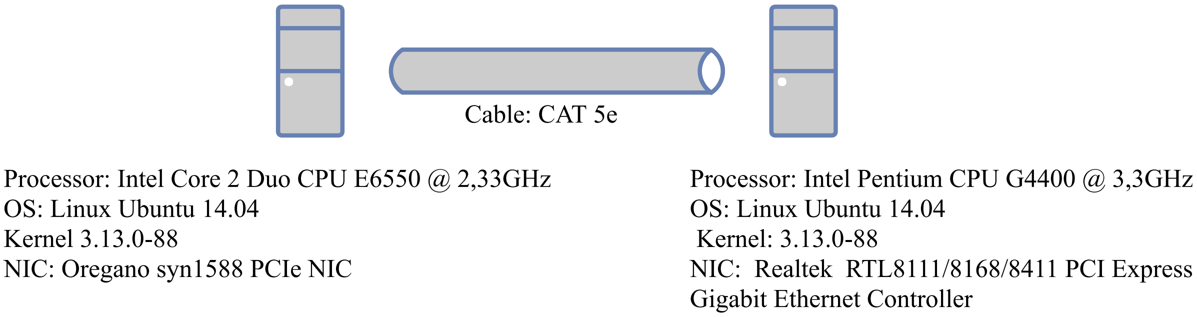

Cable: CAT 5e
Processor: Intel Core 2 Duo CPU E6550 @ 2,33GHz
OS: Linux Ubuntu 14.04
Kernel 3.13.0-88
NIC: Oregano syn1588 PCIe NIC
Processor: Intel Pentium CPU G4400 @ 3,3GHz
OS: Linux Ubuntu 14.04
Kernel: 3.13.0-88
NIC: Realtek RTL8111/8168/8411 PCI Express Gigabit Ethernet Controller

Figure 5.1: Measurement Setup

The performance tool *iperf*[1] in version 2.0.5 is used to measure the TCP data throughput of the system. The tool uses the maximum MTU size (1500 bytes) supported by Ethernet and transmits data for 10 seconds. Iperf stores the amount of sent data and calculates the bandwidth after transmission. It was used in server and in client mode on both systems in order to get the

---

[1] `https://www.iperf.fr`

maximum bandwidth for receiving and transmitting packets. The default values of the program were used to measure the bandwidth of the system.

The NIC's data throughput is highly influenced by the PCI Express bus. The maximum amount of data which is transferred between the data handling module and the PCI Express IP core, is controlled with the BURSTLENGTH register. It defines the maximum number of read or write transfers that can be initiated by the Avalon-MM bus of the data handling module. The maximum value of the burst length supported by the PCI Express IP core is 255. The burst length value is used by the PCI Express IP core to generate PCI Express packets to write data to the main memory or to trigger read requests on the main memory. The maximum supported payload size of the core is 256 bytes.

The number of buffer descriptor storable in the memories of the FPGA can be changed by generating FIFOs and SRAMs with different sizes. The former data handling concept was able to store 64 RX and 64 TX buffer descriptors.

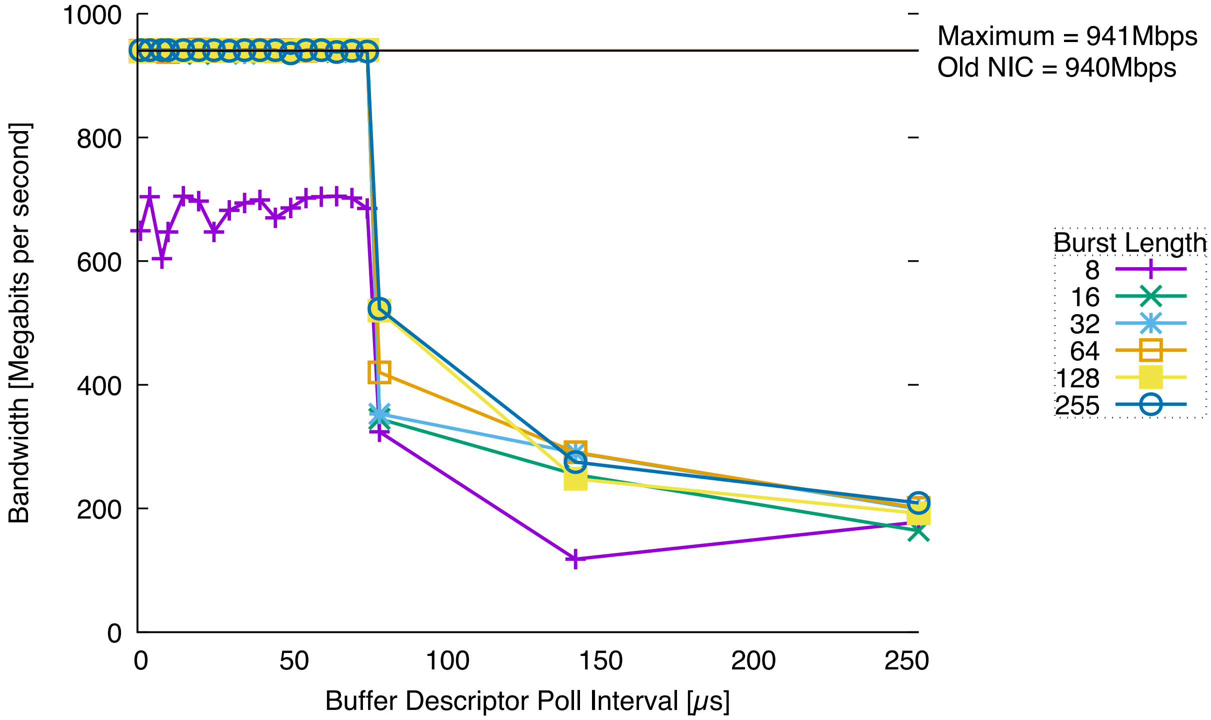
The time for polling new BDs is controlled by the register BD_POLL_TIMEOUT. It stores the number of microseconds the data handling module has to wait until it searches for new BDs. This value is set during the initialisation of the NIC.

5.2 Performance Results

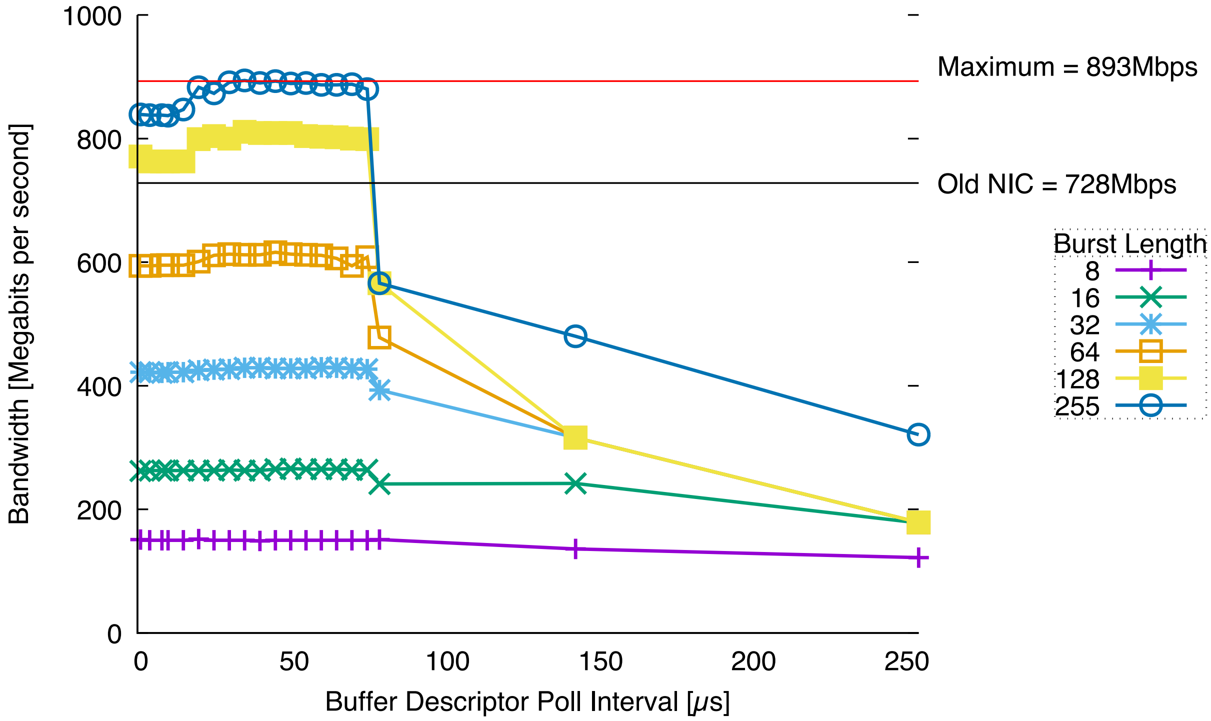
Multiple configurations of the FPGA bit stream were generated to measure the performance of different configurations of the new data handling module. Multiple FIFOs and SRAMs were generated to store different numbers of BDs on the NIC. The changing sizes enable the system to fetch different numbers of BDs at once. The following configurations were tested and the performance depending on the poll timeouts and the burst lengths were measured:

- 4 TX BD and 5 RX BD storable: Figure 5.2 - 5.3
- 8 TX BD and 10 RX BD storable: Figure 5.4 - 5.5
- 16 TX BD and 21 RX BD storable: Figure 5.6 - 5.7
- 32 TX BD and 42 RX BD storable: Figure 5.8 - 5.9

The first figure illustrates the receiving and transmitting performance for different poll timeout values and burst lengths. The timeout values are set to the same value for polling RX and TX BDs. The second figure shows the performance of the old data handling mechanism against the new one for multiple burst lengths. The best results of the configuration are picked to compare it with the results of the old.

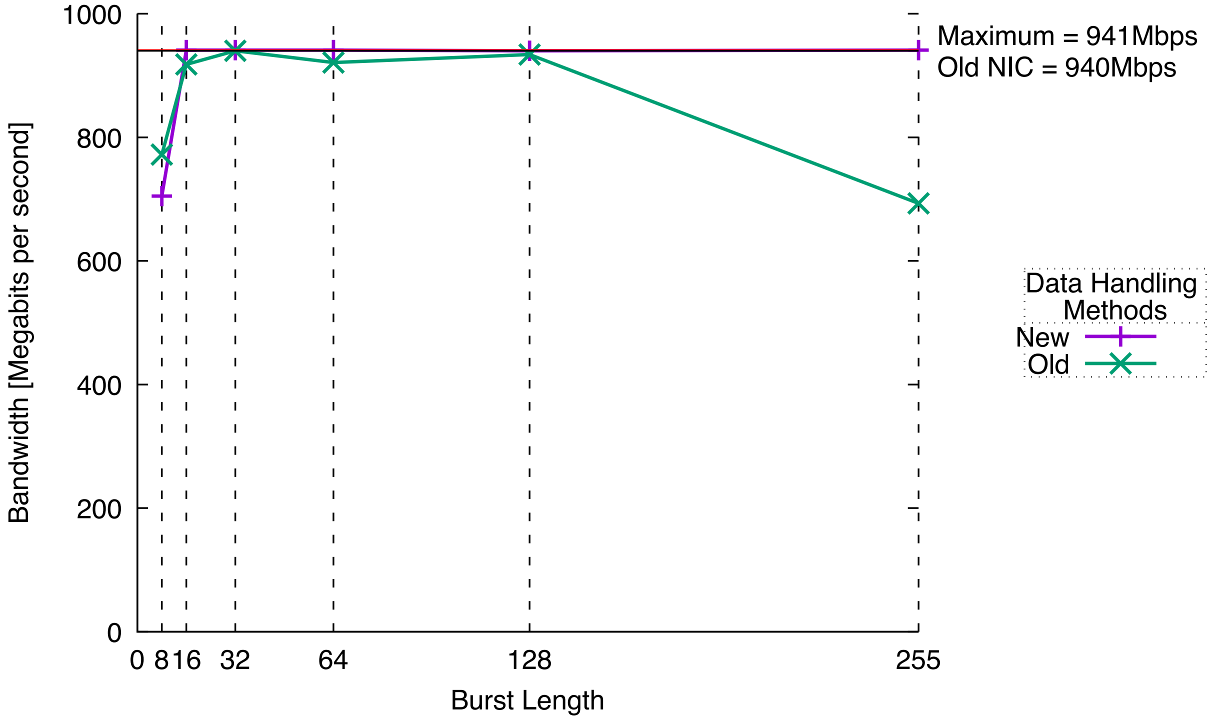


(a) Receiving Packets

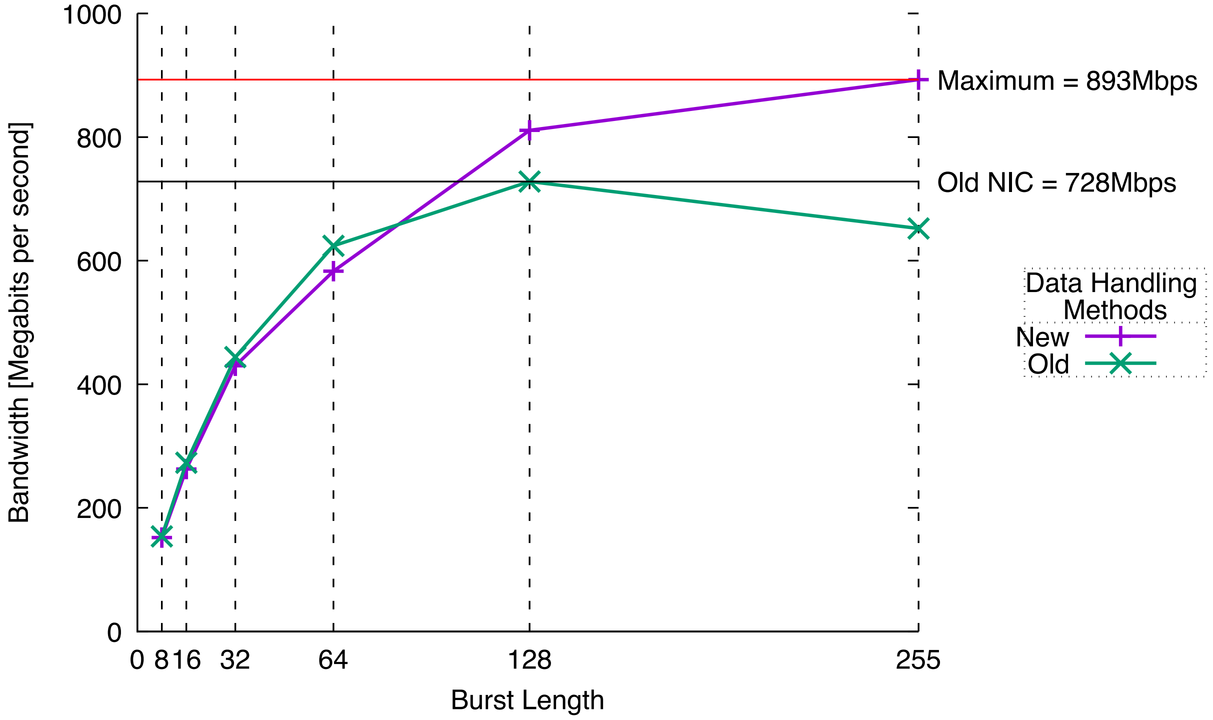


(b) Transmitting Packets

Figure 5.2: Performance comparison of the newly developed data handling architecture with the ability to store 4 Transmit Buffer Descriptors and 5 Receive Buffer Descriptors for different poll intervals

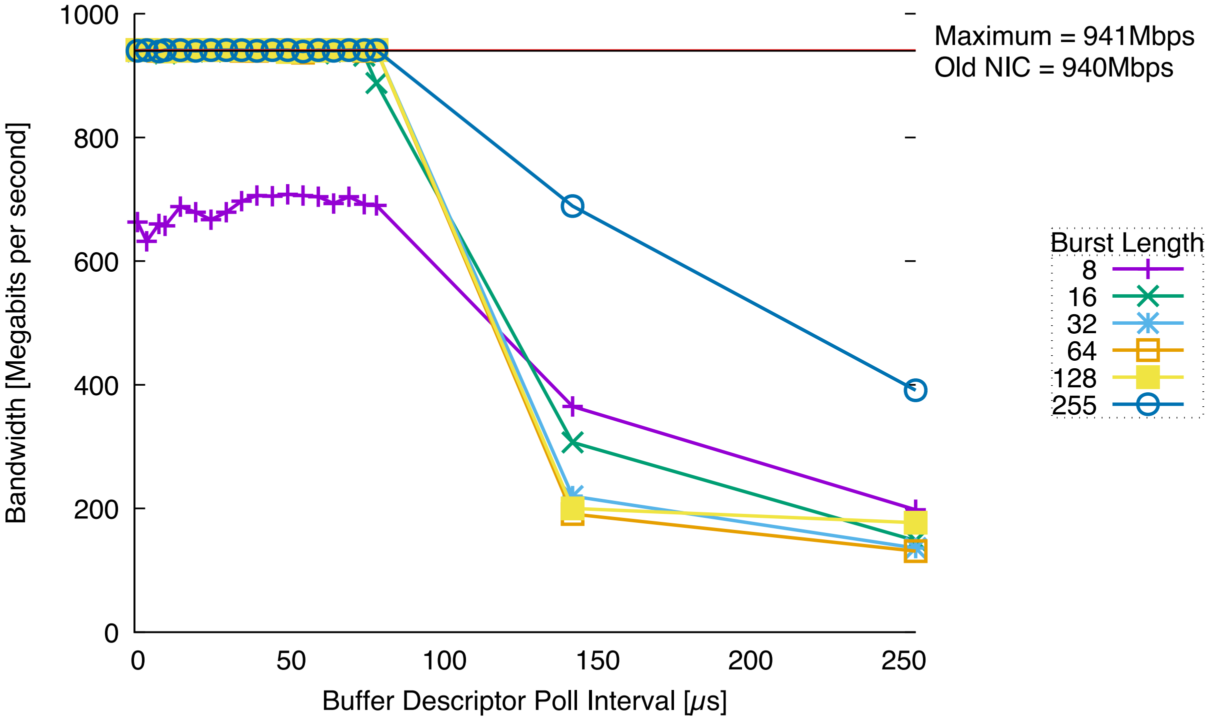


(a) Receiving Packets

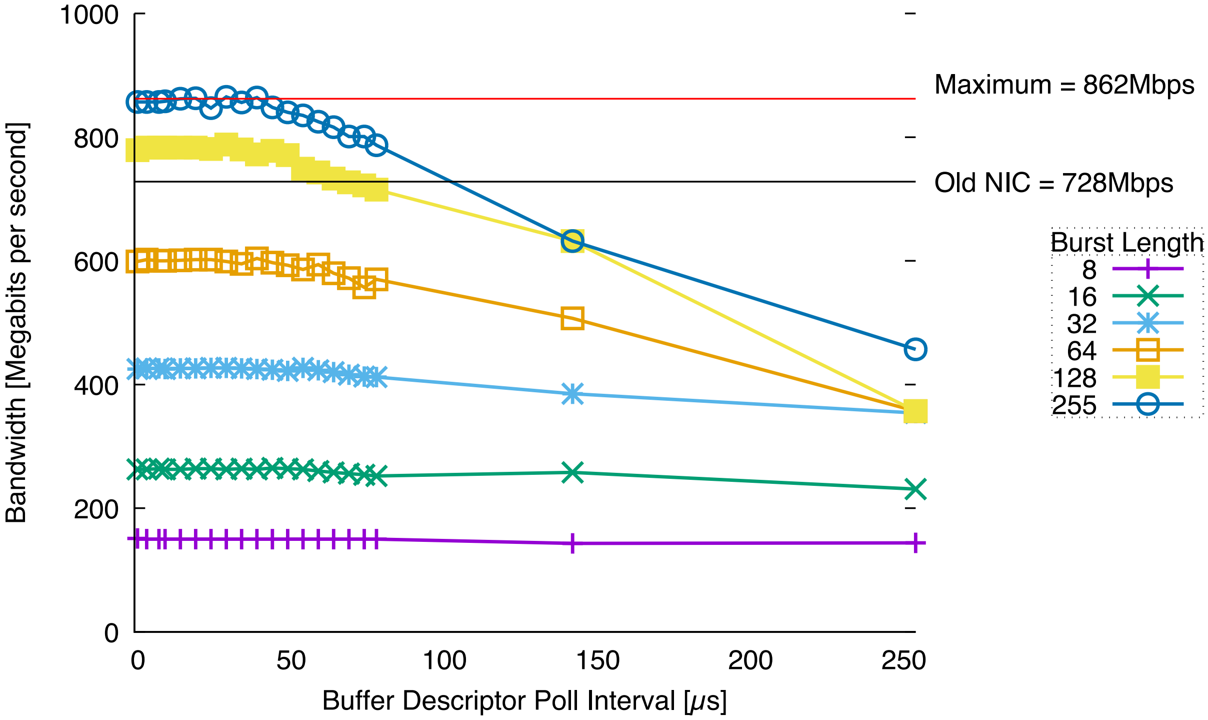


(b) Transmitting Packets

Figure 5.3: Comparison of the best bandwidth achieved by the developed data handling architecture, capable to store 4 Transmit Buffer Descriptors and 5 Receive Buffer Descriptors, against the old data handling method. d

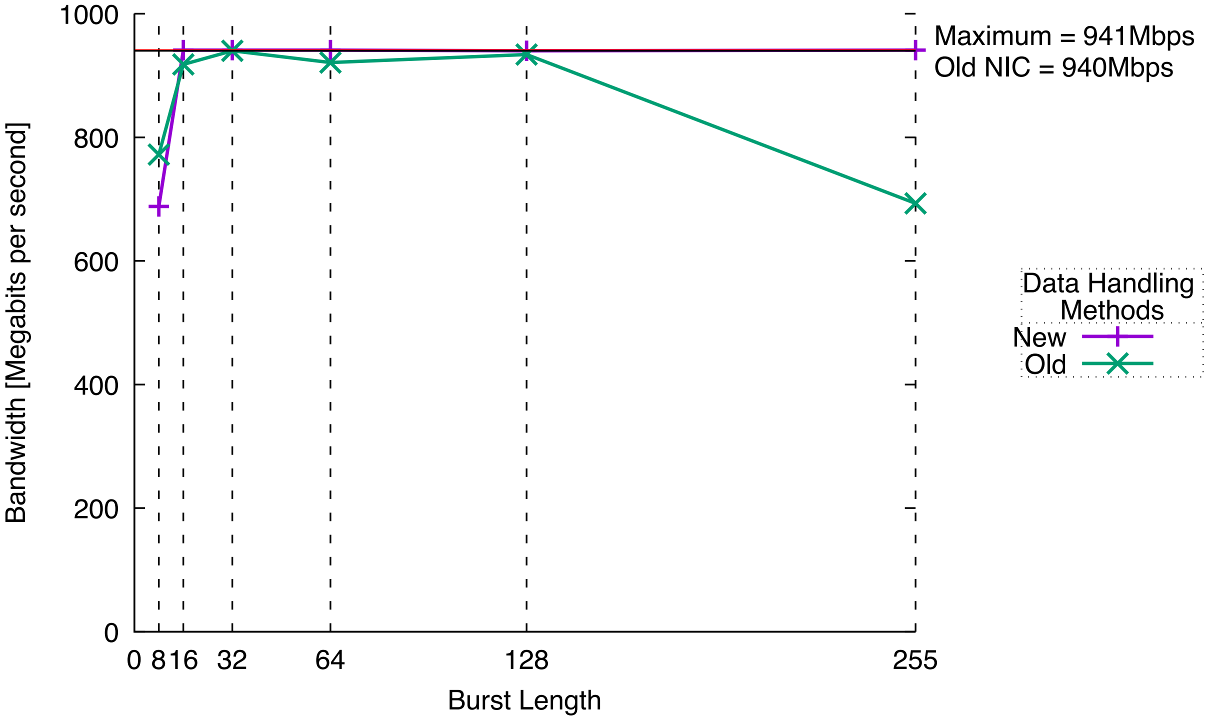


(a) Receiving Packets

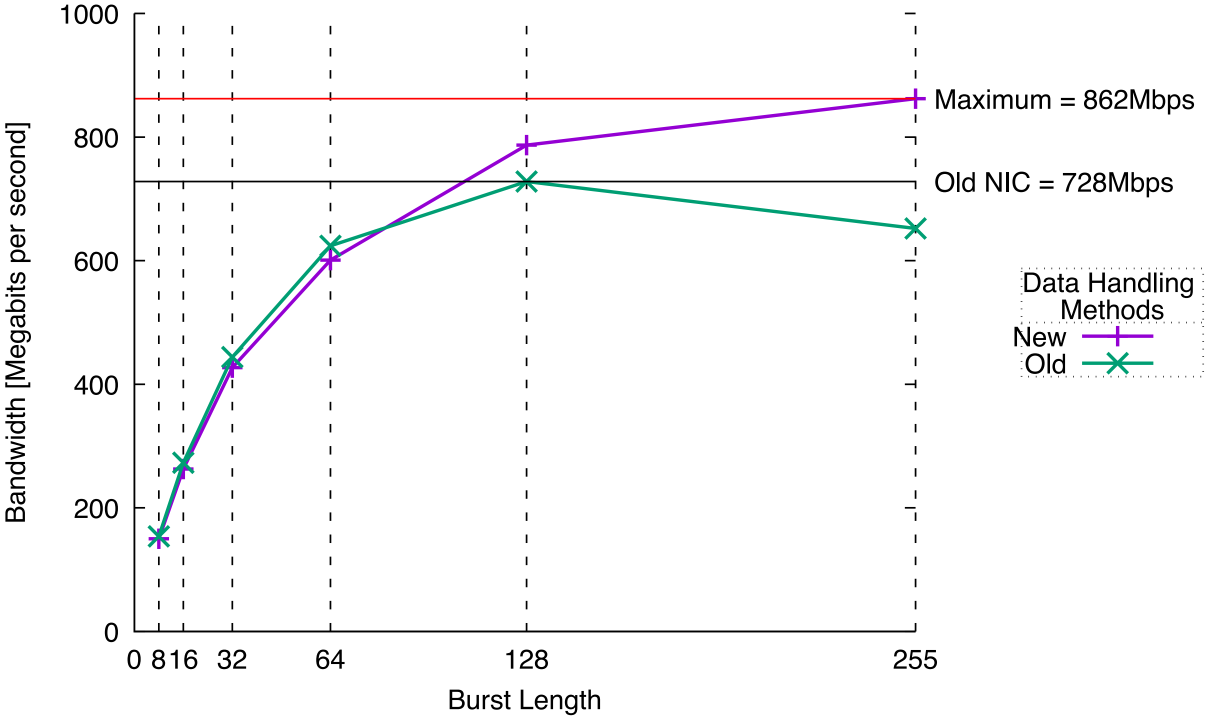


(b) Transmitting Packets

Figure 5.4: Performance comparison of the newly developed data handling architecture with the ability to store 8 Transmit Buffer Descriptors and 10 Receive Buffer Descriptors for different poll intervals

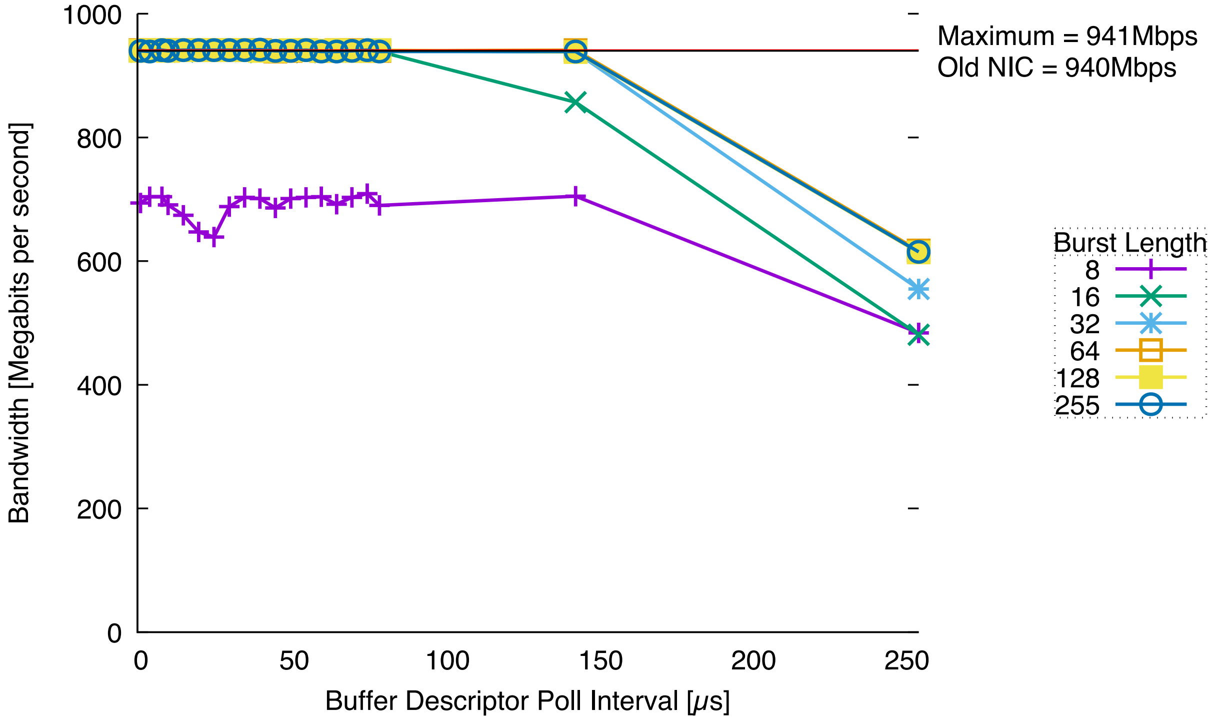


(a) Receiving Packets

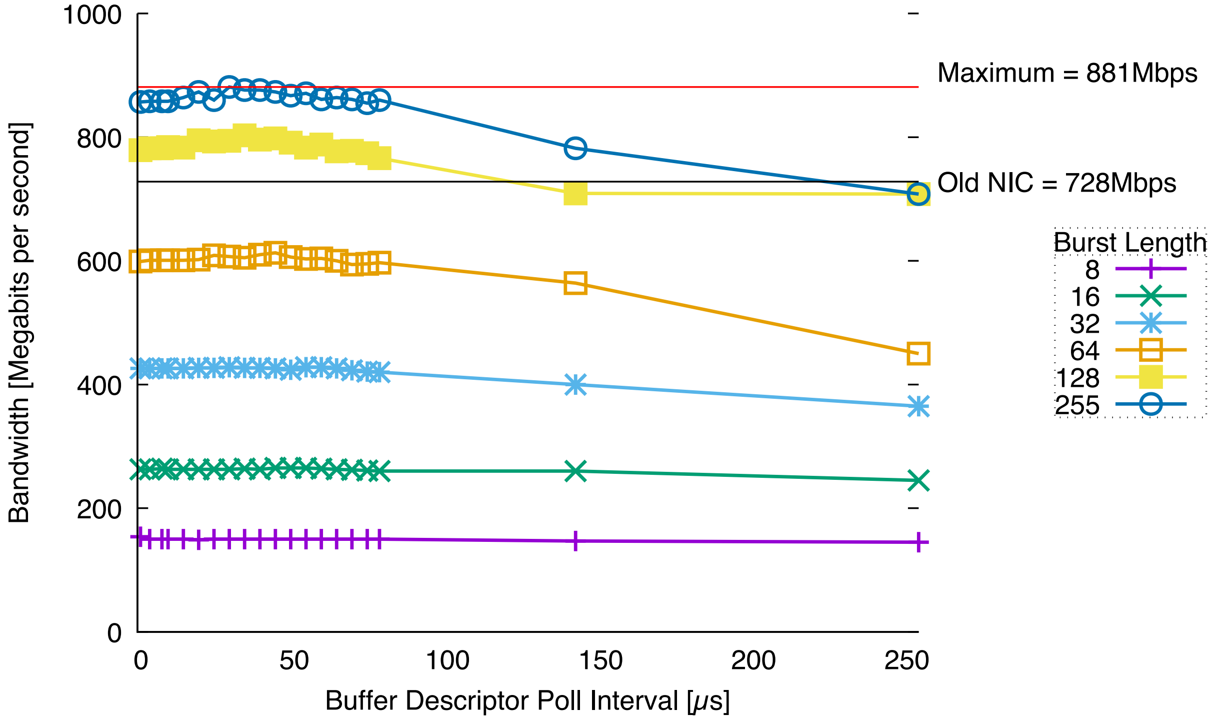


(b) Transmitting Packets

Figure 5.5: Comparison of the best bandwidth achieved by the developed data handling architecture, capable to store 8 Transmit Buffer Descriptors and 10 Receive Buffer Descriptors, against the old data handling method.

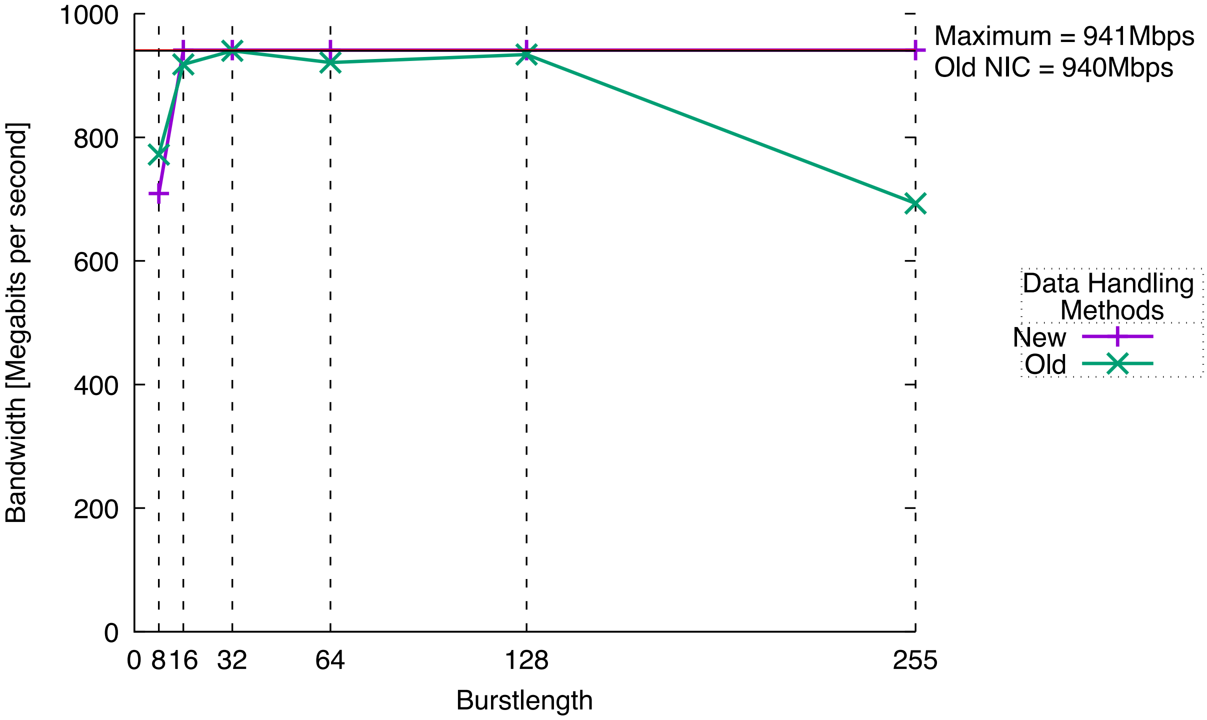


(a) Receiving Packets

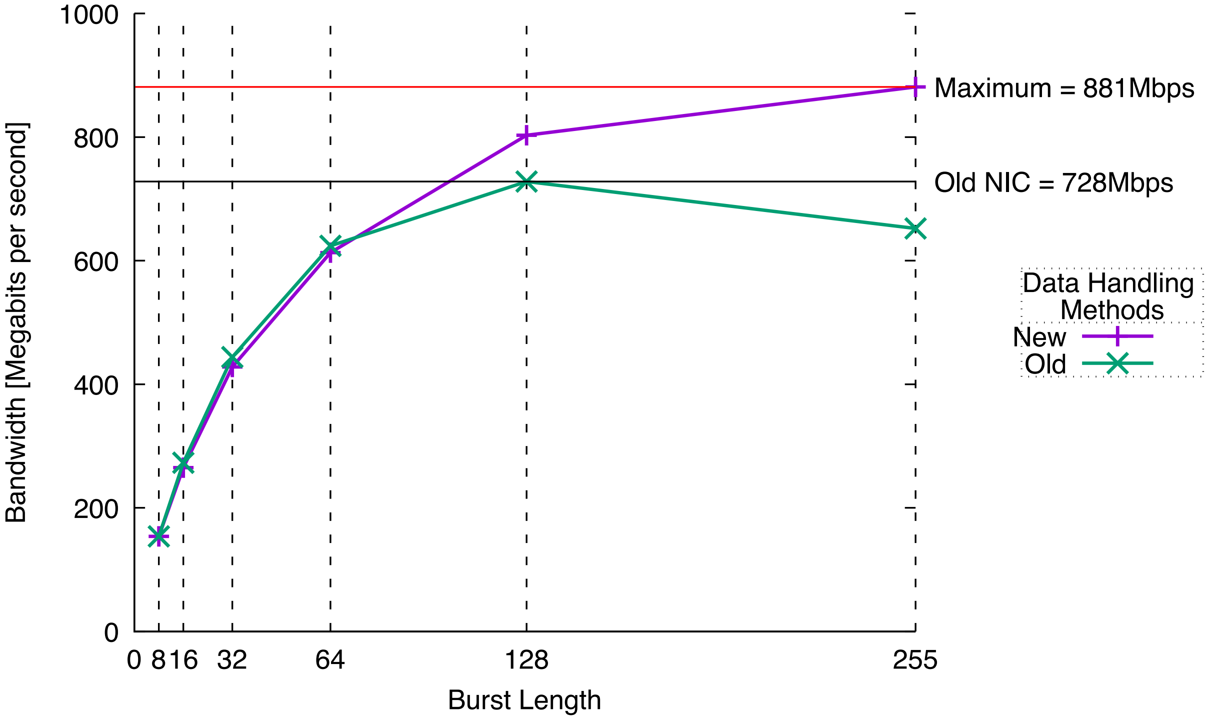


(b) Transmitting Packets

Figure 5.6: Performance comparison of the newly developed data handling architecture with the ability to store 16 Transmit Buffer Descriptors and 21 Receive Buffer Descriptors for different poll intervals

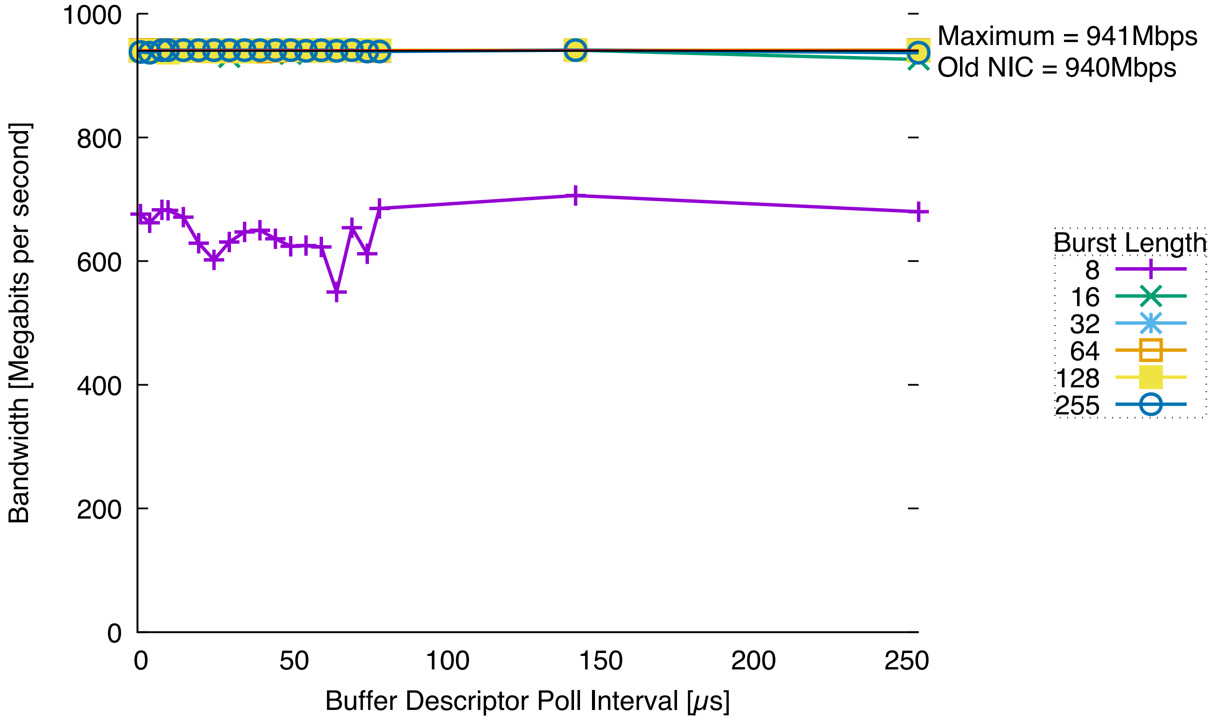


(a) Receiving Packets

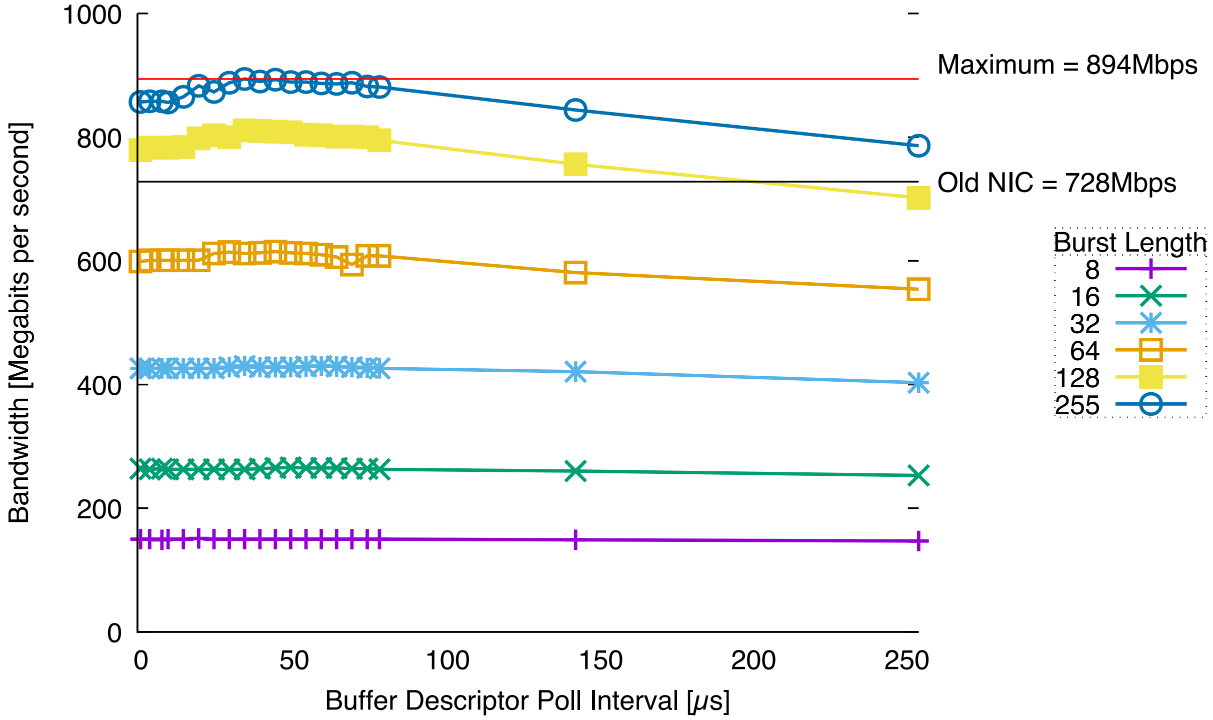


(b) Transmitting Packets

Figure 5.7: Comparison of the best bandwidth achieved by the developed data handling architecture, capable to store 16 Transmit Buffer Descriptors and 21 Receive Buffer Descriptors, against the old data handling method.

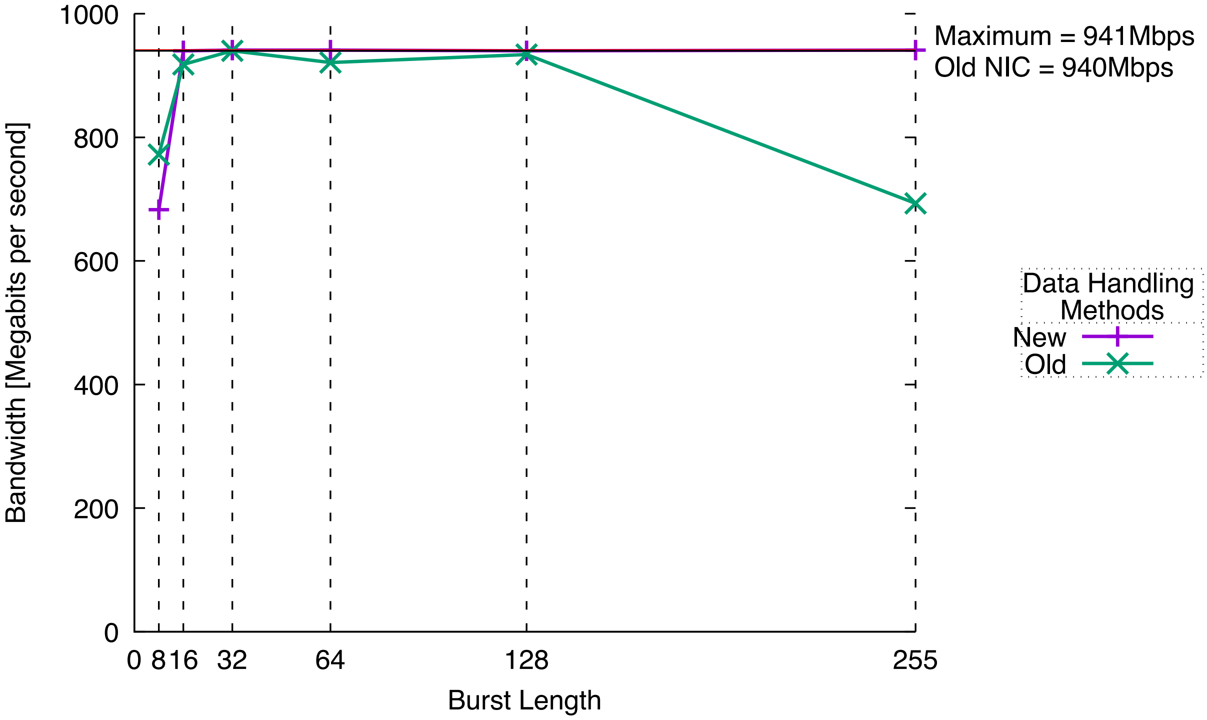


(a) Receiving Packets

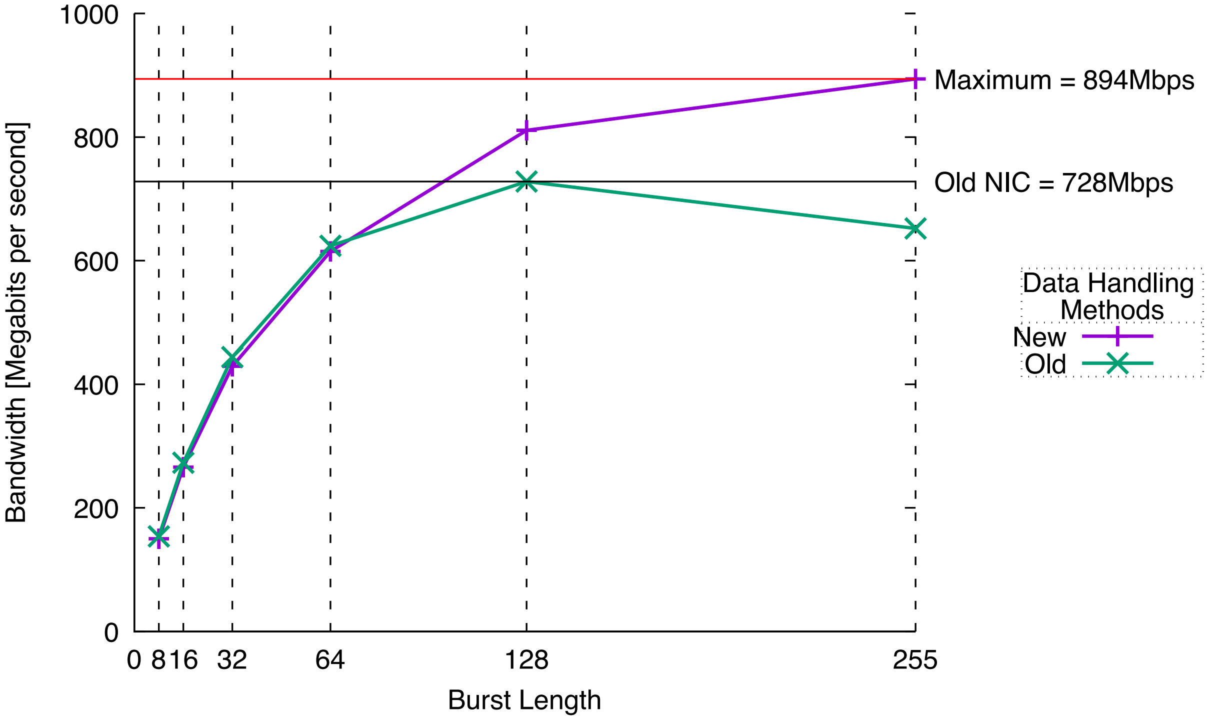


(b) Transmitting Packets

Figure 5.8: Performance comparison of the newly developed data handling architecture with the ability to store 32 Transmit Buffer Descriptors and 42 Receive Buffer Descriptors for different poll intervals



(a) Receiving Packets



(b) Transmitting Packets

Figure 5.9: Comparison of the best bandwidth achieved by the developed data handling architecture, capable to store 32 Transmit Buffer Descriptors and 41 Receive Buffer Descriptors, against the old data handling method.

The best performance is achieved if the burst length is set to the maximum value 255 and the register BD_POLL_TIMEOUT is between 15 and 60μs. Figure 5.10 compares the newly developed architecture and the old implementation with the maximum burst length set.

The receiving performance of the new data handling method has decreased for small burst lengths, but is constant at the maximum value for burst lengths equal or greater than 16. The performance of the old method decreases if the burst length is set to a value higher than 128, due to the implementation of burst transactions. The old design always initiates a burst transaction with the maximum burst length and writes dummy data or doesn't use the read data, if it has no data to write or doesn't need the read data.

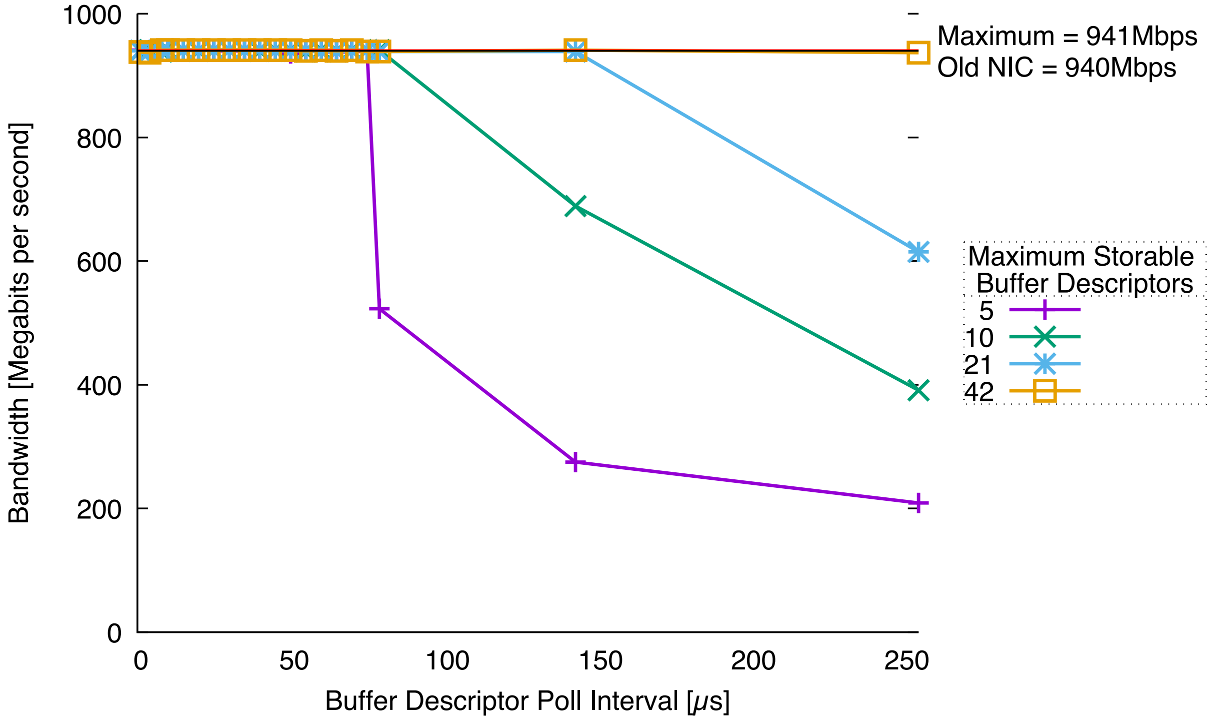
The performance for sending packets is higher if the burst length of the system is greater than 128. For lower values, the design performs nearly as good as the old one. The performance boost for transmitting packets is because multiple transmit packets can be loaded from the memory during one burst access, if the poll timeout is set correctly. The performance of the old data handling method also degrades for the maximum burst length. The reason for this is also the implementation of the burst transaction.

The performance for receiving and transmitting packets are very different. Transferring received packets from the network card to the memory has a simpler data flow than transmitting packets. Multiple free RX BDs can be stored and used, if a packet gets received. The NIC can always save received data to the main memory as long as free RX BDs are available. After the data has been saved to the main memory and the meta information of the sent packet have been stored, the NIC sends an interrupt and the CPU processes the packets.

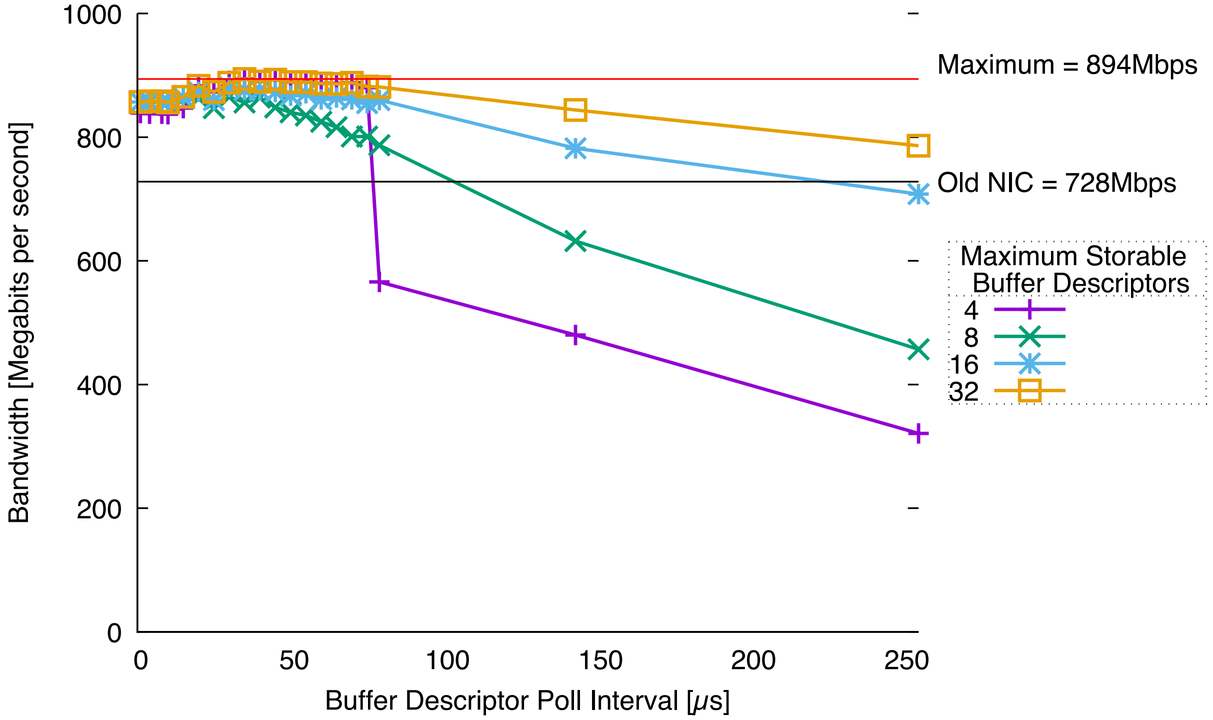
For transferring packets the system has to create TX BDs which have to be polled by the network card. Once a TX BD is available on the NIC, the packet data has to be loaded from the memory to the TX FIFO. The packet is then sent by the NIC and the meta information about the transmission is then returned to the data handling module. The meta information is then saved to the main memory and the CPU is informed by an interrupt that data was sent.

Interrupts, which are used to inform the CPU about an event on the NIC, are sent via the PCI Express bus. The device driver of the NIC has to read from the register MAC_INT_SOURCE to fetch the reason for the NIC's interrupt. The interrupt sending and reading data from the interrupt source register takes very long, because every register read takes a minimum of 1μs. The time for reading and writing register data highly depends on the workload of the PCI Express bus.

The tool $iperf$ was used to transfer packets with the Transmission Control Protocol (TCP). TCP controls the transmission of packets by using congestion control. Only if data items are acknowledged, new data is prepared and handed over to the device driver of the NIC. If the acknowledgement is not given in time to the operating system, no additional data is prepared and the transmission is slowed down. This could be the reason for the performance difference between receiving and transmitting packets.



(a) Receiving Packets



(b) Transmitting Packets

Figure 5.10: Comparison of the different architectures with the maximum burst length set

5.3 CPU Usage

Every time an application has to send a packet or a packet is received by the NIC, the CPU is triggered to process the packet. The time used by the CPU to process a packet is measured directly in the kernel. The kernel timestamp function *ktime_get_ts* of Linux is used for getting the current time in nanoseconds. The start and end time of the following functions are measured:

- Creating a packet for sending
- Process sent packet
- Process received packet
- Time between interrupt and packet processing

Creating a Packet

Figure 5.11 gives an overview of the time needed for creating a packet with the old and the new data handling method. The durations for packet creation are grouped in 50ns bins and the number of packets in this range are counted.

The newly developed data handling method, which writes the buffer descriptor to the main memory and not directly to the NIC, reduces for all investigated functions. Table 5.1 shows the mean, median, maximum and minimum value of the measured packet creation times. For the former packet creation, the time the buffer descriptor was transferred to the NIC highly influences the time used for packet creation. Now the time used for copying the data from one memory location to another location defines the time used for creating a packet.

	Mean[ns]	Median[ns]	Min[ns]	Max[ns]
Old strategy creating 98 Byte Packet	1993,33	2024	1633	2373
Old strategy creating 1500 Byte Packet	2264,30	2243	2088	3267
New Strategy creating 98 Byte Packet	319,97	306	267	490
New Strategy creating 1500 Byte Packet	922,99	896	758	4156

Table 5.1: Statistical indicators of the CPU time used for creating a packet

Time Between Interrupt and Packet Processing

The time between the CPU starts to process an PCI Express interrupt triggered by the NIC and the packet is processed by the received packet or sent packet function is illustrated in Figure 5.12. The statistical indicators are shown in Table 5.2.

	Mean[ns]	Median[ns]	Min[ns]	Max[ns]
Time between Interrupt and Packet Processing	6629,23	6412	6093	13014

Table 5.2: Statistical indicators of the time between an interrupt is processed by the device driver of the NIC and packet processing

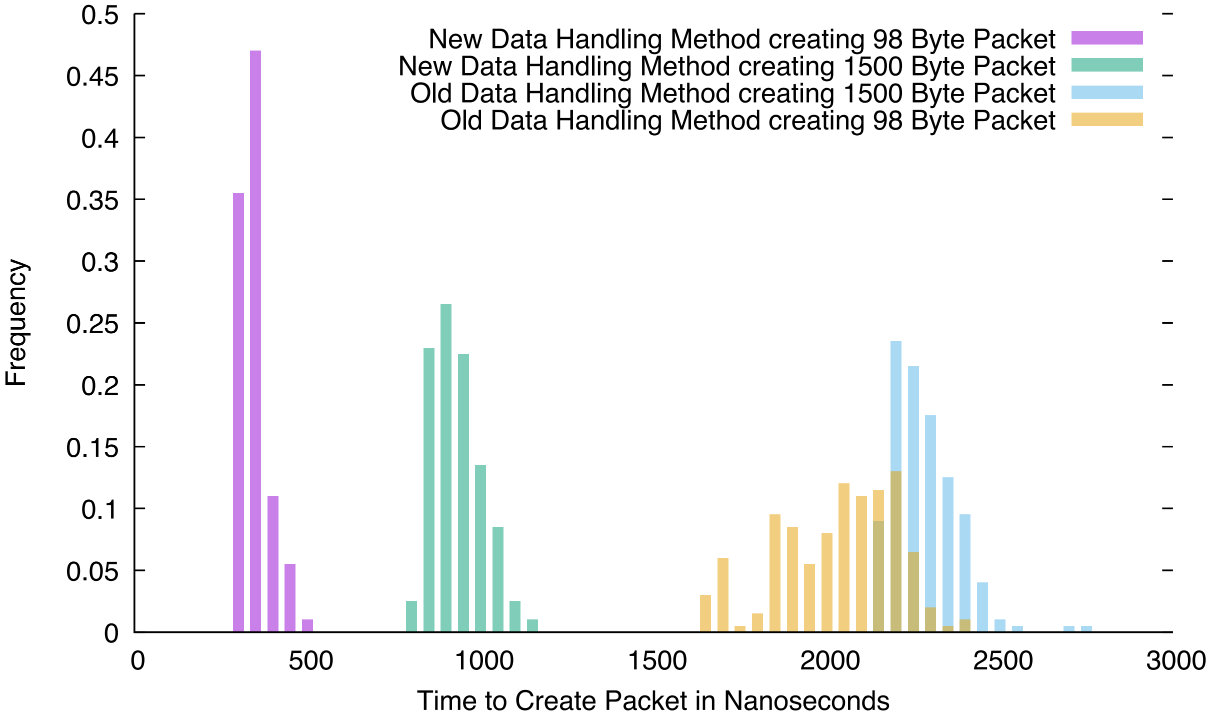


Figure 5.11: Histogram of CPU time used for creating a packet

Most of the function's execution time is spent on reading register values from the NIC. The interrupt only transfers a notification about an event on the NIC, but doesn't include any additional information. The device driver has to fetch additional data from NIC registers to get further information. As described in the former section, reading a register takes about 1us. Because the priority of the interrupt handling function is higher than the priority of the other packet processing functions, the execution of the function is not varying that much.

Process Sent Packet

The NIC sends an interrupt after a packet has been sent. The CPU has to read the buffer descriptor of the sent packet to get information about the data transmission and also has to inform the application, which issued the data sending, about the successful transmission. The CPU time used by the driver to process sent packets is illustrated in Figure 5.13. Table 5.3 shows statistical indicators of the values illustrated in Figure 5.13.

	Mean[ns]	Median[ns]	Min[ns]	Max[ns]
Old strategy processing one sent packet	6189,07	5179	1747	34403
Old strategy processing two sent packets	8840,49	8512	3311	25907
New Strategy processing one sent packet	693,09	641	463	2415
New Strategy processing two sent packets	859,98	835	604	1423

Table 5.3: Statistical indicators of the CPU time used to process s sent packet

Reading the TX BDs from the NIC highly influences the processing time of the old data handling strategy for this function. The function process time of the optimised strategy is influenced by

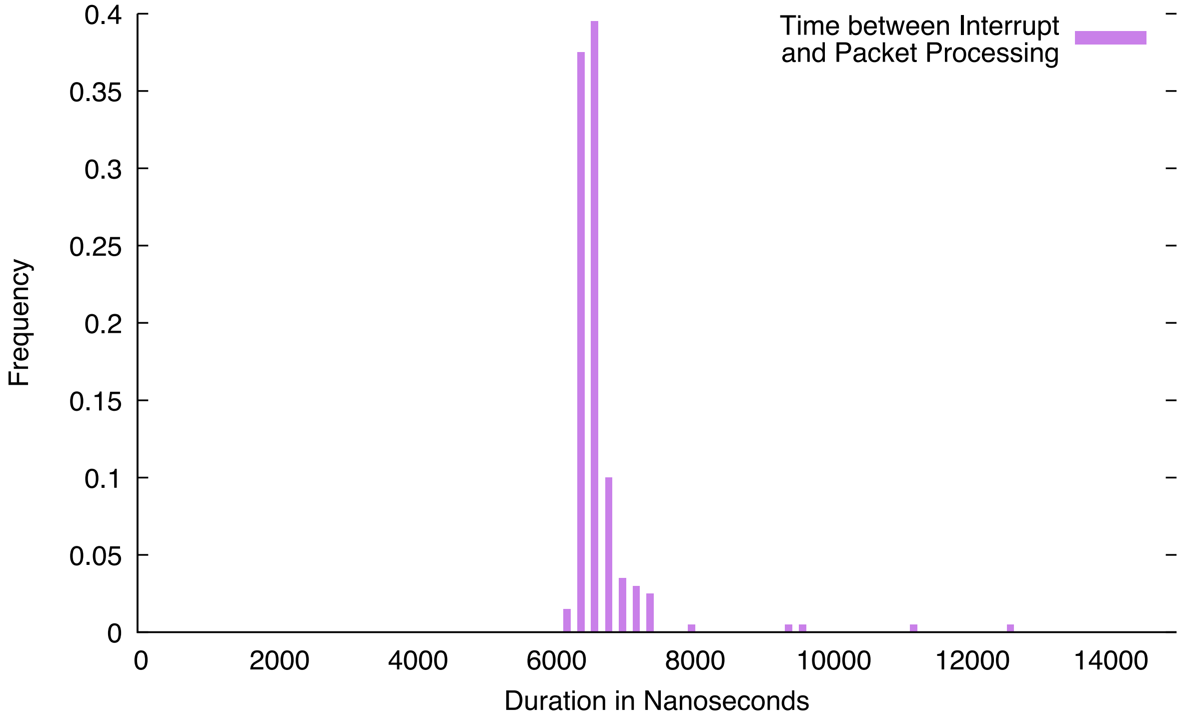


Figure 5.12: Histogram of time between interrupt occurrences on the NIC and packet is processed

the number of processed packets and the availability of the memory and the CPU.

Process Received Packet

The NIC directly transfers received packet data to the main memory before an interrupt to the CPU is triggered. The system's device driver then forwards the packet data to the destination application. Figure 5.14 illustrates the time used to handle the received packet. Table 5.4 shows the statistical indicators of the values illustrated in Figure 5.14.

	Mean[ns]	Median[ns]	Min[ns]	Max[ns]
Old strategy processing one received Packet	8051,59	7458	3549	25639
Old strategy processing two received Packets	13966,61	13128	5738	27567
New Strategy processing one received Packet	1234,22	1176	893	2626
New Strategy processing two received Packets	2077,79	1962	1607	8115

Table 5.4: Statistical indicators of the CPU time used to process received packet

The RX BDs fetching from the NIC is affecting the used CPU time of the old strategy enormously. The duration of the new data handling strategy handling the received data, is defined by the amount of packet data, which has to be copied to the memory location of the destination application.

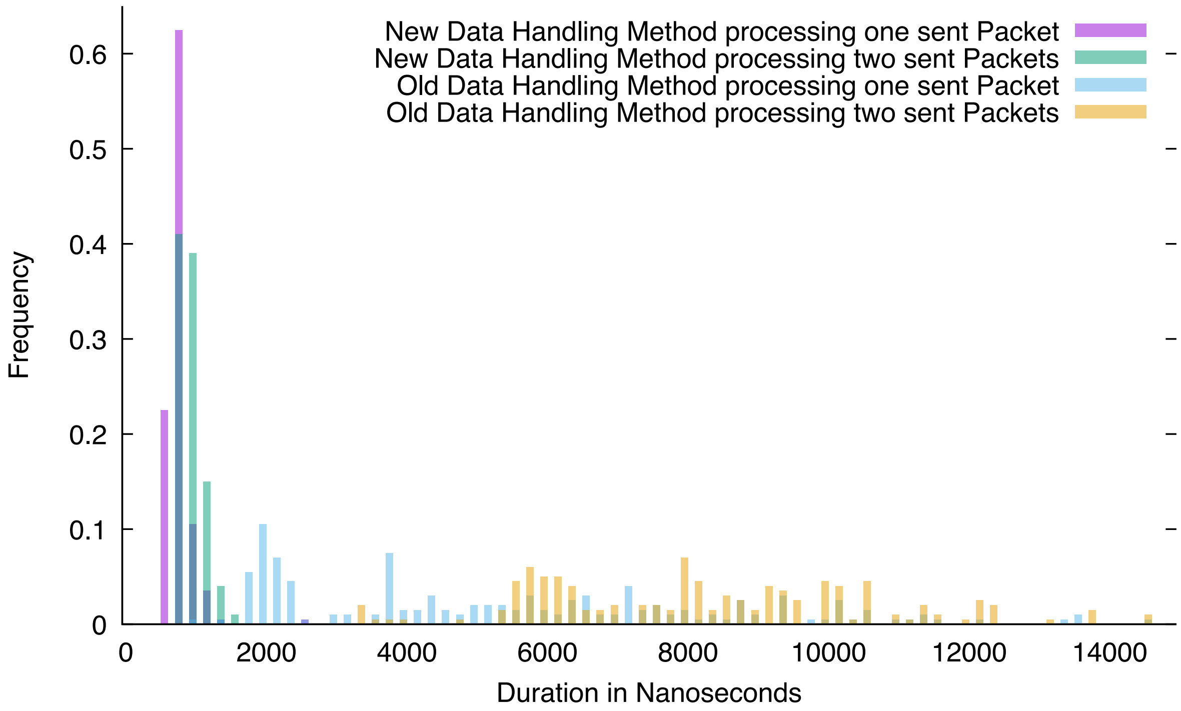


Figure 5.13: Histogram CPU time used to process sent packet

CPU Performance Analysis

The packet processing time of data is reduced by moving the BDs to the main memory enormously. The minimum duration for reading or writing register data from the NIC is not deterministic and varies from less than one to multiple μs. Depending on the usage of the PCI Express bus, the reading and writing durations are changing.

The processing time for setting up a 1500 byte Ethernet packet to send is reduced from 1993,33ns to 922,99ns and for handling the sent interrupt the duration is decreased from 6189,07ns to 693.09ns. The overall processing time for one sent packet is reduced from 8182,40ns to 1616,08ns which is a reduction of 80,2% of the CPU processing time.

The processing of received packets with the old data handling strategy took 8051,59ns while the new strategy only needs 1234,22ns. 84,67% of the CPU processing time are saved with the new strategy.

The function, necessary for getting the NIC's interrupt source, is using 6629,23ns. This function reads interrupt registers from the NIC and triggers afterwards the sent or received packet processing. The time used for processing the interrupt is not solved by the new data handling strategy. The interrupt handling from the NIC to the system is a problem of the interrupt functionality implemented in the PCI Express IP core of the NIC.

The variation of processing time for the receive and the sent process is because the CPU is halted during register reads and the processor is doing a context switch to do other tasks. The interrupt handling process time hasn't that variation, because this function is running with a higher priority. Higher prioritised functions get faster reactivated than lower prioritised ones.

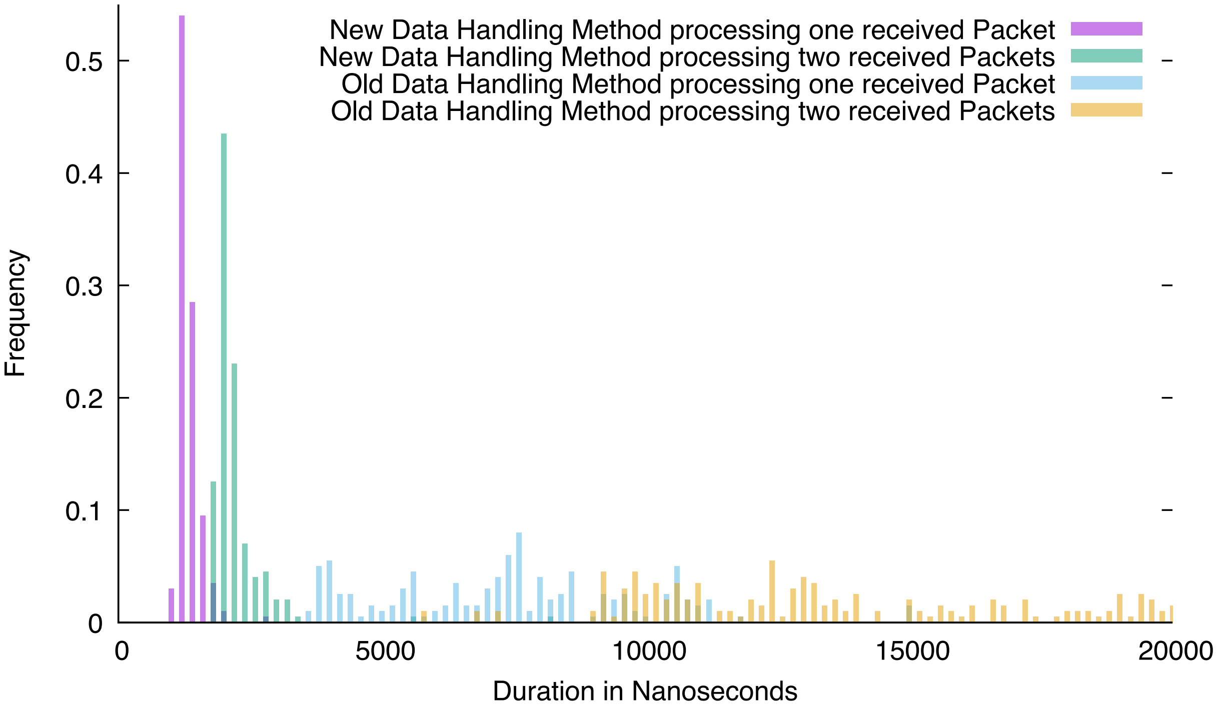


Figure 5.14: Histogram of CPU time used to process received packet

6 Conclusion and Outlook

A new strategy for handling data from a host system to a peripheral unit is presented in this thesis. Research in the area of high-performance data exchange is demanded by the increasing data link speed of today's communication networks.

This work analyses the components necessary for transferring data from the CPU to a peripheral unit. Two IP cores, used for handling data from a CPU to a peripheral unit, are analysed with respect to their efficiency in transferring data. Data handling implementations of two network interfaces are described and analysed.

The created data handling method aims to minimize the CPU usage, the requests on the memory and the requests on the shared bus system. The data exchange between the CPU and the NIC is accomplished by reading and writing to the main memory.

The technology-independent digital functionality of the strategy was implemented with the hardware description language VHDL. The functionality of the modules is split into processes, which are representing the implementation of the design. The software part is written as device driver for Linux.

With this solution, the performance for transferring data on a 1-lane PCI Express Ethernet card is improved. The bandwidth for receiving packets is as high as the former implementation but the transmission rate was improved from 782Mbps to 894Mbps. The CPU time used for creating packets, handling sent and received interrupts is reduced, because the meta information of each packet is not stored directly on the NIC but on the main memory. The time used for creating a packet is reduced from 2264ns to 922ns. The processing time for acknowledging a sent packet is reduced from 6189ns to 693ns. The time for processing received packets by the device driver is reduced from 8051ns to 1234ns.

The concept of minimising the bus requests by moving the meta information storage from the NIC to the main memory of the systems leads to an enormous reduction in CPU usage. These modifications enable the usage of more performant network adapters, like 10 gigabit network interfaces.

Outlook

The presented solution improves the data handling strategy between the CPU and a peripheral unit by fetching multiple meta information, represented as buffer descriptors, from the main memory at once. Updating multiple on the NIC stored buffer descriptors is suggested as future work, to fully utilise the bus system for writing used buffer descriptors back to the main memory.

Measurements have shown that every time an interrupt is triggered, the system needs 6629ns before packet processing is started. If the system only triggers an interrupt, when multiple packets have been transferred, the speed of data processing could be increased. The time used for processing one and two sent packet is 693ns and 859ns. The time used for setting up a separate interrupt handling procedure is much higher than processing two packets at once. Processing two packets with one interrupt reduces the CPU time used by 95%. If the system processes even more packets, CPU time could be reduced further.

The implemented interrupt functionality of the PCI Express IP Core generates the same interrupt for every interrupt source. The PCI Express core sends the interrupt as packet information to the CPU. The device driver has to read two interrupt source registers to get the source of the interrupt. The interrupt implementation of the PCI Express IP Core can be redesigned to send the interrupt source with the interrupt to the CPU. This would reduce the CPU time used reading the data from the NIC and also increases the utilisation of the bus.

Literature

[BA13] Ben Abdallah, Abderazek: *Multicore Systems On-Chip: Practical Software/Hardware Design : 2nd Edition*. 2nd ed. Paris : Atlantis Press : Imprint: Atlantis Press, 2013 (Atlantis ambient and pervasive intelligence ; v. 7). 10.2991/978-94-91216-92-3. – ISBN 94–91216–92–9

[Cad17] Cadek, Gerhard: *syn1588® PCIe NIC Revision 2.1 – Data Sheet*. http://www.oreganosystems.at/download/syn1588_pcie_nic_ds.pdf. Version: März 2017

[Ela18] Elahi, Ata: *Computer Systems : Digital Design, Fundamentals of Computer Architecture and Assembly Language*. Cham : Springer International Publishing Imprint: Springer, 2018 10.1007/978-3-319-66775-1. – ISBN 3–319–66775–0

[GK83] Gajski, D. D. ; Kuhn, R. H.: Guest Editors' Introduction: New VLSI Tools. In: *Computer* 16 (1983), Dezember, Nr. 12, S. 11–14. http://dx.doi.org/10.1109/MC.1983.1654264. – DOI 10.1109/MC.1983.1654264. – ISSN 0018–9162

[Hop06] Hoppe, Bernhard: *Verilog : Modellbildung für Synthese und Verifikation*. München Wien : Oldenbourg, 2006. – ISBN 3–486–58004–3

[KB09] Kesel, Frank ; Bartholomä, Ruben [.: *Entwurf von digitalen Schaltungen und Systemen mit HDLs und FPGAs*. 3., korr. u. aktualisierte Aufl. München : Oldenbourg, 2009. – ISBN 978–3–486–73181–1

[LL14] Larsen, Steen ; Lee, Ben: Survey on System I/O Hardware Transactions and Impact on Latency, Throughput, and Other Factors-Chapter Two. In: *Advances In Computers* Bd. 92. 2014. – ISBN 978–0–12–420232–0, S. 67–104

[Ltd18a] Ltd., Xillybus: *Xillybus - Getting started with Xillinux for Zynq-7000*. http://xillybus.com/downloads/doc/xillybus_getting_started_zynq.pdf. Version: 2018

[Ltd18b] Ltd., Xillybus: *Xillybus host application programming guide for Linux*. http://xillybus.com/downloads/doc/xillybus_host_programming_guide_linux.pdf. Version: 2018

[Moo06] Moore, G. E.: Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. In: *IEEE Solid-State Circuits Society Newsletter* 11 (2006), September, Nr. 3, S. 33–35. http://dx.doi.org/10.1109/N-SSC.2006.4785860. – DOI 10.1109/N–SSC.2006.4785860. – ISSN 1098–4232

[noa09] IEEE Standard VHDL Language Reference Manual. In: *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)* (2009), Januar, S. c1–626. http://dx.doi.org/10.1109/IEEESTD.2009.4772740. – DOI 10.1109/IEEESTD.2009.4772740

[noa10] *PCI Express® Base Specification Revision 3.0.* November 2010

[noa18] Avalon® Interface Specifications. (2018), 60. `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/mnl_avalon_spec.pdf`

[P03] P, Ravi: *PCI express system architecture.* Boston : Addison-Wesley, 2003. – ISBN 0–321–15630–7

[RDK$^+$00] Rixner, Scott ; Dally, William J. ; Kapasi, Ujval J. ; Mattson, Peter ; Owens, John D.: Memory Access Scheduling. In: *Proceedings of the 27th Annual International Symposium on Computer Architecture.* New York, NY, USA : ACM, 2000 (ISCA '00). – ISBN 1–58113–232–8, 128–138

[Rei09] Reichardt, Jürgen: *Lehrbuch Digitaltechnik : eine Einführung mit VHDL.* München : Oldenbourg, 2009 `10.1524/9783486593600`. – ISBN 3–486–59360–9

[SGLAJLD17] Sánchez-Garrido, J. ; López-Antequera, A. M. ; Jiménez-López, M. ; Díaz, J.: Sub-nanosecond Synchronization over 1G ethernet data links using white rabbit technologies on the WR-ZEN board. In: *2017 40th International Conference on Telecommunications and Signal Processing (TSP)*, 2017, S. 688–693

[SM15] Stallings, William ; Manna, Moumita M.: *Operating systems : internals and design principles.* Global edition, eighth edition. Boston München : Pearson Education Limited, 2015 (Always learning). – ISBN 1–292–06194–4

[Sol14] Solomon, Richard: PCI Express® Basics & Background. (2014), 45. `https://pcisig.com/sites/default/files/files/PCI_Express_Basics_Background.pdf`

[TB16] Tanenbaum, Andrew S. ; Bos, Herbert: *Moderne Betriebssysteme.* 4., aktualisierte Auflage. Hallbergmoos : Pearson, 2016 (Always learning). – ISBN 3–86894–270–X

[VKVF16] Vesper, M. ; Koch, D. ; Vipin, K. ; Fahmy, S. A.: JetStream: An open-source high-performance PCI Express 3 streaming library for FPGA-to-Host and FPGA-to-FPGA communication. In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016, S. 1–9

[VN14] Véstias, M. ; Neto, H.: Trends of CPU, GPU and FPGA for high-performance computing. In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, S. 1–6