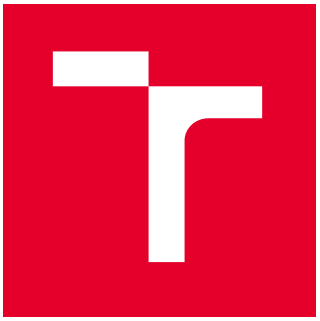


BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF MICROELECTRONICS

ÚSTAV MIKROELEKTRONIKY

HIGH-SPEED PACKET ACCUMULATION IN FPGA

VYSOKORYCHLOSTNÍ AKUMULACE PAKETŮ V FPGA

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. David Beneš**
AUTOR PRÁCE

SUPERVISOR — **Ing. Vojtěch Dvořák, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Mikroelektronika**

Ústav mikroelektroniky

Student: Bc. David Beneš ***ID:*** 220860

Ročník: 2 ***Akademický rok:*** 2023/24

NÁZEV TÉMATU:

Vysokorychlostní akumulace paketů v FPGA

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s open-source platformou NDK a sběrnicí MFB (Multi Frame Bus). Pro platformu NDK navrhněte obvod pracující s pakety na sběrnici MFB a implementujte ho ve zvoleném jazyce HDL. Cílem tohoto obvodu bude efektivně spojovat malé pakety ze stejných kanálů do větších celků (super-paketů). Propustnost implementovaného obvodu by se měla v ideálních případech blížit k 400 Gb/s. Funkčnost implementovaného řešení ověřte UVM verifikací. V závěru práce zhodnoťte dosažené výsledky a uveďte výhody a nevýhody zvoleného řešení. Pro realizaci návrhu lze použít sadu skriptů a již existujících komponent v databázi společnosti CESNET z.s.p.o.

DOPORUČENÁ LITERATURA:

According to recommendations of supervisor

Termín zadání: 5.2.2024 ***Termín odevzdání:*** 21.5.2024

Vedoucí práce: Ing. Vojtěch Dvořák, Ph.D.

doc. Ing. Lukáš Fujcik, Ph.D.
předseda rady studijního programu

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Master's Thesis

Master's study program **Microelectronics**

Department of Microelectronics

Student: Bc. David Beneš ***ID:*** 220860

Year of study: 2 ***Academic year:*** 2023/24

TITLE OF THESIS:

High-speed packet accumulation in FPGA

INSTRUCTION:

Study the open-source NDK platform and the MFB (Multi Frame Bus). For the NDK platform, design a circuit working with packets on the MFB bus and implement it in the selected HDL language. The goal of this circuit is to efficiently combine small packets from the same channels into larger units (super-packets). The throughput of the implemented circuit should ideally be reach 400 Gb/s. Verify the functionality of the implemented solution by the UVM verification. In the Master's thesis conclusion, evaluate the results achieved and state the advantages and disadvantages of the chosen solution. A set of scripts and already existing components in the database of CESNET z.s.p.o. can be used to implement the design.

RECOMMENDED LITERATURE:

According to recommendations of supervisor

Date of project specification: 5.2.2024 ***Deadline for submission:*** 21.5.2024

Supervisor: Ing. Vojtěch Dvořák, Ph.D.

doc. Ing. Lukáš Fujcik, Ph.D.
Chair of study program board

Abstract

This paper presents the concept of a digital circuit that has the potential to reduce the transmission overhead on the communication link between a high-speed network card with FPGA and a host PC for small packets. This circuit is specifically designed for the NDK platform developed by CESNET z.s.p.o., which is specified in the first chapter. The motivation for writing this thesis is presented in the second chapter, which is dedicated to the communication path between the host PC and the FPGA. The design of the resulting digital circuit and its testing is described in the final part of this thesis.

Keywords

CESNET, NDK, MFB, 400 Gbps, FPGA, PCIe, DMA, Data shifts

Abstrakt

Tato práce popisuje návrh číslicového obvodu, který má potenciál snížit režii přenosu malých paketů na komunikační lince mezi vysokorychlostní síťovou kartou s FPGA a hostitelským počítačem. Tento obvod je určen speciálně pro platformu NDK vyvinutou sdružením CESNET z.s.p.o., proto je první kapitola věnována její specifikaci. Motivace k sepsání této práce je popsána v následující kapitole, která je věnována komunikační lince mezi hostitelským počítačem a FPGA. Poslední část popisuje návrh číslicového obvodu a jeho testování jak z pohledu funkčnosti, tak z propustnosti.

Klíčová slova

CESNET, NDK, MFB, 400 Gbps, FPGA, PCIe, DMA, Posuvy dat

Rozšířený abstrakt

Přenosové rychlosti dnešních počítačových sítí dosahují několika stovek gigabitů za sekundu. To otevírá nové možnosti pro aplikace, které vyžadují rychlý a snadný přenos velkých objemů dat, ale zároveň klade velké nároky na zpracování takového provozu. Jedním z nástrojů, které dokážou takovýto provoz zpracovat, jsou síťové karty založené na čipech FPGA. Jejich konfigurovatelnost umožňuje vytvořit *firmware* podle cílových požadavků, nebo pro specifické aplikace upravit firmware již existující. V případě této diplomové práce se jedná o platformu NDK (*Network Development Kit*) vytvořenou a udržovanou sdružením CESNET z.s.p.o., která poskytuje prostředí pro práci se síťovým provozem na FPGA.

V rámci této diplomové práce je navržen číslicový obvod pro zpracování paketů na sběrnici MFB (*Multi-Frame Bus*). Toto zpracování spočívá v efektivním spojení vícero malých paketů ze stejného DMA kanálu do většího celku. Samotná sběrnice MFB je určena pro přenos dat mezi dílčími komponentami v rámci NDK. Aby byl využit plný potenciál paralelního zpracování na čipu FPGA, umožňuje tato sběrnice přenášet vícero paketů ve stejném hodinovém taktu. K dosažení požadované propustnosti 400 Gb/s je MFB konfigurována tak, aby byla schopna přenést čtyři nejkratší Ethernetové rámce najednou při pracovní frekvenci 200 MHz.

Motivací pro tvorbu takového obvodu je přenosová rychlost pro malé pakety mezi síťovou kartou a hostitelským počítačem. Tato rychlost může v rámci NDK klesnou až o několik desítek procent z důvodu velké režie přenosu. Ta je způsobena několika faktory, z nichž ty nejvýznamnější jsou využití sběrnice PCIe a typ použitého DMA řadiče. Abychom tedy správně využili dostupné prostředky, je vhodné odesílat větší celky a jejich obsah extrahovat na procesoru hostitelského počítače. V rámci NDK se tyto větší celky nazývají Super-Pakety, a aby jejich zpracování nebylo příliš složité, jsou koncipovány tak, aby mezi jednotlivými dílčími pakety byly minimální mezery.

Teoretická část této práce čtenáře nejdříve seznámí s architekturou NDK a se základním principem komunikace přes sběrnici PCIe pomocí DMA přenosů. V rámci první kapitoly, věnující se NDK je uveden přehled jednotlivých komponent, ze kterých se platforma skládá a specifikace datových sběrnic, které jsou pro ni typické. Jedná se především o sběrnice MFB, MVB (*Multi-Value Bus*) a MI (*Memory Interface Bus*). Druhá kapitola se zabývá komunikací přes PCIe a zaměřuje se na popis režie datového přenosu jak přes samotnou sběrnici PCIe, tak i v rámci řadiče DMA. V praktické části této práce jsou definovány požadavky na cílový obvod a jeho samotný návrh. Kapitola věnující se návrhu číslicového obvodu je rozdělena do několika částí, počínaje popisem obecné funkčnosti a konče popisem jednotlivých částí obvodu. Následující kapitola se věnuje funkční verifikaci, která ověřuje funkčnost obvodu dle definovaných požadavků, a testům propustnosti. Tyto testy jsou rozděleny na dvě části – první část je věnována měření propustnosti v rámci verifikace, která ověřuje, zda výsledný číslicový obvod dokáže pracovat bez nutnosti brzdit vstupní rozhraní; druhá část se věnuje měření propustnosti na reálné síťové kartě. Výsledky těchto testů ukazují pozitivní dopad na propustnost malých paketů a jsou shrnuty v poslední kapitole.

Bibliographic citation

BENEŠ, David. *Vysokorychlostní akumulace paketů v FPGA* [online]. Brno, 2024 [cit. 2024-05-09]. Dostupné z: https://www.vut.cz/studenti/zav-prace/detail/159926. Diplomová práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav mikroelektroniky. Vedoucí práce Vojtěch Dvořák.

Author's Declaration

Author:	**Bc. David Beneš**
Author's ID:	**220860**
Paper type:	**Master's Thesis**
Academic year:	**2023/24**
Topic:	**High-speed packet accumulation in FPGA**

I declare that I have written this paper independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the project and listed in the comprehensive bibliography at the end of the project.

As the author, I furthermore declare that, with respect to the creation of this paper, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation S 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll., Section 2, Head VI, Part 4.

Brno, May 20, 2024

author's signature

Acknowledgement

I would like to thank my supervisor Ing. Vojtěch Dvořák Ph.D. for his invaluable advice and support. I would also like to thank Ing. Jakub Cabal for consulting this paper and his patience with the constant flow of questions. Finally, I would like to thank Ing. Daniel Kondys for his willingness to check the grammatical and factual aspects of this work.

Brno, May 20, 2024

Author’s signature

Contents

FIGURES

TABLES

INTRODUCTION

As internet networks become faster, they also become more vulnerable to cyber-attacks such as *Distributed Denial of Service* (DDoS) attacks. The attacks are increasingly difficult to detect due to the high-speed processing requirements of modern networks. To combat these threats, powerful tools capable of monitoring backbone networks are essential. An example of such a tool is an FPGA-based network card called XpressSX AGI-FH400G, developed by CESNET z.s.p.o. This card is powered by the *Network Development Kit* (NDK), a framework that can be used to create hardware-accelerated applications for FPGAs. These applications can analyze and aggregate network traffic for the CPU, enabling processing speeds of up to 400 Gbps.

While the NDK framework is effective, it has certain limitations, especially when transferring a large number of small network packets to the host memory. These small packets can lead to increased transmission overhead, reducing the overall throughput of the system by several tens of percent. To address the said issue, this thesis proposes the design of a digital circuit that can effectively accumulate (buffer) small packets within several virtual lanes and send them as a whole new unit, reducing the overall overhead. This circuit will be compatible with the NDK firmware, allowing seamless integration into the existing system.

This work focuses on the design of a digital circuit that can handle data packets on the NDK framework's scalable *Multi-Frame Bus* (MFB). The MFB is a high-speed point-to-point bus that can transfer multiple packets simultaneously at 200 MHz. The number of transferred packets depends on its configuration. For example, a design capable of transmitting data at 400 Gbps must be able to handle up to four smallest packets simultaneously in one clock cycle. The design must also be able to adapt to different transfer rates and resource constraints as the NDK supports various network cards. This creates a number of requirements that the proposed design must meet, including scalability, parallel processing, and effective data shifting to maximize the throughput.

The first chapter of this paper introduces the NDK platform, the framework in which the proposed solution will be implemented. This chapter focuses on the communication buses commonly used in the NDK platform, as understanding them is essential for designing this digital circuit. The second chapter describes the data transmission in the NDK system (host included) to show the cause of the performance loss in the transmission of small packets. This chapter includes a description of PCIe, its overhead, and the DMA transfers. The third chapter outlines the design requirements and the concept of the proposed digital circuit called the *Frame Packer*. In addition, this chapter also includes a more detailed description of each subcomponent within the designed architecture. The fourth chapter of this thesis is dedicated to verifying the functionality and measuring the performance of the designed component. For verification purposes, the verification plan and test environment are introduced. The last chapter provides a summary of the verification results, code coverage, and verification throughput test. In addition, the resources required implementation of this circuit are presented, as well as the impact on the throughput of the NDK system.

1 NDK FRAMEWORK

The Network Development Kit (NDK) is a framework for FPGA cards developed by CESNET z.s.p.o. It enables users to develop their own applications by providing access to various interfaces such as Ethernet, PCIe or to external memory. [1]

Most of the NDK firmware and software is open source and available on GitHub. [2] The VHDL modules used in the NDK firmware are part of the Open FPGA Modules (OFM) library. [3] As the name suggests, the OFM library is also open source, and the output of this thesis shall be included in it.

1.1 NDK Architecture

The NDK architecture for a generic FPGA card is shown in the figure below (**Figure 1.1**). It is unique in its support for FPGAs from both Intel and AMD/Xilinx, the two major chip producers. A list of currently supported cards is available on the CESNET GitHub repository.[4] Additionally, the NDK's highly scalable design enables a wide range of transfer speeds, with a current maximum of 400 Gbps. Even higher speeds will be possible in the future with sufficient FPGA resources and suitable peripherals.[5]

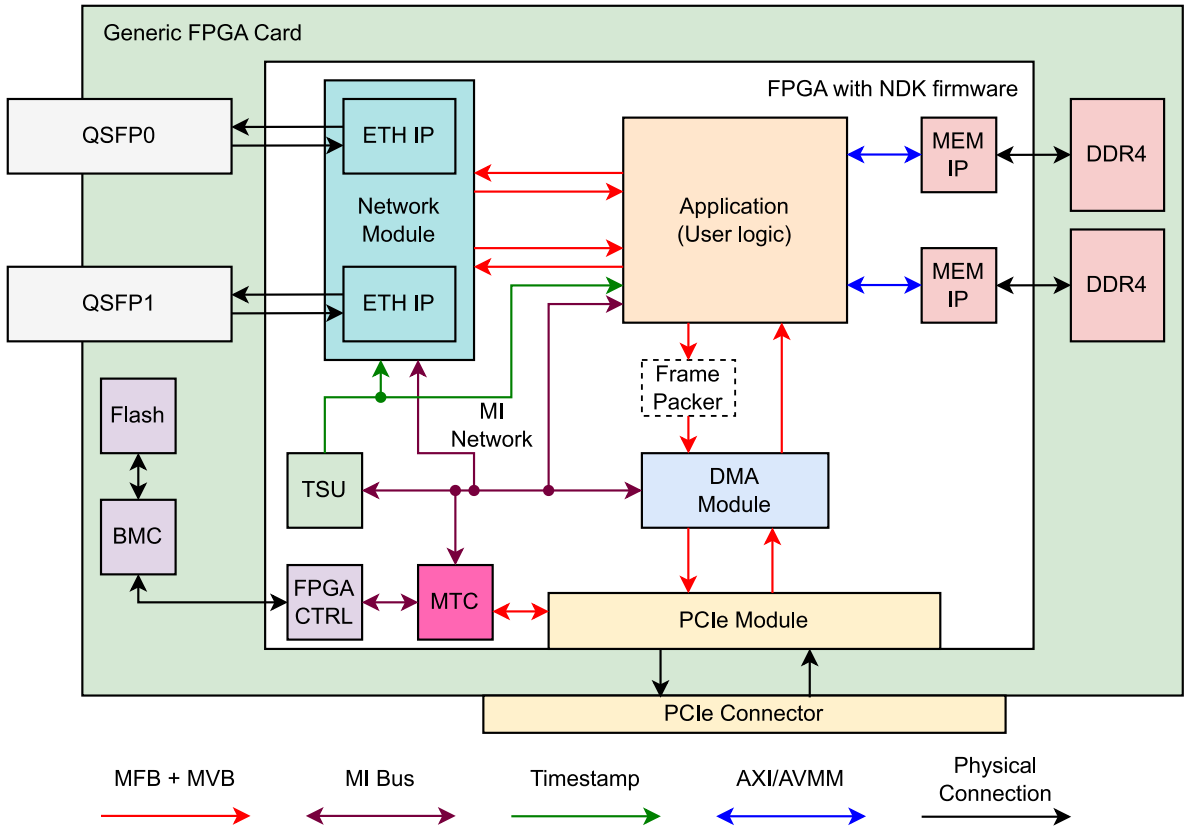


Fig. 1.1 The Simplified Architecture of NDK [5]

This digital circuit has two main components for communicating with the outside world. The first is the Network Module, which uses IP cores (designed by CESNET or by card vendor) to implement the Ethernet sublayers. This module is responsible for transferring frames between the Application core (The User logic) and the network, and for converting the IP interface to the bus protocol used by the NDK platform. [6] The second module – PCIe module – handles the communication protocol on the PCIe bus and transferring data and instructions between the PC and the FPGA card. In supported FPGAs, this is implemented on Hard IP blocks that can be connected directly to the PCIe bus. [7]

FPGA cards often have DDR memories. For this reason, the NDK provides a memory controller so that users can use standard buses (AXI/AVMM) to control external memory. [8] The controller itself is directly connected to the DDR memory. Another large block in the NDK design is the DMA module, which can communicate directly with the PC memory via PCI-Express when it is used. The open-source version of the DMA in the NDK design is called DMA Calypte. [9]

The last two modules in the diagram are the Time-Stamp unit (TSU) and the FPGA controller. The TSU generates accurate timestamps for the Network module and Application, core which can be used for performance or traffic analysis [10], and FPGA controller can reboot the FPGA or boot a new one from the QSPI flash memory. [5]

The NDK design has three types of buses – the Memory Interface bus (MI), the Multi-Value Bus (MVB), and the Multi-Frame Bus (MFB). The MI bus allows the software to access the internal register of the design. [11] This is done by MTC unit (MI Transaction Controller), which transfers PCIe transactions and pushes them in form of MI requests to the sort of MI network. The MVB bus is often used along with the MFB to transfer fixed-length metadata [12], and the MFB stream bus is a high-speed data bus that is able efficiently transfer large amounts of data between components (point-to-point transfer). [13] More detailed information about this type of bus and the others will be described in the following sections as it is essential to the final outcome of this work. The outcome itself is indicated in the figure above (**Figure 1.1**) as the *Frame Packer*, which should put small packets together to create so called *Super-Packet*. This Super-Packet could, in theory, increase the efficiency of the data transfer by lowering the overhead on the PCIe bus and effectively use *Descriptors* in DMA transfer. The topic of the efficiency is discussed in the following chapter.

1.1.1 MI Bus

The Memory Interface (MI) is a master/slave bus used by software to read and write data for configuring, control state, or statistics reading of components in the NDK platform. For this purpose, each component has its own address and is connected to a sort of MI network which is controlled by MI Transaction Controller (MTC). [14] Specification described in this chapter is based on MI Specification in NDK documentation. [11] The following table defines the MI bus signals from the perspective of the main component (**Table 1.1**).

Table 1.1 A list of signals that define MI Bus – Slave perspective

Signal Name	Signal Direction	Signal Width	Signal Description
ADDR	in	ADDR_WIDTH	Address
DWR	in	DATA_WIDTH	Data Write
BE	in	DATA_WIDTH/8	Byte Enable
WR	in	1	Write flag
RD	in	1	Read flag
ARDY	out	1	Address Ready
DRD	out	DATA_WIDTH	Data Read
DRDY	out	1	Data Ready

The MI bus supports two types of requests – write and read. A write request is initiated by setting the WR flag high. To route signal to the correct component, an address signal is used. Data intended for writing to the slave component is carried by a signal called DWR. This data is only valid when the WR flag is set. The data, along with BE and ADDR, must remain valid until the slave component acknowledges the request by setting the ARDY signal high. The simplified timing diagram of the write operation is shown below (**Figure 1.2**). During the first write request, data D0 is written to the component with address A0. There is a delay due to the late set of the ARDY signal by the main component. The second part shows writing two data items in two clock cycles – without delay.

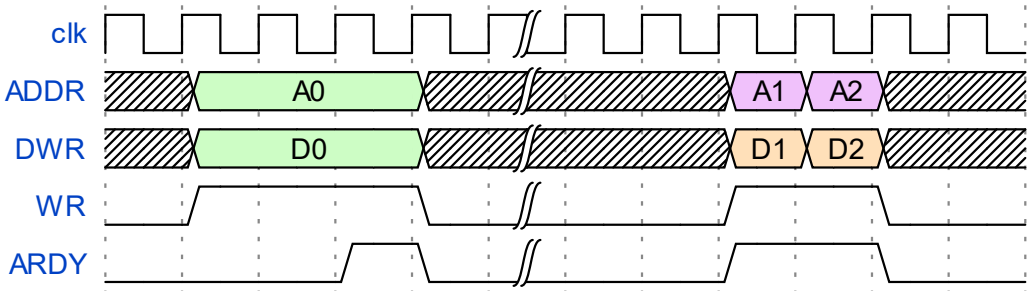


Fig. 1.2 Timing diagram of write process on MI Bus

Another type of request is a read request, indicated by the RD flag. Similar to the write process, this request is accepted by setting ARDY high. However, accepting the request does not imply confirming the sent data. Data confirmation is indicated by the DRDY flag, which can be issued simultaneously with the ARDY signal or later. Setting ARDY high indicates that the slave component is ready to accept another request and setting DRDY signal a valid response to the read request. The following timing diagram illustrates the read request process. The first part of the diagram shows read request to address A0. The response is immediate and confirmed by sending data item D0 and setting the ARDY and DRDY signals high. In the second read request, there is a request to address A1, to which is responded by setting ARDY high without sending any data. Master component sends another request to address A2

and the main component responds to these requests in the order they were received by setting DRDY signal high for each DRD item.

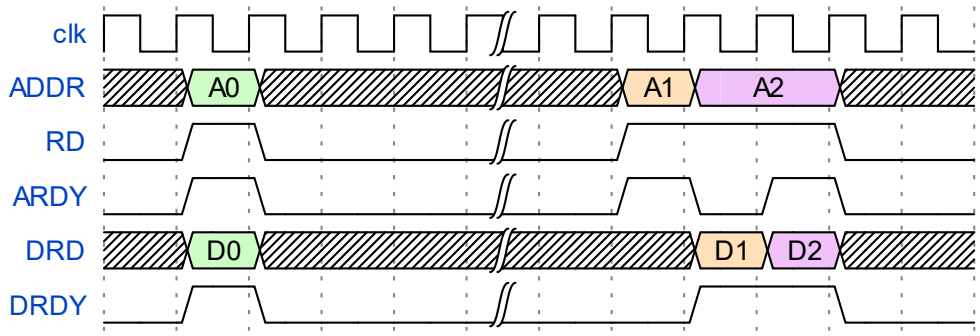


Fig. 1.3 Timing diagram of read process on MI Bus

Both, write and read request can be combined in a single communication. The typical example of this operation is called read-modify-write, which is used to control multi-port memory. [15] Additionally, byte enable can be used to transmit data that is smaller than the data bus width. The full specification of MI Bus is available in NDK documentation. [11]

1.1.2 MVB Bus

The Multi-Value Bus (MVB) was created by CESNET to enable parallel processing of data in the NDK architecture. It can transfer multiple data in a single clock cycle and is often used as a separate channel to transfer fixed length metadata for the data transmitted on the MFB bus. The following table shows the signals that the MVB is made of (**Table 1.2**).

Table 1.2 A list of signals that define MVB Bus

Signal Name	Signal Width	Signal Description
DATA	WORD_WIDTH	Transmitted data
VLD	ITEMS	Item Valid
SRC_RDY	1	Source Ready
DST_RDY	1	Destination Ready

Data on the MVB are called words, which consist of a specified number of items (ITEMS). The width of each item is defined by the generic parameter ITEM_WIDTH when bus is created. The data width is calculated by multiplying these two parameters. The valid signal (VLD) indicates which items in the data word can be processed. Since not all items must have valid data, gaps between items may occur. The following figure (**Figure 1.4**) shows the structure of the described bus.

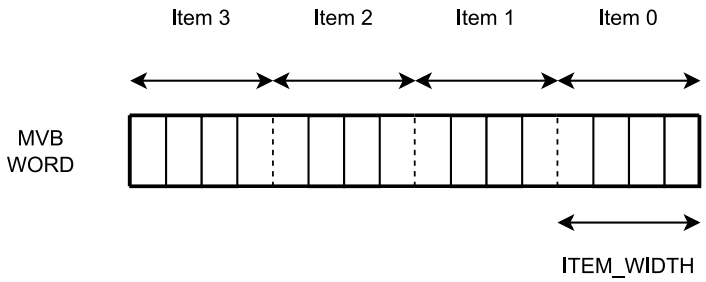


Fig. 1.4 Structure of the MVB Bus

The MVB introduces the Source Ready (SRC_RDY) and Destination Ready (DST_RDY) signals, which implement a request-acknowledgment mechanism. Transmitter validates its data and creates write request by setting SRC_RDY high. This request is confirmed by the receiver when the DST_RDY is set high as well. Both signals must be high for a transfer to complete. The port direction of each signal depends on whether the component is receiving or transmitting data on the MVB. For a receiving component, SRC_RDY is an input and DST_RDY is an output. For a transmitting component, the port direction is reversed.

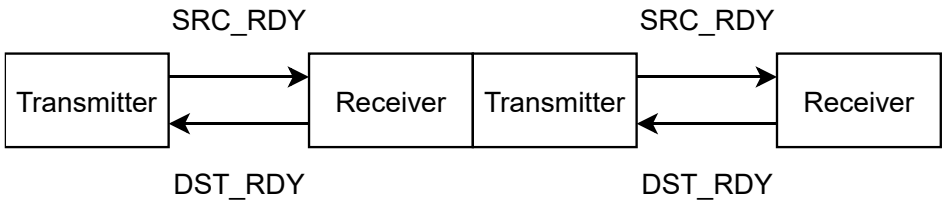


Fig. 1.5 Port direction in SRC/DST_RDY logic

Data transfer begins when SRC_RDY is set high. To indicate readiness to receive data, the receiver sets DST_RDY to high. The VLD signal, which has a width equal to the number of items in the data word, indicates which data items are meant to be processed. Each bit in the VLD signal validates the corresponding data item. The possible communication on the MVB bus is shown below (**Figure 1.6**). Each *X* in sent data corresponds to the VLD signal and indicates that the data in the item is not valid. The detailed specification of the MVB is available in the NDK Documentation. [12]

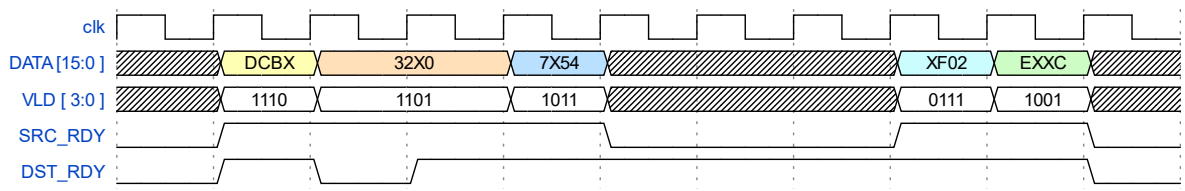


Fig. 1.6 Timing diagram of communication on MVB Bus

1.1.3 MFB Bus

The Multi-Frame Bus (MFB) shares the same principle of Multi-Buses as the MVB. Its purpose is to increase efficiency by utilizing unused space in data bus. For instance, the AXI-Stream protocol by ARM [17] requires data to start at the beginning of the data word, which can lower bus utilization when frames are not the same size as the word. The same applies to the Avalon-Streaming protocol by Intel. [18] For example, a 512b wide data bus can transfer the shortest possible Ethernet frame in one clock cycle, resulting in high efficiency when transferring 64B frames. However, Ethernet frames can reach up to 1518B [19], leading to frame sizes that don't align perfectly with the word. A 96B frame would leave half of the word empty since the new frame must start at the beginning. The Multi-Bus principle, on the other hand, allows frames to start at a different location within the data word, which is allowing multiple frames (or just parts of frames) to share the same word. [16] This chapter describes how is this method implemented and what rules must be followed to increase efficiency of the data transfer and its processing across the FPGA. **Table 1.3** lists all the signals that define the MFB bus.

Table 1.3 A list of signals that define MFB Bus

Signal Name	Signal Width	Signal Description
DATA	WORD_WIDTH	Transmitted data
SOF	REGIONS	Start of Frame – Region mask
EOF	REGIONS	End of Frame – Region mask
SOF_POS	SOF_POS_WIDTH	Position of the beginning(s) of a frame(s)
EOF_POS	EOF_POS_WIDTH	Position of the ending(s) of a frame(s)
SRC_RDY	1	Source Ready
DST_RDY	1	Destination Ready

Data are transmitted in the MFB words, represented by the signal DATA. The width of this signal is determined by the bus configuration parameters REGIONS, REGION_SIZE, BLOCK_SIZE, and ITEM_WIDTH. The structure of the bus is as follows: The parameter REGIONS defines the number of regions in the word. Each region is divided into blocks, and the number of blocks per region is defined by the parameter REGION_SIZE. These blocks, in turn, are made up of items, and the number of items per block is specified by the parameter BLOCK_SIZE. Finally, the parameter ITEM_WIDTH defines the size of the smallest recognizable unit on MFB, which is the item. The size of the MFB word can be calculated by multiplying these parameters together.

The REGIONS parameter defines the number of sections into which the word is divided. The rule is that the region must contain up to one start of frame (SOF) and one end of frame (EOF), so it can fit the whole frame or the end of one frame and the beginning of another frame.

The maximum number of frames (or parts of frames) that can be transmitted in a single clock cycle is equal to the number *REGIONS + 1* in a word. The second parameter, REGION_SIZE, determines the number of blocks in each region. Data transmission must follow the rule that the beginning of the frame must be aligned with the block. The third parameter, BLOCK_SIZE, sets the number of items that are in each block. An item is the smallest unit that can be recognized on the MFB bus, and its size is defined by the parameter ITEM_WIDTH. Items are locations where the end of frames must be aligned.

The configuration of the MFB is labelled as MFB#(REGIONS, REGION_SIZE, BLOCK_SIZE, ITEM_WIDTH). Although these parameters can be set randomly, there are only a few configurations that are supported in NDK. For example, the 100G Ethernet protocol is most effective (resources vs. frequency) when the bus configuration is MFB#(1,8,8,8). NDK design for 400 Gbps must be able to handle configuration MFB#(4,8,8,8), which can transfer 4 smallest Ethernet frames at the same time. The communication with PCIe is configured as MFB#(2,1,8,32) due to DWORD alignment on the PCIe bus. The following figure illustrates described configuration (**Figure 1.7**).

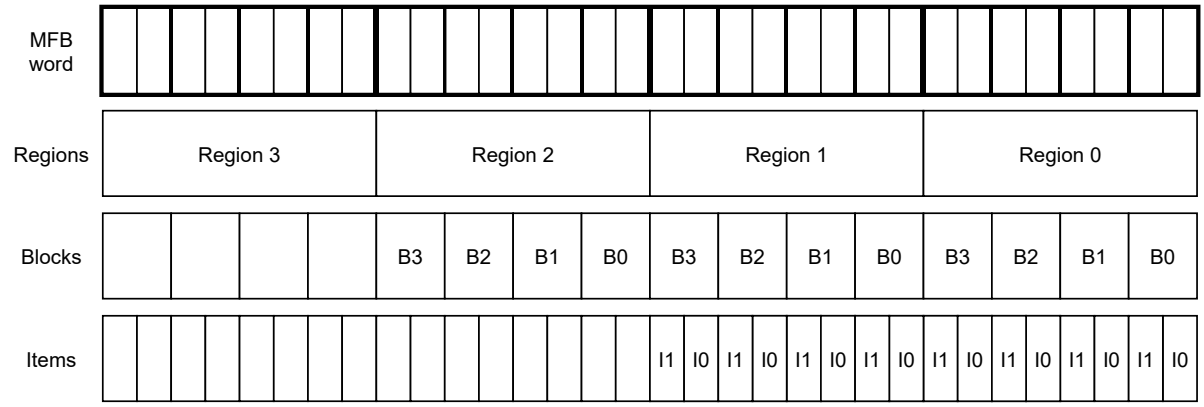


Fig. 1.7 Configuration of MFB Bus

As mentioned before, the MFB is able to transfer multiple frames in a single clock cycle or share unused space with other frames. To do this, it needs to know the position of each frame within the data word. This is where the SOF, EOF, SOF_POS, and EOF_POS signals come in. The SOF signal indicates the region where the start of the frame is located. The EOF signal works in the same way. The sizes of these signals are based on the REGIONS parameter, and its data serves as a mask for the start or end of the frame. The following diagram illustrates the SOF and EOF mechanism. A SOF with a value of "1111" indicates that the beginning of the packet is in every region and an EOF of "1101" indicates that the packet ends are in every region except the second. From this variation, it is possible to roughly determine where each packet is located. That is, there is an entire packet in the first region, there is a packet that starts in the second region and ends in the third region, there is a packet that starts in the third region and ends in the fourth region, and finally there is a fourth packet that starts in the fourth region and ends in one of the following words.

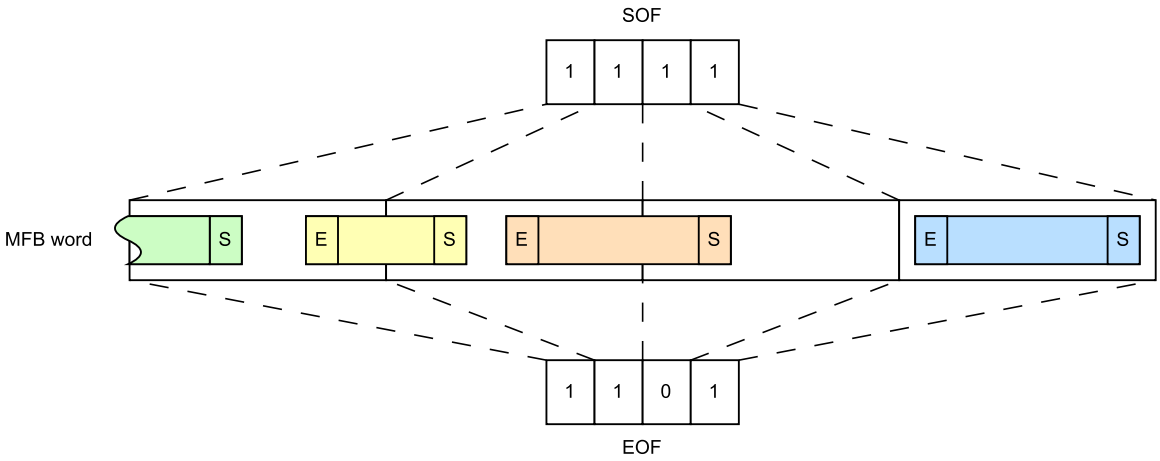


Fig. 1.8 MFB configuration – SOF & EOF example

The figure above (**Figure 1.8**) has one remaining issue – the exact location of frames cannot be pinpointed within the MFB word. The SOF and EOF masks only provide information about whether a frame starts or ends within a region. To determine the precise location, two additional signals are needed, SOF_POS and EOF_POS. The width of SOF_POS is determined by the number of blocks and its value indicates the location of the frame's start within the region, which is aligned to the blocks. The width of EOF_POS must have a higher resolution, as the end of frames can be aligned to the items. Its width is determined by the number of blocks and number of items within the MFB word. [13]

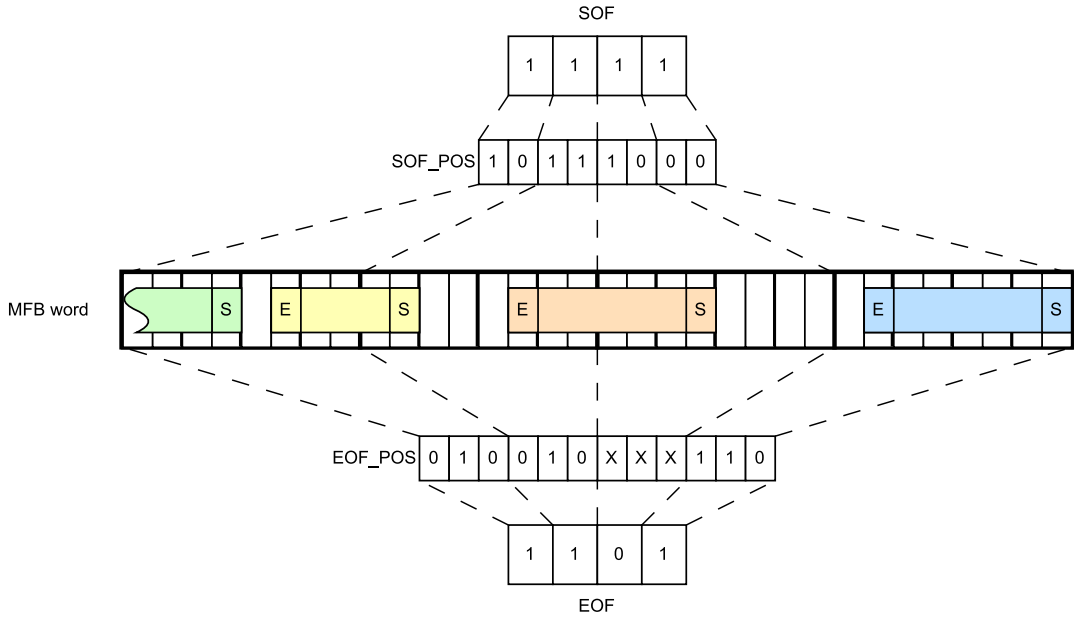


Fig. 1.9 MFB configuration – SOF_POS & EOF_POS example

The last two signals in the table (**Table 1.3**) are SRC_RDY and DST_RDY. These signals were described in the MVB bus, and they function in the same way. The following timing diagram (**Figure 1.10**) shows how data is transferred on the MFB bus from the transmitter to the receiver – SRC_RDY is the input of the component and DST_RDY output. The transmitted frames, labeled from A to F, comply with the figure presented below (**Figure 1.11**). The MFB configuration used in this example is MFB#(4,4,2,8).

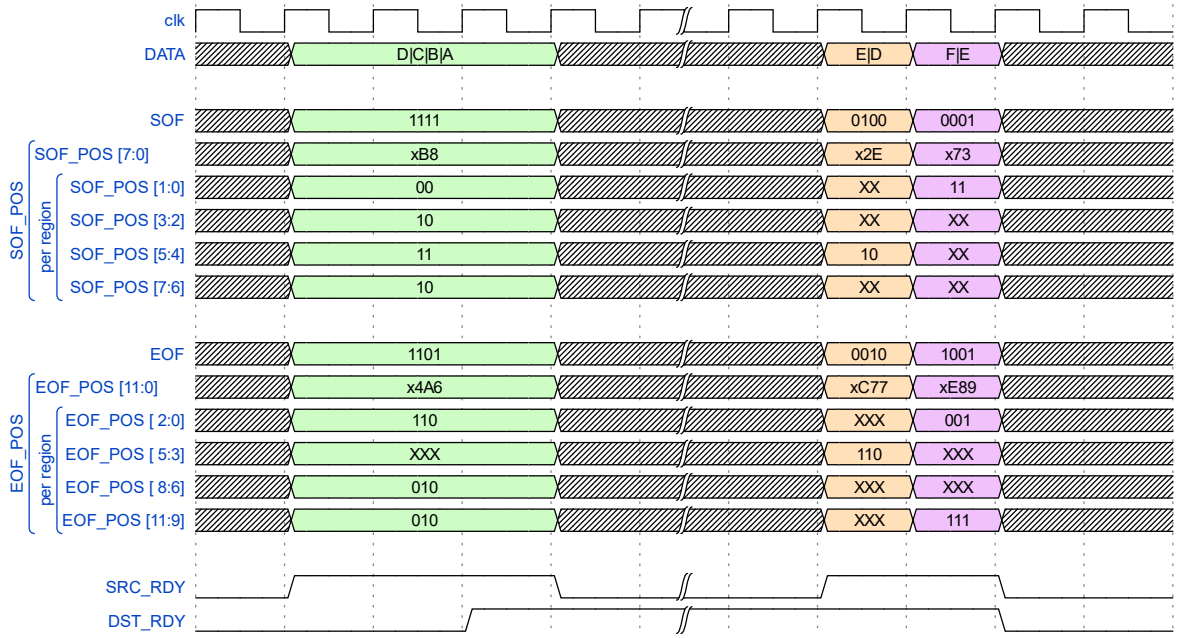


Fig. 1.10 Timing diagram of data transmission on MFB Bus

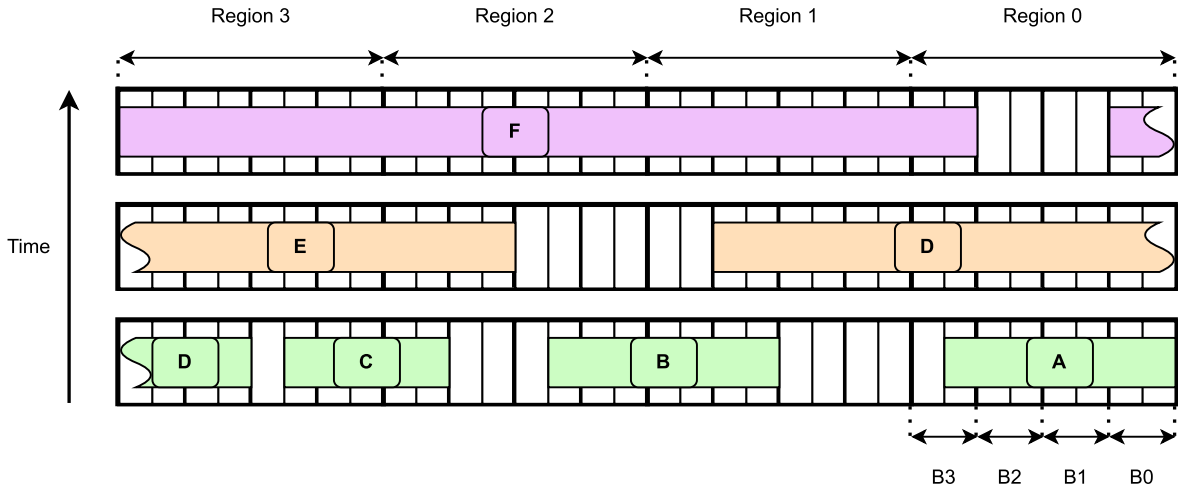


Fig. 1.11 Transmission of frames on MFB Bus

2 DATA TRANSMISSION

The main goal of this work is to efficiently move data between an FPGA and software running on a host PC. To achieve this, the NDK can directly transfer packets from Ethernet (or processed data) to the computer's memory using a DMA controller and suitable software. The interface that allows FPGA to access host memory is called PCIe. The DMA transfer process and its effectiveness on PCIe will be discussed in detail in this chapter.

2.1 PCI-express

PCIe (Peripheral Component Interconnect Express) is a high-speed serial interconnection that is primarily used to connect components such as the CPU, DDR memory, graphic card, or acceleration cards with FPGAs. It is the successor of the PCI (Peripheral Component Interconnect) standard, which was first introduced by Intel in 1992. The original PCI was a 32-bit bus that could transfer data at a maximum speed of 1056 Mbps. [22] In contrast, the current PCIe 4.0 standard can transfer data at speeds up to 32 GBps, and the newer PCIe 5.0 even up to 64 GBps in x16 lane configuration. [23] This chapter will describe its architecture and transmission efficiency of the actual data.

2.1.1 PCIe Architecture

PCIe uses a bus-based architecture, meaning that the peripherals are connected to a central unit. In this case the Root Complex. These peripherals (or endpoints), communicate with the Root Complex using lanes, where lane is a pair of serial links. For example, *PCIe 4.0 x16* has 16 lanes, where one wire in each pair is for transmitting data, and the other is for receiving. The Root Complex itself manages communication for all connected endpoints and sets up direct memory access (DMA) for them. [21] The topology of the PCIe is shown in the **Figure 2.1**.

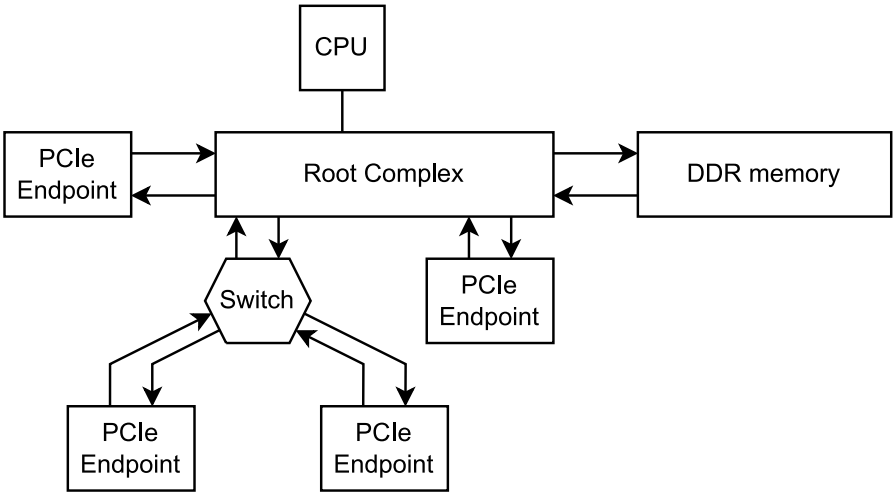


Fig. 2.1 PCIe Bus topology [21]

The PCIe is implemented in the first three layers of the ISO/OSI model. The Physical Layer (PL), the Data Link Layer (DLL), and the Transaction Layer (TL). When data is sent from one device to another, it must pass through each of these layers. In Transaction Layer, the data are divided into units called Transaction Layer Packets (TLPs). Every TLP has its own *Address* header, which defines the destination device, *Payload* space, which is used to carry data, and an optional *Cyclic Redundancy Check* (CRC), to detect errors in the data transmission. The CRC is handled by the device logic and is typically labelled as ECRC. The control of the checksum is done by the device logic at the receiver. The header can be from 12 to 16 bytes long, and the CRC 4 bytes long. [24]

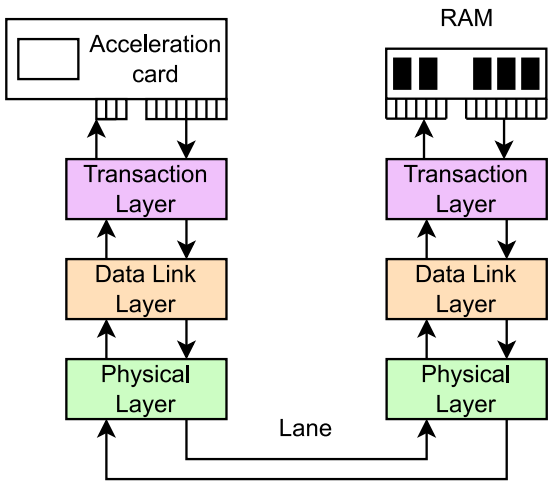


Fig. 2.2 PCIe layers between two devices [24]

The Data Link Layer ensures reliable transmission of the TLP by detecting errors, managing the link connection, and controlling the order of data transmission. To detect errors, the DLL adds a CRC to the TLP packet labelled as LCRC. This CRC is 1 byte long. The DLL also ensures that data is sent in the correct order by adding a *Sequence Number* to each TLP packet creating Data Link Layer Packet (DLLP). Sequence Number is 2 bytes long and tells the receiver which TLP it should expect to receive next. To ensure that TLP packets are received correctly, the DLL uses a *Replay Buffer*, which stores sent packets until a confirmation message is received. The confirmation message is sent in a separate DLLPs, which are typically 6 bytes long and includes an ACK or NACK flag to indicate whether the corresponding TLP was received successfully or not. When the transmitter receives an ACK, it deletes the corresponding TLP from the Replay Buffer. On the other hand, when NACK is received, it resends the TLP. The transmitter will continue to resend TLPs until it receives an ACK, or the Replay Buffer fills up. In the second case the communication will break. [20][24][25][26] Described principle of DLL is shown in **Figure 2.3**.

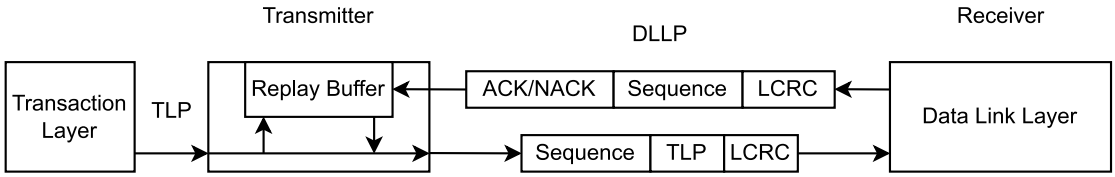


Fig. 2.3 Communication on Data Link Layer

The last part of the PCIe ISO/OSI model is the Physical Layer. Its purpose is to transform data from the DLL to the bitstream which can be sent over the wires. Once the DLL pushes its data to PL it must pass through several steps. Firstly, the data is *Scrambled*, which means it is evenly distributed across the available lanes. Then, the data is encoded to ensure DC balance on the bus and enable clock recovery at the receiver. The typical encoding is 128b/130b, meaning that 128 bits of data are encoded into a 130-bit signal. The older PCIe generation used 8b/10b encoding, which caused high loss in throughput. The encoded data is then processed by the SERDES (Serializer-Deserializer) converter, which transforms the bytes on each lane into a bitstream. To differentiate between DLP packets, START and END symbols are added to the original packet. Each of these symbols is typically on byte long. [27] The final packet that is send over the physical wires is shown in the figure below (**Figure 2.4**).

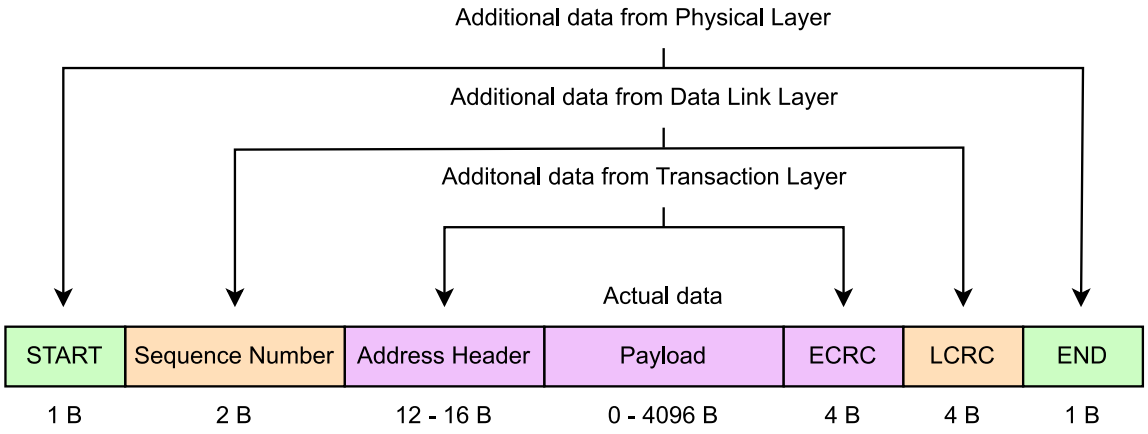


Fig. 2.4 Packet sent over the PCIe [24]

2.1.2 Efficiency of Data Transfer Over PCIe

There are two main factors that can affect the efficiency of data transfer over the PCIe bus. Data transfer overhead and system configuration. Data transfer overhead is mainly caused by encoding, bus traffic and TLP packet overhead. In system configuration is the most crucial parameter Maximum Payload Size (MPS). This parameter determines the maximum amount of data that can be transferred within a single TLP. Since each component on the bus has its own MPS parameter, the lowest value is considered globally in the PCIe system. This chapter will be based on the source of AMD/Xilinx [20].

The first discussed factor in the transfer overhead is *Symbol Encoding*. It is a method of converting data signals in PL into more bits to improve reliability and ensure DC balance on the bus. While this is essential for proper data transmission, it also reduces the throughput of the transfer since specification gives the maximum speed for encoded signals. The common encoding used in newer generations of PCIe is 128b/130b, which means that for every 16 bytes of data, there are additional 2 bits that are considered a part of the transmission. This results in an average throughput loss of approximately 2%. In contrast, older generations of PCIe used 8b/10b encoding, which caused throughput loss of 20%. [20]

The second factor in the transfer overhead that is affecting overall performance of the bus is the *Traffic Overhead*. This type of overhead is caused by packets whose task is to manage the link and ensure reliable data transfer. One example of traffic overhead is the *Credit-based Flow Control* system, which ensures that the data is not discarded due to a small buffer in the receiver. The initial credit number is obtained from the initialization process. When the transmitter sends data to the receiver, it consumes one credit per sent packet. Once the credit number reaches zero, the transmitter waits until the credit number is updated. This is usually done once the buffer on the receiver site is empty. [26]

As a part of a transfer overhead may be considered an ACK/NACK mechanism described in the previous section. This confirmation, even though needed for transfer reliability, is consuming bandwidth of the link. The overhead cause by this mechanism can be reduced by collapsing messages into one, meaning the message with the sequence number 3 deletes TLP packets with indexes from 0 to 3. But by doing this a throttle may occur in the form of a full Replay Buffer. [20]

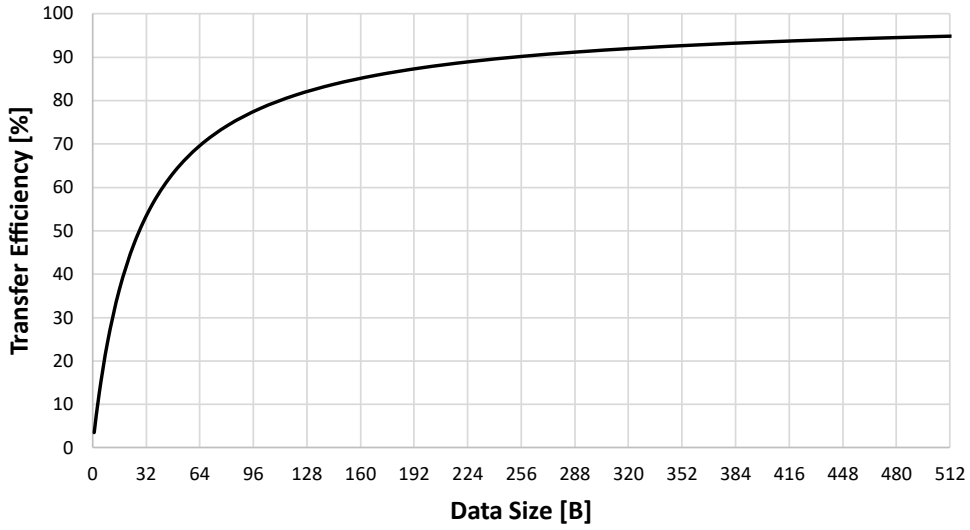


Fig. 2.5 Transfer inefficiency caused by Packet overhead – worst case

Another factor that can reduce efficiency of the system throughput are the packets in which the data is transmitted. Such a packet is shown in **Figure 2.4**. In addition to the actual data, there is also overhead added by PCIe layers. In the worst case this overhead can be up to 28 B.

This inefficiency is significant when transferring large amounts of small data to different addresses. The figure above (**Figure 2.5**) shows how the *Payload* can change the efficiency of the transfer. The MPS is considered as the maximum possible value – 4096 B, meaning each data packet is sent in a its own TLP. The following equation describes the efficiency dependence on payload of each TLP packet.

$$Transfer\ Efficiency = \frac{Data\ Size}{Data\ Size + TLP\ Overhead} \cdot 100\ [\%] \tag{2.1}$$

The last factor affecting PCIe performance is the system parameter *Maximum Payload Size* (MPS). In the last example this parameter was set to its maximum value – 4096B. However, this value is not widely used for several reasons. The first reason is that every component in the data path must have the same ability to handle such payload. This may be possible, but it is problematic in terms of component prices. The second reason is efficiency. It is possible to achieve sufficient efficiency by setting the MPS to lower values. For example, transferring a 512B data packet with the MPS set to 128B would require 4 TLPs. This means that the overhead in the worst case would be 112B. The efficiency of this transfer would be 82%. Setting the MPS parameter to 256B would increase the efficiency to 90% and by setting MPS to 512B, the efficiency would be 94 %. The point is that increasing the MPS parameter gives better efficiency, but the increase is disproportionate to the effort – the price of better hardware and complexity of the software. This is why most systems have MPS set to 128B or 256B. [20] The following equation describes transfer efficiency on MPS parameter.

$$Transfer\ Efficiency(MPS) = \frac{MPS}{MPS + Overhead} \cdot 100\ [\%] \tag{2.2}$$

To summarize this chapter. Transferring data between two endpoints requires a certain amount of overhead. While some factors are out of the user's control – such as symbol encoding or packet overhead – understanding them can help optimize the system for maximum efficiency. The main factor that can be influenced is to fully utilize the TLP packet at the PCIe level, so the overhead is minimal.

2.2 DMA

Direct Memory Access (DMA) is a way for devices to communicate directly with the computer's memory (Random Access Memory – RAM) without using the Central Processing Unit (CPU). By doing this, it is possible to efficiently transfer large amounts of data without consuming the CPU's time, so it can focus on other tasks such as data processing. [28] DMA is usually made possible by a bus that connects the CPU, RAM, and external device. The most common type of bus used for DMA is PCIe.

At the heart of DMA transfers is a device called the *DMA Controller*. [28] This device temporarily takes control of the PCIe bus instead of the CPU to transfer large amounts of data. By transferring data this way, the transfer can be more efficient because the DMA Controller can be optimized for the task. However, the controller cannot work on its own, as it would cause conflicts in the host memory. For this reason, the function of the controller is handled by using *Descriptors* created by the driver running on the CPU. [29] The descriptor is an instruction for the DMA Controller. Each of these instructions gives access to the part of the memory where the controller can read or write data. The way the DMA is built or descriptors are passed can vary, so the following section will only focus on DMA transfer in the NDK Framework.

DMA Controllers can be designed in two ways: *Streaming DMA* and *Packet DMA*. Streaming DMA is optimized for transferring continuous streams of data, such as audio or video, because it can send multiple packets together. This allows the DMA Controller to increase the throughput of the system by reducing the overhead on the PCIe. [30] However, it can make it difficult for software to process other data because the received data is stored in a continuous memory space. Packet DMA, on the other hand, distinguishes between packets and sends them separately to the memory space allocated by the descriptor, making it easier for different CPU cores or applications to access the data. This approach, however, has a higher transfer overhead than streaming. [31]

2.2.1 DMA Transfers in NDK

The NDK uses a vendor independent DMA Controller to communicate with the host memory. The first DMA Controller developed is called *DMA Medusa*, [31][9] which can transfer data at up to 400 Gbps and supports up to 512 DMA channels. The second controller, currently available in configuration MFB#(1,4,8,8), is called *DMA Calypte* and is designed to achieve the lowest possible transfer latency. Both controllers are implemented entirely on the FPGA [32] and are connected directly to the PCIe Hard IP, which in turn is connected to the PCIe bus. For the 400 Gbps transfer rate, the card with the FPGA is connected to the host PC via two PCIe 4.0 x16 slots or single PCIe 5.0 x16 slot.

The operation of this DMA Controller is packet-based, which means it uses its given memory space for a single packet. This makes it easier to build software on top of it but may result in higher transfer overhead, which becomes amplified when transferring large amounts of small packets. There are two types of overhead, on the PCIe bus in the form of TLPs whose payload is not fully utilized (see prev. section), as well as overhead caused by the management of the transfer. The management overhead is caused by the increased use of descriptors that need to be periodically updated so that the DMA Controller can send a continuous stream of data. This update process consumes some of the PCIe bandwidth. The following example describes the process of transferring data from the peripheral device to the host RAM by using the RX module of the DMA Controller. A schematic representation of this example is shown in the figure below (**Figure 2.6**). It should be noted that the RX module is used to *receive* data from the Application core in the FPGA and to forward it to the host memory via PCIe. [33]

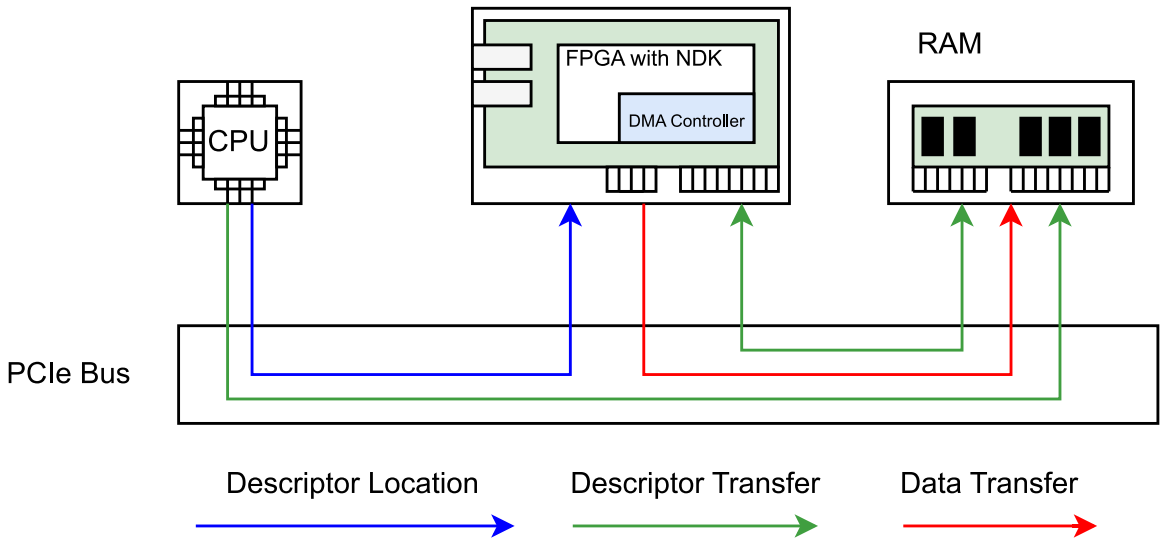


Fig. 2.6 The DMA architecture over the PCIe bus [35]

Before data can be sent, the DMA controller must be set up. This is done by the DMA driver running on the host CPU by creating a set of descriptors and sending them to its memory. The descriptors are stored in a specific address space known to the DMA Controller and are automatically updated as they are used. The final part of the setup is for the DMA Controller to load the descriptors from the host memory. When the setup is complete and the controller has data to send to the host memory, it takes control of the bus and starts transferring packets. As an example, these packets are the Ethernet frames sent when using the ndk-app-minimal reference model. Each frame is first converted to the TLP and then sent over the bus to the host memory. If the frame is larger than the payload size (MPS) of the TLP, the frame is divided into smaller chunks. Once the frame is written into the host memory, the descriptor is marked as used and another is used for the next frame. Periodically, the status of the used descriptors is sent back to the host memory where the DMA driver is running on the CPU and the descriptors are replaced by new ones. The transfer overhead, based on this example, is higher when the small frames are sent over the bus because the descriptors need to be updated more often than in the case of larger frames. [33] [34] [35]

To improve the speed of data processing on the CPU or enable multiple applications to use FPGA card, the DMA implemented in the NDK architecture supports *channels*. This approach divides the traffic from the FPGA into multiple virtual lanes, called DMA Channels. From a software perspective, each channel behaves as an independent network interface, each of which can be accessed individually. Each channel operates independently within the DMA module, with each channel controlled by an independent driver running on the CPU. This control is provided by the MI bus and its task is to filter the data transfer to the host memory by setting the control register to either *start* or *stop*. When a channel is stopped, the DMA Controller drops all incoming frames associated with that channel. [31][32]

3 DESIGN OF THE ARCHITECTURE

The aim of this thesis is to design a digital circuit that can increase the transmission efficiency of small packets, thus improving the overall throughput of the system for this type of data. A previous chapter discussed the concept of transmission efficiency and proposed a solution to increase it by combining small packets into larger units. In the NDK platform, these larger units are called *Super-Packets*. By implementing this component, the DMA used in the NDK design can mimic the behavior of Streaming DMA while keeping the benefits of the packet-based DMA approach. This chapter describes the specific requirements for the digital circuit and proposes a solution for the component that could theoretically increase the transfer rate for small packets without causing a drop in performance when sending larger packets.

3.1 Design Requirements

This section provides a more detailed description of the task so that the component can be properly designed. As mentioned earlier, the design should merge several small packets received from the MFB interface and create a single larger super-packet. The packets within the Super-Packet should be aligned to MFB blocks, i.e. there should be minimal empty space. The size of the Super-Packet is determined by the available memory space specified by the descriptor. However, the Super-Packet doesn't have to completely fill the allocated space, so its length ranges from the minimum size of an Ethernet frame to the maximum available memory. This length can be set by a generic parameter but should not exceed 2048 B in total.

The component must also manage the DMA Channels, as DMA module would interpret the SP as a single packet with a single channel number so only packets with the same channel number can be accumulated into one SP. However, the MFB bus itself is a challenge. The MFB bus allows multiple packets to be transmitted within a single clock cycle, and these packets do not necessarily belong to the same channel. As mentioned earlier, the MFB configuration MFB#(4,8,8,8) allows data from up to five different packet to be transmitted. To overcome this problem, the incoming traffic must be sorted by the channel number associated with each packet in the MFB word. To ensure consistency, the output interface of the component must be the same as the input interface. By dividing incoming traffic into several separate channels, multiple output interfaces are created. As the component must be compatible with the existing system, a merging mechanism is essential as the DMA input interface is designed for the single MFB bus. In other words, the input and output interface is common for all DMA Channels and the merging mechanism is needed as processing multiple DMA Channels requires divide the traffic into equal number of processing units.

The alignment principle described so far is shown in the following figure (**Figure 3.1**). In this example, the MFB input with the configuration MFB#(2,4,2,8) contains packets from two channels. The proposed component should be able to sort the data according to the channel to which it belongs. The packets are then stored in output FIFOs, where they are

aligned to the blocks so the gap between packets is minimal. The width of each FIFO corresponds to the width of the MFB, and its depth depends on the size of the Super-Packet.

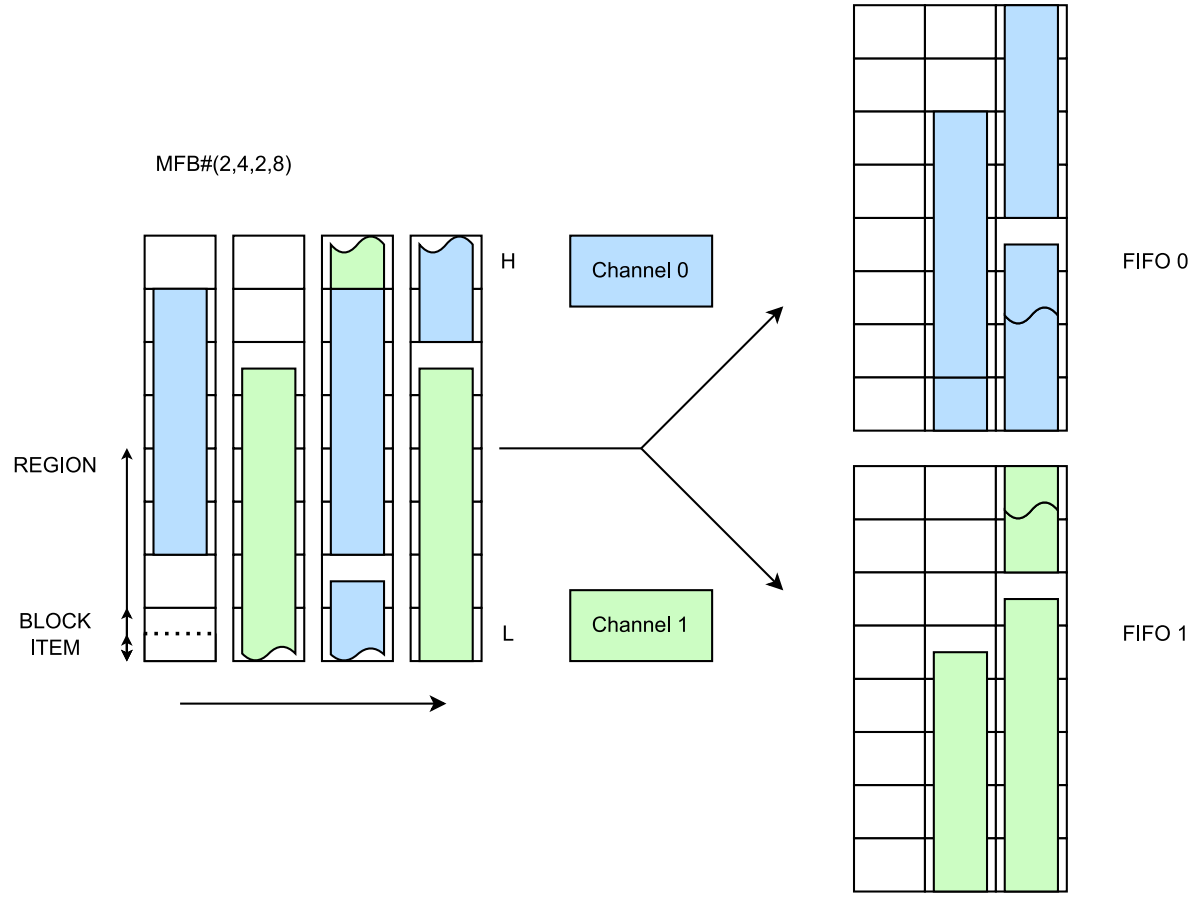


Fig. 3.1 The basic principle of the Frame Packer

To maximize efficiency, it is necessary to store parts of super-packet until they reach the required size. However, there are some potential challenges. Firstly, during periods of traffic when a channel is not fully utilized, it may take longer to assemble the Super-Packet of minimum length. To overcome this, a timeout mechanism must be implemented to prevent the packet from taking too long to reach its destination. It should be noted that even if timeout doesn't occur, the assembly itself will increase the latency of the system, which is insignificant in certain use cases, such as network monitoring. Secondly, packets within the Super-Packets must be coherent. This means that if the stored Super-Packet is partially filled with small packets, and an incoming packet exceeds its maximum size, the stored data must be sent before the new packets can be processed. Given the constraint that packets must be sent in the order they were received, sending larger packet before the stored data is not an option.

As mentioned earlier, the NDK design is scalable, and the proposed solution must be as well. To achieve a throughput of 100 Gbps, the NDK design is capable of handling the MFB#(1,8,8,8) configuration. However, the target throughput is meant to be 400 Gbps. This is typically achieved by using the MFB#(4,8,8,8) configuration at frequency of 200 MHz.

Such speed creates a constraint where the longest combinational path must not exceed a delay of 5 ns. It's worth noting that the NDK supports high-end FPGAs from both major chip manufactures, AMD/Xilinx and Intel, which means there are plenty of resources available for the given task. On the other hand, it also requires a generic design that cannot use vendor specific hard blocks, which can perform some tasks efficiently.

3.2 Top Level Concept

This section describes the top-level architecture of the proposed design based on the requirements outlined in the previous section. It consists of several parts and the interfaces to communicate with the existing system. The input interface of the proposed component consists of MFB and MVB bus with the configuration necessary for the required speed. The same configuration is used for the output interface since the compatibility with the existing NDK architecture must be preserved. The following **Figure 3.2** illustrates the internal structure of the Frame Packer.

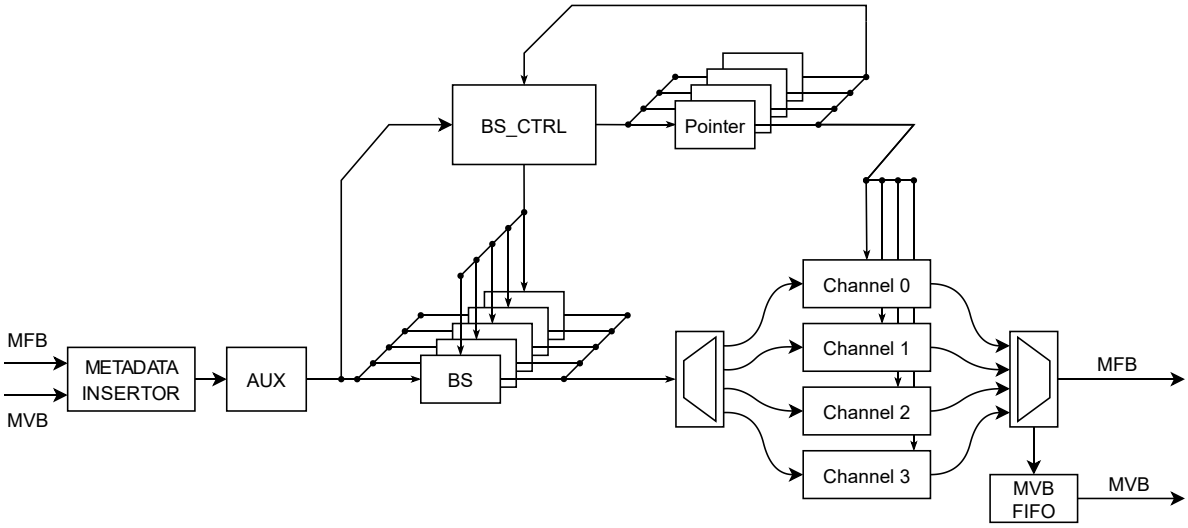


Fig. 3.2 Top Level view of the Frame Packer – 4 Regions, 4 Channels

As the MFB and MVB interfaces are not synchronized at the input of the component, a unit called the *Metadata Insertor* is used to synchronize them. Once the Metadata Insertor has both the data from MFB and the metadata from MVB, the stream is passed to the next stage where the auxiliary signals are generated. These signals help to process the data in each Channel unit and to correctly calculate the select signal for each Barrel Shifter. In the next stage, the data and auxiliary signals are processed by the *Barrel Shifters*. Their job is to shift data to reduce empty space between packets to the minimum. Shifting the data at the input of the component saves the resources compared to the case where there are Barrel Shifters in each Channel unit. Note that the number of channel units is set by the generic parameter. The number of Barrel Shifters in this case depends on the REGIONS parameter of the MFB bus. Since the MFB can carry up to REGIONS + 1 packets simultaneously, there must be the same number of Barrel Shifters so

that each can handle its own part of the packet independently. Once the data has been shifted, it is sent to the channel to which it belongs. This distribution is based on the metadata from the MVB bus and is performed using *Demux* component available in the OFM repository.

Each Channel has its own *Pointer* which is updated by the data from the *Barrel Shifter Controller*. This pointer is used to track the assembly status of each word within the channel and is another parameter that must be considered when calculating the shift select signal for the Barrel Shifters. Inside the *Channel* unit itself, the Super-Packet is assembled by appending parts of shifted packets. Once the Super-Packet is complete, the SRC_RDY signal is generated and the output component, called the *Merger*, considers the completed channel, and gives it enough space to send the Super-Packet to the output MFB interface. Finally, the MVB FIFO is used to store Super-Packet metadata and passes it to the MVB interface when requested. This metadata is generated in the Channel from which the Super-Packet originates.

3.3 Components

The previous section introduced the basic structure of the Frame Packer and its concept. This section gives a more detailed description of each part. Although the principle of concatenating packets into Super-Packets (per each channel) is quite simple, a closer look at the structure might convince you otherwise. The implementation of such a digital circuit is not so simple, especially in the case where multiple packets are transferred at the same time.

3.3.1 Metadata Insertor

This component is part of the OFM library [36]. Its purpose is to insert metadata from the MVB bus into the auxiliary META signal of the MFB bus and to align the metadata with the SOF (or EOF) of the frame. In this case, the MVB bus transmits the *Length* and *Channel ID* of each packet. The reason for including this component is that the MVB and MFB buses are independent buses. Therefore, they are not synchronized, which means that MVB data can arrive at different times but in the same order as packets on the MFB bus. The principle of the Metadata Insertor is shown in the following figure (**Figure 3.3**).

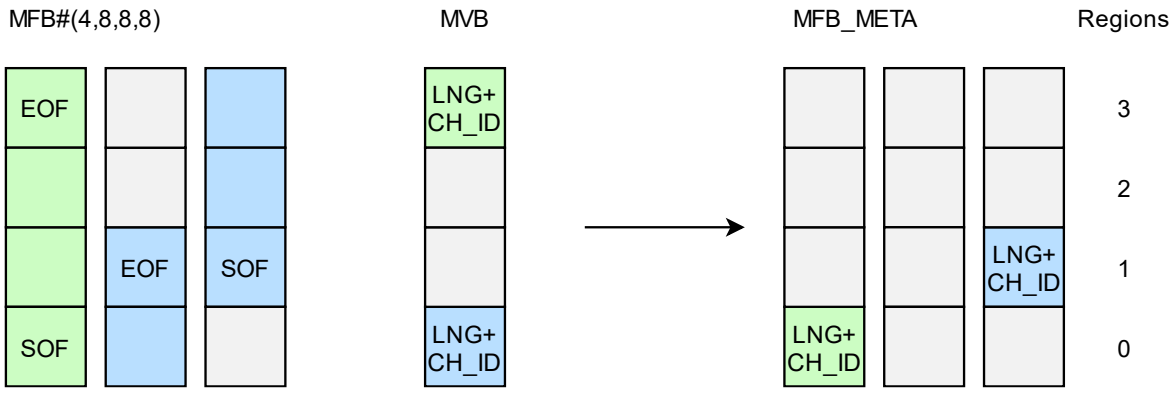


Fig. 3.3 Principle of Metadata Insertor

3.3.2 Auxiliary Generator

Once the MFB data and MVB metadata are synchronized, the stream is passed to the stage where auxiliary signals are generated. These signals help calculate the shift select signal for each Barrel Shifter and recreate additional MFB signals, such as SOF or EOF, as these signals will not be valid after the data shifts.

The first signal generated is *Block Valid*, which is a vector that validates each block of the packet in the MFB word. This vector is generated separately for each packet, resulting in an array of vectors. The length of this array is based on the REGIONS parameter of the MFB bus (i.e., the maximum number of packets in a word). This array of vectors is then used for the calculation of the shift select signal and to recognize valid blocks in each channel unit. The following figure illustrates the described principle of validating blocks (**Figure 3.4**).

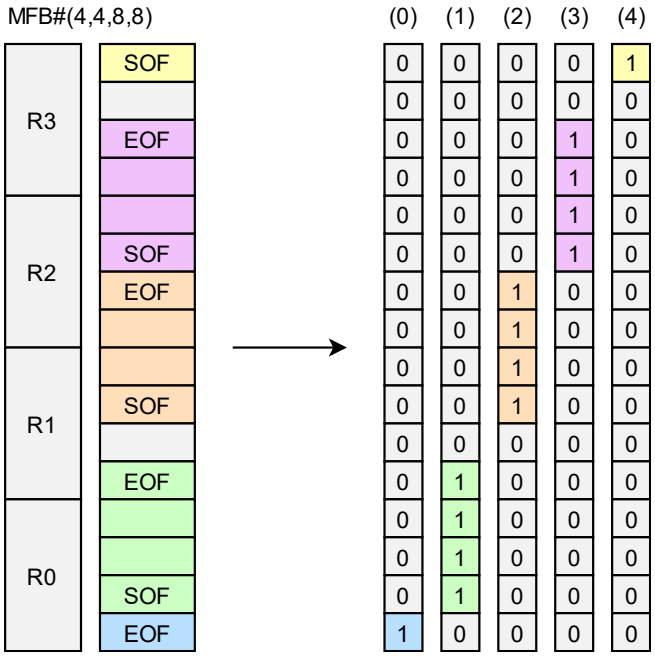


Fig. 3.4 Block Valid array

The next two auxiliary signals help to detect each packet's boundary – *SOF_ONE_HOT* (or SOF_OH) and *EOF_ONE_HOT* (or EOF_OH). These signals are in a similar format to the Block Valid array, and their purpose is to validate the block where the packets begin and end, respectively. Their purpose is to recreate the additional MFB signals after the MFB data has shifted. These signals, along with the Block Valid, are generated using the *Auxiliary Signals* component [37], which is part of the OFM library. For the purposes of the Frame Packer, this component was adjusted to generate additional SOF_OH and EOF_OH signal. The last auxiliary signal, extracted from the MFB bus, is the SOF_POS. Technically, this signal is not processed, it is just passed in the form of an array into the component responsible for the calculation of the shift select signal. As it is an offset of each packet within the MFB word, it is included in the calculation of the shift select signal.

The last two auxiliary signals are generated from the metadata – The *Packet Length* and *Channel ID*. It is necessary to take into consideration gaps between the regular packets within the Super-Packet in order to correctly calculate its final length. These gaps are caused by packets being aligned to the blocks and it ranges from 1 to 7 items when the parameter MFB_BLOCK_SIZE is set to 8. The easiest way to deal with this is to round up the incoming lengths that are in the number of items to the next nearest block as the maximum size of the empty space between the packets will not exceed the size of the block.

The last data to adjust is the *Channel ID*. Each Barrel Shifter must receive a valid Channel ID in order to calculate the shift select signal and then distribute the shifted data to the correct channel. For this purpose, a component called LAST_VALID is used. The internal structure is shown in **Figure 3.5**. Its purpose is to pass MVB data if they are valid and to store the last valid data if there is none on the input. This is necessary for the first Barrel Shifter, which is processing only packets originating in one of the previous words.

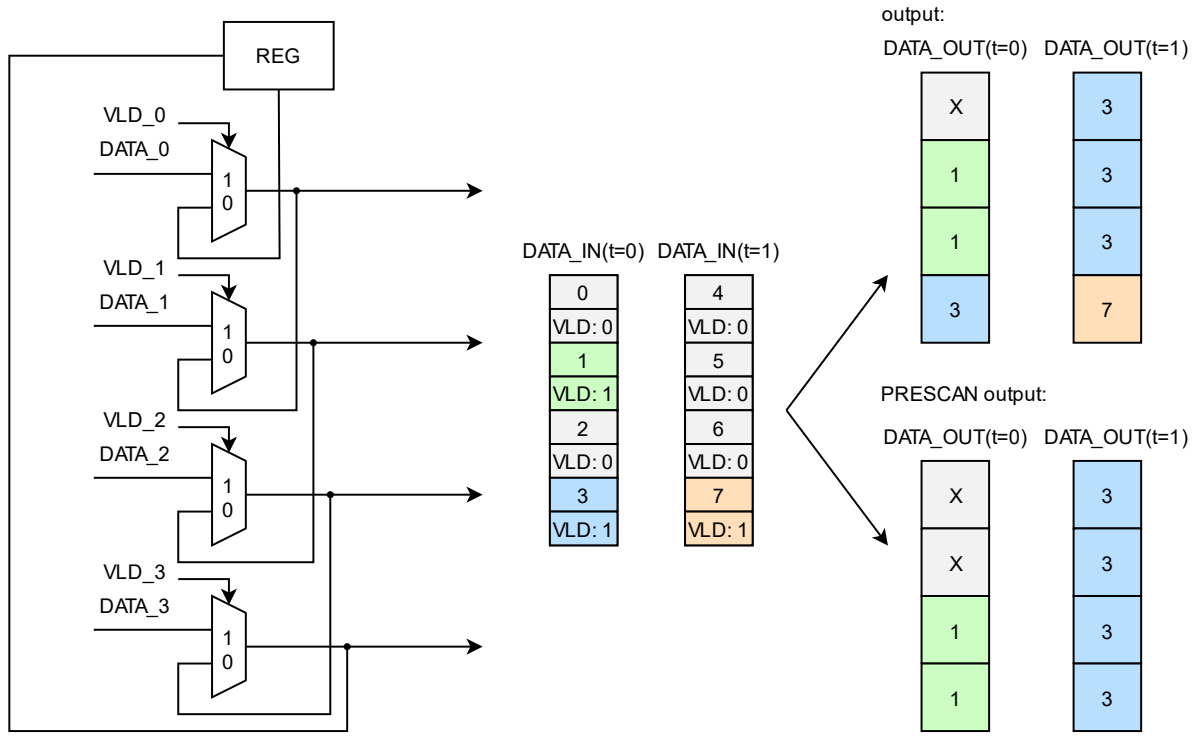


Fig. 3.5 LAST_VALID structure and expected output

In order to process data from multiple channels, it is necessary to handle each packet independently. To differentiate between packets, the MFB_DROPPER component was used to filter each one of them into a separate path. Its job is to mask the packets that started in the region marked as *drop*. To do this, it uses a packet tracking mechanism called *packet_continues*, which is a simple way of determining whether the packet ends in the MFB region (or MFB word) or continues into the next one. For the purposes of the Frame Packer, the inner signal that indicates packet continuation has been added to the output interface.

The following figure (**Figure 3.6**) shows the internal structure of the *Auxiliary Generator* with the described principle. The first DROPPER is set to drop all packets (except those from the previous word). The others are set so that the second DROPPER only passes packets from the with SOF in the first region, the third DROPPER only passes packets with SOF in the second region, and so on. This makes it possible to distinguish between packets and to process up the five packets in a clock cycle without pausing.

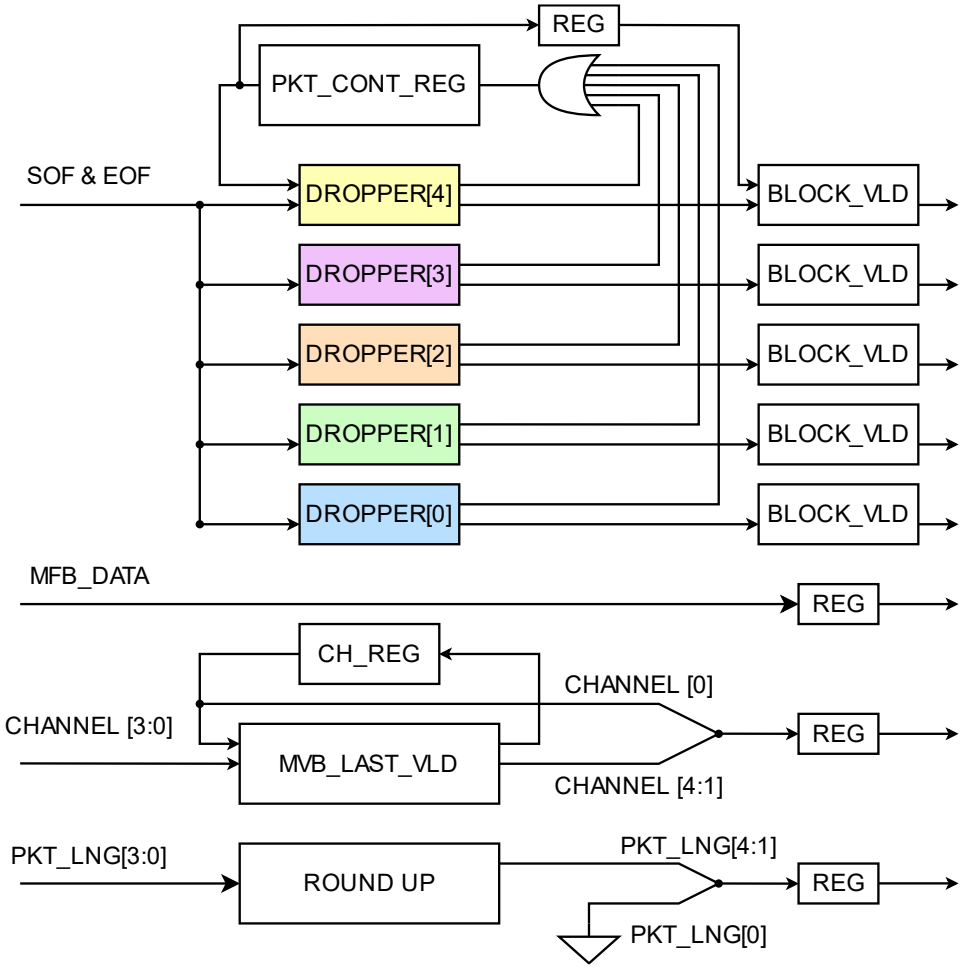


Fig. 3.6 Internal structure of the Auxiliary generator

3.3.3 Barrel Shifters

As mentioned above, the purpose of the Barrel Shifters is to minimize haps between packets. The number of Barrel Shifters depends on the REGIONS parameter of the MFB bus, or rather, on the number of packets that can be transferred over the bus in a single data word. That is because each of these packets can be routed to a different channel, thus requiring a different shift in order to match stored data further in the Channel unit. The previously generated auxiliary signals are also shifted along with the data. These include the *Block Valid* signal of each packet, SOF and EOF in one-hot format, and *Packet Length*. The packet length is valid with the SOF, so the block with the length should be placed in the same place as the SOF.

However, for simplicity the Length is copied into each block over the whole region where the SOF is, and it is extracted only from the block with the SOF.

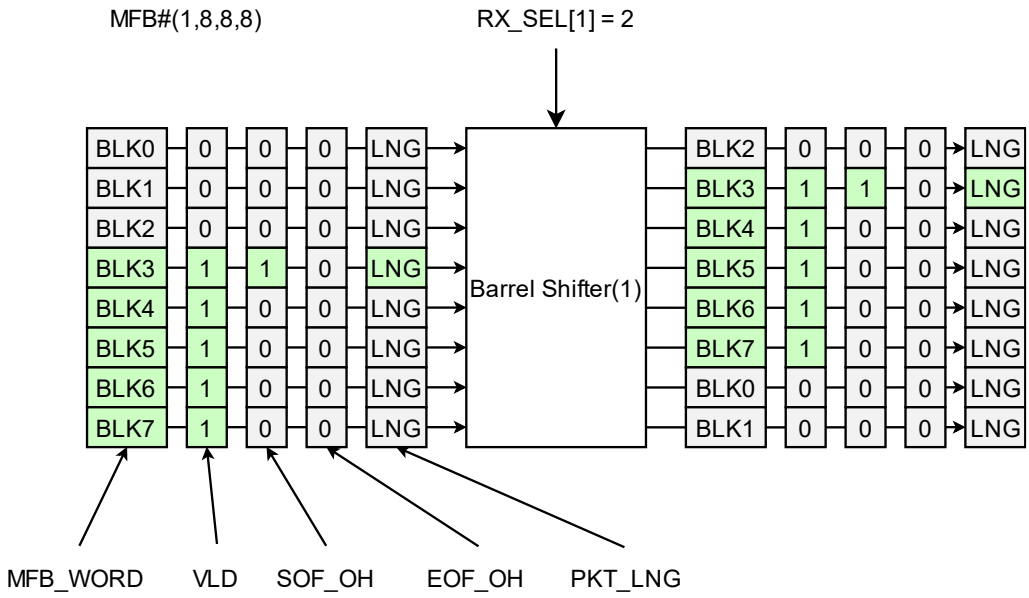


Fig. 3.7 Principle of the Barrel Shifter

3.3.4 Pointer Controller

The Pointer is a parameter that is included in the calculation of the shift select signal. Its purpose is to track the status of an *assembly register* in the given channel. Its value points to the block that will be used as a reference for aligning new data. The assembly register itself is used to hold valid blocks until the entire word register is filled with blocks. The width of the pointer register is calculated from the width of the MFB word or, rather, by the number of blocks and is extended by one bit. The extra bit is used to indicate the overflow, which means that the assembly register is full of valid blocks and is ready to be processed. The number of Pointer units is equal to the number of generated channels. The following figure shows the internal structure of each Pointer Control unit (**Figure 3.8**).

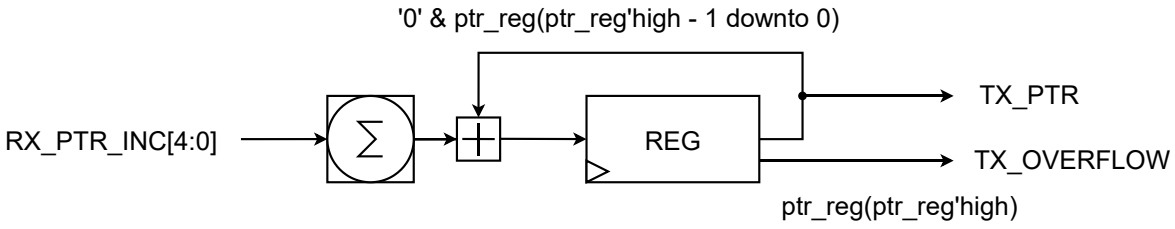


Fig. 3.8 Structure of the Pointer Controller – 4 regions

3.3.5 Barrel Shifters Controller

The principle of removing the empty spaces between packets is based on rearranging packet blocks and matching them after each other. This requires a register inside each channel that stores the incomplete words. In the scope of this thesis, it will be referred to as the *Assembly register*. The task of the Barrel Shifter Controller is to calculate the shift value for each Barrel Shifter (packet), so they do not overwrite any stored blocks, nor leave any empty blocks. The way the shift select signal is calculated is further described in this section.

Let's assume there is only one region and one channel to work with. The first packet that is processed after reset must be aligned to the beginning of the MFB word. This means that the shift of the first word with the valid blocks depends only on the SOF_POS of the incoming packet as it is a position of the first packet block within the word. Note that this packet is processed by the Barrel Shifter with index = 1.

$$TX\_SEL[1] = SOF\_POS[1]\ [blocks] \tag{3.1}$$

The pointer indicating the status of the assembly register is incremented by the number of valid blocks in the first word. Now the incomplete word is waiting in the assembly register to be completed, and the pointer is pointing to the free block to which the next valid blocks destined for this channel would be stored. Note that this part of the packet is going to be processed by the Barrel Shifter with index = 0 (assuming packet from the first word continues to another). Since the packet can't start in the first Barrel Shifter and continues for the beginning of the next word, the SOF_POS[0] is always zero. Based on this, the new shift select value is calculated as follows:

$$TX\_SEL[0] = SOF\_POS[0] - PTR\ [blocks] \tag{3.2}$$

Note that the result of this equation will end up in negative numbers, but since the signal is unsigned, the result will underflow and end up in the range 0 – 7 (for configuration MFB#(1,8,8,8)). This underflow value is still valid and suitable as the shift select value. The last case occurs when there are two packets in the word (one ends and the second begins). Both are destined for the same channel. When calculating the second packet shift, the number of valid blocks of the first packet in the word must be taken into consideration. Otherwise, the blocks would overwrite each other resulting in loss of the data. This results in the following calculation:

$$TX\_SEL[1] = SOF\_POS[1] - (PTR + sum(Block\_Valid[0])) \tag{3.3}$$

The new *alignment* is defined by the *Pointer* and the number of valid blocks of the packet ending in the same word. The timing diagram of this process is shown in **Figure 3.9**. The signal values are in the binary format. The input is labeled RX_* and the output is labeled TX_*. The first clock is the beginning of the green packet, which starts in the fifth block of the MFB word. The shift select signal with the same value as SOF_POS[1] is generated in TX_SEL[1]. The next word is full of valid blocks belonging to the green packet from the previous clock cycle. The shift select signal is based on the value of the pointer and is processed by the first

barrel shifter. In the following cycle, there are two parts of packets - green and blue. The green one ends there, while the blue one begins. For each part of the packet a separate shift select signal is generated. Note that the second select signal must take into account the number of the last valid blocks of the green packet. The last part of the blue packet arrives later and contains only the last part of the blue packet. This part is handled by the first barrel shifter, and its select signal is based on the value of the pointer.

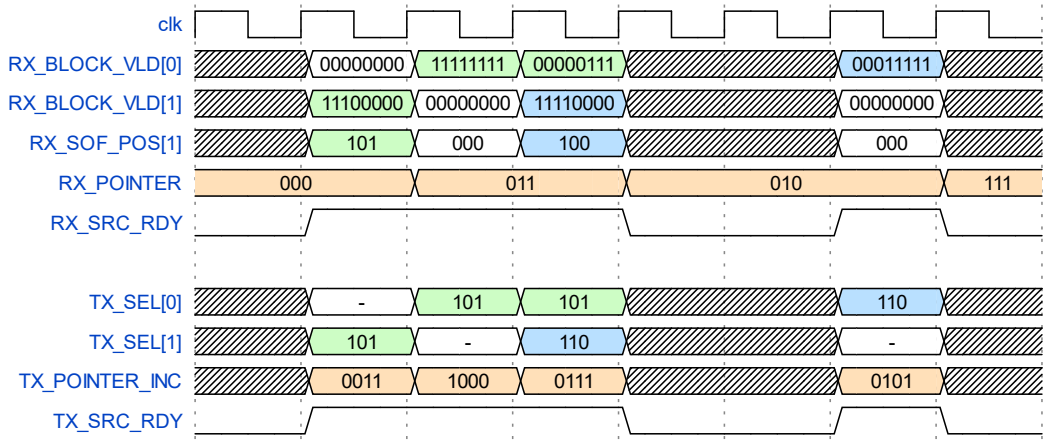


Fig. 3.9 Timing diagram of generating shift select signal

The shift select calculation described above can be represented as a digital circuit which is shown in **Figure 3.10**. Each *BS_CALC* block uses an equation derived from the previous example. The first BS_CALC[0] only considers the Pointer parameter in the calculation. Other parameters, although connected, are set to zero. This has been added to make this VHDL component generic, so it can be used multiple times if the generic number of regions is changed between 1 and 4.

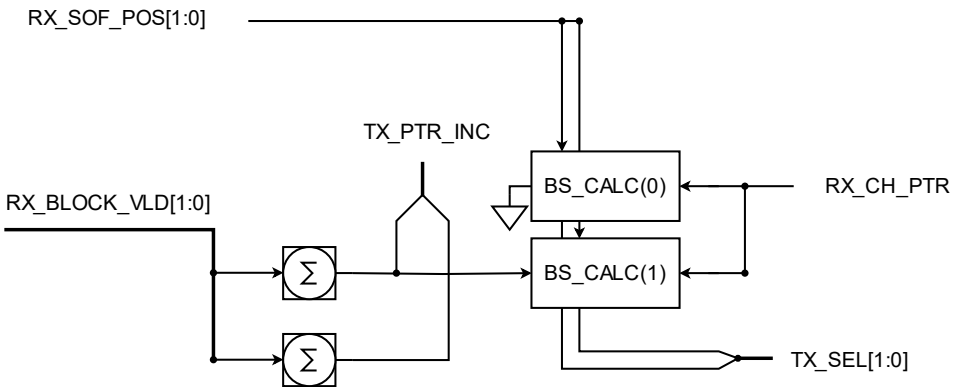


Fig. 3.10 BS_CTRL – One region and one Channel

To support multiple regions, two major changes were made. In the initial version, the BS_CALC with index = 1 must have taken into account the sum of the valid blocks of the previous packet. In the extended version, each BS_CALC component must do the same,

but with each packet that may occur in the data stream. For this reason, the number of valid blocks of packets that have been processed by BS_CALC components with lower index must be summed. The second major change is within the BS_CALC itself. Since the SOF_POS is only valid for the given region, it has to be corrected for BS_CALC components handling packets from the second region onwards. This is done by adding an offset equal to the number of blocks in the previous regions. Therefore, the final calculation is as follows:

$$TX\_SEL[n] = (n-1) \cdot MFB\_REGION\_SIZE + SOF\_POS[n] - (PTR + sum(Block\_Valid[prev])) \tag{3.4}$$

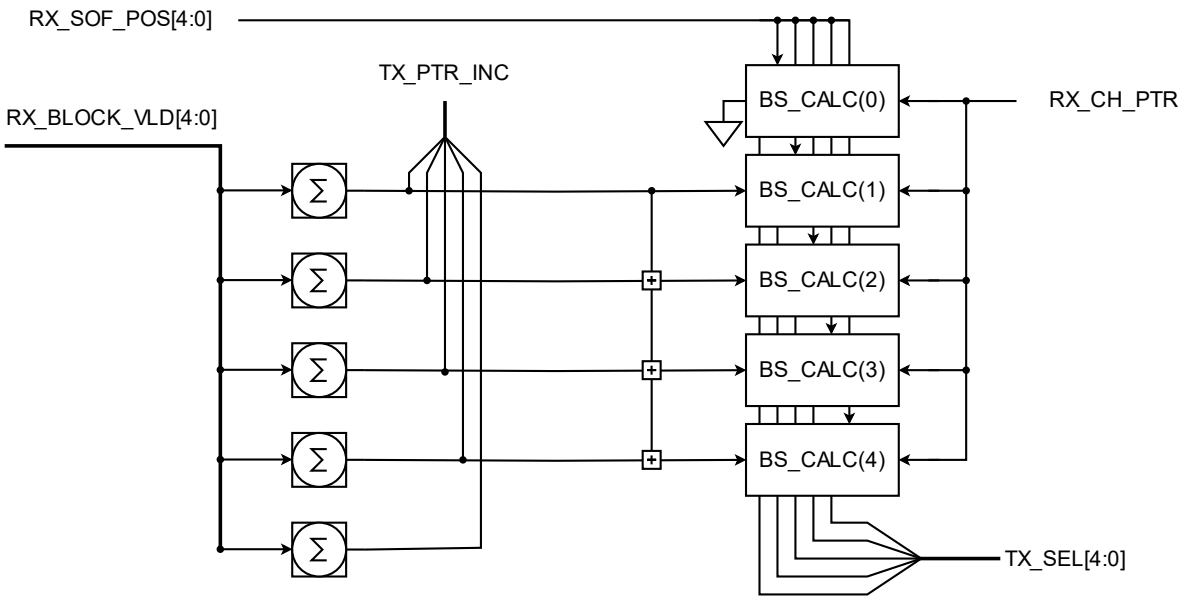


Fig. 3.11 BS_CTRL – Four regions and one Channel

The last step is extending the component to support multiple channels. This is done by adding a new input that indicates which channel each Barrel Shifter is processing. This input is called RX_CHANNELS and it is an array with a *Channel ID* value for each BS_CALC. This number is then used as a select signal to map the Pointer of the assembly register from the given channel to the calculation. Similar operation is done with each sum of valid blocks. In this case a demux is used. This component forwards the input to one of the outputs (channels) and others will be set to zero. This is necessary in cases when there are multiple packets each heading to different channel. Without demux, all these signals would be considered into calculation even though the packet is not there. This would have resulted in assembled word whose parts are invalid. At the input of the BS_CALC all these demux output are merged again with select signal from RX_CHANNELS. By doing this it is possible to catch cases when there is packet in the first and fourth region heading to the same channel as the valid blocks from the first region will propagate up to the fourth BS_CALC. The following figure shows an extended version of the initial design (**Figure 3.12**).

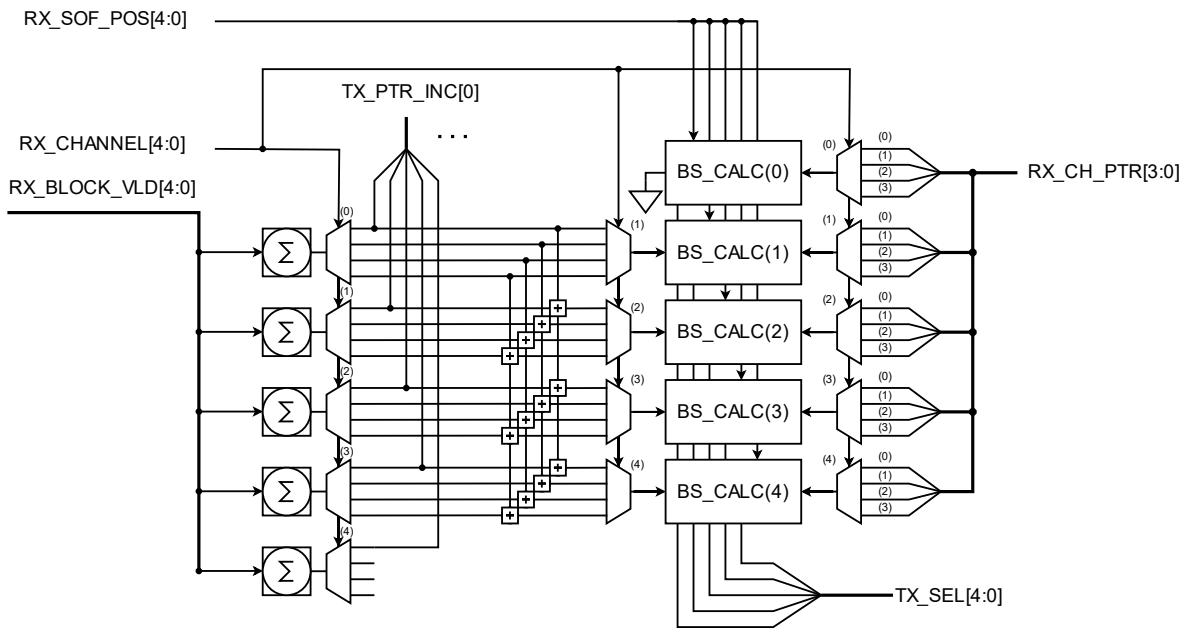


Fig. 3.12 BS_CTRL – Four regions and four Channels

3.3.6 Demux

The main purpose of the demuxes is to route auxiliary signals only to the destined channels. Each Barrel Shifter has its own DEMUX, whose select signal is the Channel ID associated with the given Barrel Shifter (or packet). Note that the demux is required only for the auxiliary signals. The data from the Barrel Shifters can be routed directly to each channel, as the *Block Valid* signal is responsible for the correct selection.

3.3.7 Channel – Packet Accumulator

The purpose of each Channel is to complete a Super-Packet from the incoming blocks with the same *Channel ID* and to ensure that it reaches the length set by the generic parameter. But the completion itself is not enough as the data would be without a context. For this reason, a MFB protocol must be recreated and must match with the SOF/SOF_POS of the first packet and EOF /EOF_POS of the last packet within the Super-Packet. For this purpose, the following structure has been designed to assemble the blocks coming from the Barrel Shifters and to decode their auxiliary signals.

At the input of the Channel component, there is an array of multiplexers whose task is to select valid blocks from each Barrel Shifter. For the MFB configuration MFB#(1,4,8,8), there are four two-input muxes (one for each block) whose select signal is generated in the MUX_CTRL subcomponent. The select signal for each multiplexer is encoded in the Block Valid array. Since the bits in the array do not interfere (thanks to the demux), the select signal is extracted as the index of the array whose block bit is set high. The same principle applies to the auxiliary signals (SOF_OH, EOF_OH, LNG).

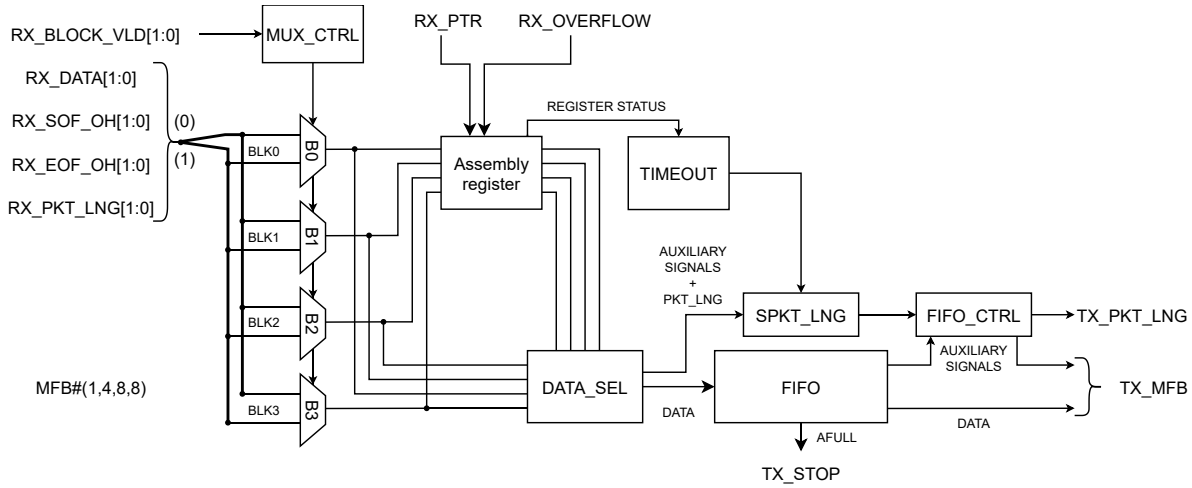


Fig. 3.13 The internal structure of the channel unit – Configuration MFB#(1,4,8,8)

Once the valid blocks have been selected, each MUX output is routed to the assembly register and to the DATA_SEL component. If the number of incoming valid blocks does not fill the data word, the incoming blocks are saved in the assembly register. To know which blocks to save, the value of the Pointer is used to lock the valid blocks until the word is ready to be processed. The DATA_SEL component is used to handle any overflow. The following figure (**Figure 3.14**) illustrates how the blocks are processed when the assembly register overflows. The blocks are shifted in such a way that the overflowed blocks are saved in the assembly register, and those that make up the word are selected by the DATA_SEL component. The overflow signal itself is used to indicate whether the word is ready or not, and the pointer is used to select the blocks that make up the word.

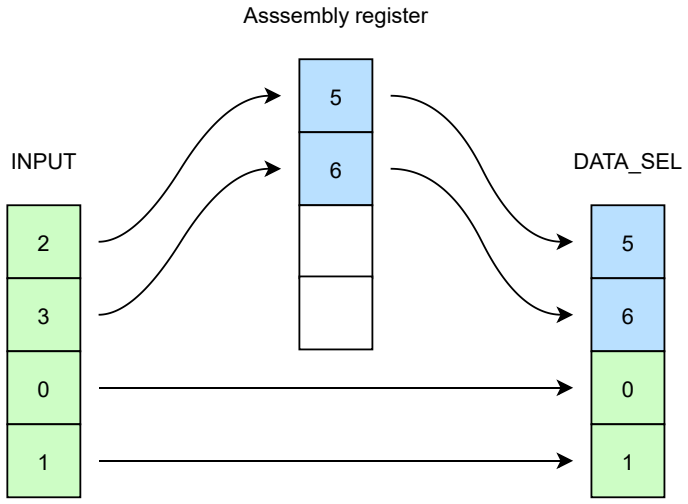


Fig. 3.14 Overflow processing

When the data word is complete, SOF_OH and EOF_OH are converted back to MFB control signals. That is SOF, EOF, SOF_POS and EOF_POS. This is done by converting

the bit that is set high to its index within the vector. Note that only one of these bits can occur in each region. Finished and aligned packets are sent to the FIFO where they wait until the Super-Packet is completed.

The last auxiliary signal to be processed is the length of the regular packets. This is extracted from the block where the SOF bit is set high. The length is then passed together with SOF and EOF to the SPKT_LNG component. The task of this component is to count the number of packets in the FIFO and sum their lengths. If the sum exceeds the minimum length of the Super-Packet, it sets the internal SRC_RDY signal to notify FIFO_CTRL that the Super-Packet is ready. The number of packets to be read from the FIFO and the length of the finished Super-Packet are sent along with this SRC_RDY signal. A timeout signal is also connected to ensure that packets are not stuck in the Channel for too long. When the timeout triggers, the Super-Packet is marked as ready, even though the desired (minimum) length has not yet been reached. The timeout component continuously checks the status of the register, and if the last valid block is marked with EOF, the timeout counter starts counting. The counter does not begin counting if there is an incomplete packet.

The last part of the Channel component is FIFO_CTRL. Its task is to wait for the SPKT_LNG to set SRC_RDY high and load its data. Once it has all the data it needs to operate, it starts to read packets from FIFO. Its task is to pass the Super-Packet length to the output of the Channel component and to create new MFB control signals, such as SOF or EOF, to form a Super-Packet. These control signals are created by masking the SOFs or EOFs of small packets coming from the FIFO. Note that this description is given for the MFB(1,4,8,8) configuration, but the component can handle more demanding configurations, such as MFB(4,8,8,8).

3.3.8 Merger

Once the Super-Packet has been created, it needs to be propagated to the output MFB stream. The merge of the several MFB streams is usually done by a component called the MFB_MERGER. Initially, this version, which is generally available in the OFM library, was used. However, it was unsatisfactory in terms of resource consumption. This was due to the structure of this component, which covers many scenarios that do not occur in this case. For this reason, a simplified version of this component was created. The internal structure of the Frame Packer's MERGER has been adapted to work as the larger multiplexer with the internal select signal. This merger takes advantage of the properties of Super-Packets and treats them as if they were carried over a single-region bus, which is, however, extended to a larger width. The select signal for this merger is internal and switches between the inputs (channels) that are marked as ready and gives each input the same opportunity to send its Super-Packet. After the Super-Packet has been sent, it tries to switch to another input marked as ready. Since this component knows from which input it is reading the data, the Channel ID is added to each packet's metadata. Together with the packet length, this metadata is sent to the MVB FIFO, which creates MVB headers for the Super-Packets.

4 TESTING

After designing the component, it was essential to test whether the proposed architecture of the Frame Packer fulfils the requirements outlined in Chapter 3.1. Firstly, a set of functional tests was performed to verify functionality of the VHDL code in corner cases. Later, the component implemented into target FPGAs was also tested on real hardware to ensure that timing requirements were met and that the resources required did not exceed the resources available on the FPGAs. Finally, performance tests were executed to ensure that the component is not blocking the traffic and meets intended throughput requirements. The performance test is also essential as the objective of this component was to increase small packet throughput via DMA and PCIe while not affecting performance for big packets. Although the aim of this thesis is to achieve 400G throughput, the 100G version was also tested. The principle and VHDL models are the same for both designs, but some cases may differ. These cases needs to be tested as well, since both versions are required to work. The reason for this is that lower throughput requires fewer resources, allowing the smaller and cheaper FPGAs to take advantage of this design. All tests that verify the functionality of the design are described in this chapter.

4.1 UVM Verification

For the purpose of functional testing, verification was implemented using the Universal Verification Methodology (UVM). The UVM is generally used to test the NDK components, as it is widely supported within the NDK platform. Its reusability allows drivers and monitors to be reused for NDK-specific protocols (MFB, MVB, and MI). Many tests, such as the compliance of the interfaces with the MFB protocol, are already implemented as a standard set of tests in the OFM library. The remaining tasks were to set up a verification environment, implement a reference verification model of the Frame Packer and define what the component should be able to do.

4.1.1 Test Environment

In the UVM verification, the test environment typically includes components for handling communication with the Device Under Test (DUT) (monitor and driver) and component for evaluating that communication (scoreboard). These components are specific for each type of interface. The ones for the MFB and MVB interfaces were already available in the OFM library. However, for the MVB, the packet length must be based on the actual length of the packets sent to the DUT. Otherwise, the length would be random and therefore useless in the context of the DUT. Because of this requirement, the standard sequencer can only be used for the Channel ID. Therefore, the lengths that go to the MVB interface are extracted from the MFB sequence items in the generator. The following figure illustrates the structure of the environment for the test of the Frame Packer (**Figure 4.1**).

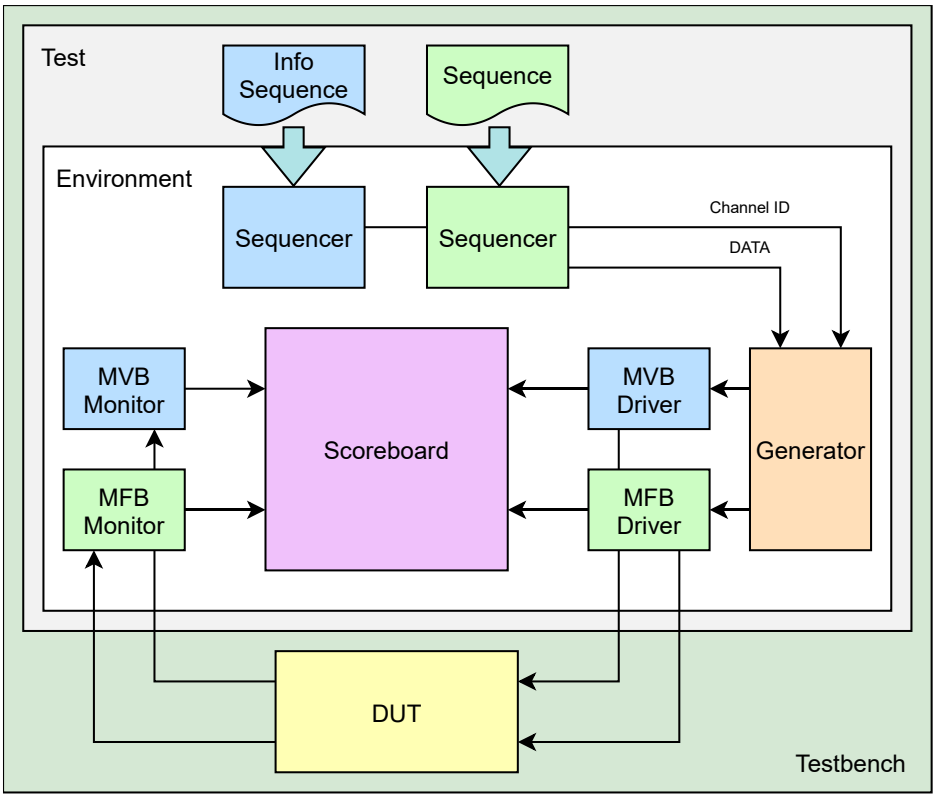


Fig. 4.1 Test Environment for the functional verification

4.1.2 Scoreboard

The role of the scoreboard is to verify that the data returned by the DUT is correct. To do this, a model of the Frame Packer is required as a reference for this evaluation. The simplest way to create a model would be to use the minimum length that all Super Packets should have and start concatenating packets from the same channel. However, this is not possible due to the timeout that can occur in the DUT. When the timeout is triggered, the currently accumulated packets are marked as a Super-Packet, even though the length has not reached the minimum size. This would not be easy to implement in the model as it is difficult to simulate the timeout in the DUT. This is due to the fact that verification processes entire transactions (packets) rather than data words like the DUT. For this reason, a probe is implemented instead. This probe is connected to each channel unit of the Frame Packer and looks at the number of packets processed and the number of packets that make up the Super-Packet. Then the model only concatenates the set number of packets from the given channel as reported by the probe. In addition, there is no need to test the timeout mechanism of the Frame Packer, as the packet length would be different, or the packets would be stuck in the design if the timeout failed. In the latter case, the timeout implemented in the comparator of the verification would trigger an error. This comparator is available in the OFM library and is used to compare the data itself. One of its features is that there is an implemented timeout which raises an error if the data from the DUT hasn't arrived within a specified time.

To test the output MVB interface, a simple if statement has been added to check that the size of the packet received is the same as the length specified in the MVB metadata. The protocol of each bus is controlled by a built-in mechanism called a *Property Check*, which is implemented in the testbench. Its task is to detect any violation of the protocol by checking control signals of the given bus. These methods guarantee the functionality of the Frame Packer and the integrity of the data.

For performance testing, the scoreboard implements the speed meter. Its only task is to measure the input and output speed. The measurement is performed periodically every 10 us of simulation time. This method does not give the exact throughput of the DUT, but it does give a pretty good insight into the state of SRC_RDY at the output. This flag indicates the readiness of the DUT to send data, i.e. whether it is able to handle new data or not. The input speed is used as a reference, as the input speed also influences the output performance. Summary statistics, such as mean and standard deviation, have also been added in order to evaluate the measured data properly.

4.1.3 Tests

In order to verify the functionality of the Frame Packer, it is necessary to define a set of test cases that cover all the requirements of the component as defined in Chapter 3.1. It would be highly impractical to test all the combinations that can occur on the input, as this number approaches infinity, therefore it is necessary to define a plan. This plan is called the verification plan and is usually based on the expected functionality of the system. For the Frame Packer, these requirements are in Appendix A and are defined as follows.

The first requirement (REQ_0) is to concatenate the packets so that they behave as one Super-Packet. This requirement is essential for the Frame Packer to work properly. The second requirement (REQ_1) is the alignment of the packets within the Super-Packet. The verification should be able to ignore empty spaces when comparing each byte of the Super-Packet. REQ_2 refers to maximum length of the packet, which is defined by a generic parameter Maximum Transfer Unit (MTU). If this requirement is violated, the DMA will most likely get stuck. REQ_3 considers the size of the Super-Packet that the Frame Packer tries to reach. It is set by a generic parameter. REQ_4 refers to the number of channels. The goal was to create a 400G version. This speed requires at least 32 channels. As the design is available for 100G (which can work with fewer DMA channels), the number of channels can also be set by a generic parameter. REQ_5 is called Channel Coherence. This requirement is essential to the operation of the digital circuit as the packets within the Super-Packet must have the same Channel ID. This requirement is extended by REQ_6 which restricts packet overtaking within the channel. Packets must be sent in the same order as they were received at the input. REQ_7 prevents packets from getting stuck in the channel for too long by using the timeout. This problem occurs when the Super-Packet is not assembled, and no other incoming packets have the same channel ID. The timeout can be set generically, so the verification should be provided for several timeout periods. Requests from 8 to 10 are related to the functional test of the component, using

different packet sizes. REQ_11 requests correct functionality of the previous tests for the MFB configuration MFB#(1,8,8,8). REQ_12 requires the functionality of the previous tests for the MFB configuration MFB#(4,8,8,8). The results are presented in the final chapter in the *Verification Results* section (Chapter 5.1).

To verify the functionality of the Frame Packer, a test called ex_test is used. It uses a basic sequence library available in the OFM repository. This sequence is partially configurable and highly randomized. The sizes of the packets sent to the DUT are randomized by several functions (Gauss, Increment, Decrement, Constant) to test the most common scenarios. To utilize these sequences, it would be necessary to write specific drivers or monitors to handle each bus interface. As the MFB (and MVB) interface is common for most of the components within the NDK, the OFM library contains reusable agents that handle that. These agents communicate directly with the signals on the respective interfaces of the DUT and also check compliance with the bus protocols described in Chapter 1.1. In addition, the agent is able to put random gaps between packets and align the packets within the word randomly. Similarly to the randomization of data length in the sequence, the agents are also configurable and can generate randomized traffic to test the DUT under different stresses.

Once the functionality of the component is tested (within the given requirements), it is useful to have an insight into its performance. The *ex_test* is not suitable for this purpose as it tries to test most of the cases that might occur. For this reason, another test has been implemented that configures the agents and sequences in such a way that the bus is fully utilized. To push the DUT to its limits, a test called *speed* was used. This test sends the data to the DUT as fast as possible and without spaces between the packets. Its only randomized parameter is the length of the packets. Gaps between packets should be minimal or non-existent. Using the speed meters in the scoreboard, it is possible to determine whether the component is able to handle full traffic.

4.2 Hardware Test

Even if the Frame Packer works in the simulation as a standalone module, this does not necessarily mean that it works in the target hardware. The main reason for the hardware tests is to measure the real performance of the component within the NDK, as the throughput provided by the verification is only an approximation.

To test any application, it is necessary to implement it in an environment similar to the final typical system. The NDK-APP-Minimal design is used for this purpose. As the name suggests, this design is NDK-based and is equipped with the minimal possible application core (see Chapter 1) in order to function. Its only purpose is to provide interconnection between the DMA and Ethernet modules. As the Frame Packer should be placed before the DMA module (in the RX path) to measure its real performance, the application core is a suitable place for testing. To distribute packets to different DMA channels, the app-core is equipped with a channel router. This component enables the user to modify the MVB header of each packet to route the packets in a couple of different modes

such as a round-robin distribution mode. The last component present in the test environment is called the MFB_PIPE. This component serves as a register for the MFB bus and helps in the final build of the design. The customized Application Core is shown in the figure below (**Figure 4.2**) and is named as NDK-APP-FramePacker in order to distinguish it from the minimal NDK design.

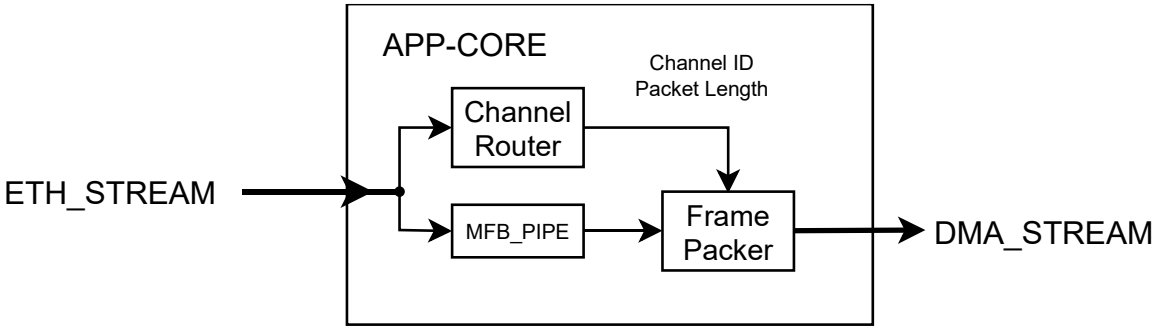


Fig. 4.2 Implementation of the Frame Packer in the Application Core

To test basic functionality, the NDK is equipped with a set of debug components. These include hardware packet generators, speed meters, debug counters, and configurable switches. Control of these integrated features is achieved using software tools, such as the NDP (Netcope Data Plane) and NFB (Netcope FPGA Board) tools. The NDP tools are used to send and read data to/from the host PC from/to the FPGA card using the PCI-express (DMA). The NFB tools are used to access the control registers of the internal components (switches, counters...). These registers are accessed via the MI network (see Chapter 1).

One of the debug components used to test the performance of the Frame Packer is called the *Generator Loopback Switch* (GLS). This component is located between the Application Core and the DMA modules and allows debugging of the NDK using, as the name suggests, packet generators, loopbacks, and switches. To measure the speed, it is also equipped with speed-meters, which are basically counters of valid bytes that can be accessed using NFB tools. The real speed is evaluated by the software with the knowledge of the frequency at which the FPGA design operates. The internal structure of the GLS is shown in the **Figure 4.3**.

In this test, the GLS is configured as follows. The MUX D is set to use the TX hardware generator. It would be possible to use software-generated packets at this point, but this would affect the throughput due to the heavier overall load on the system (PCIe, CPU, RAM). The MUX B is set to pass data from the generator to the Network module, where an internal loopback is located to return the data to the RX pipeline. This loopback is also configured using the NFB tools. When the data stream comes back, it goes through the application core (Frame Packer) and through MUX C and MUX A. At this point, the data stream flows to the software via DMA and PCI Express Modules. The throughput is measured using speed meters that are connected to the MI and controlled by MI requests from the NFB tool.

Note that the software (or any other part) does not check the correctness (integrity) of the received packets. This test only evaluates the throughput of the design.

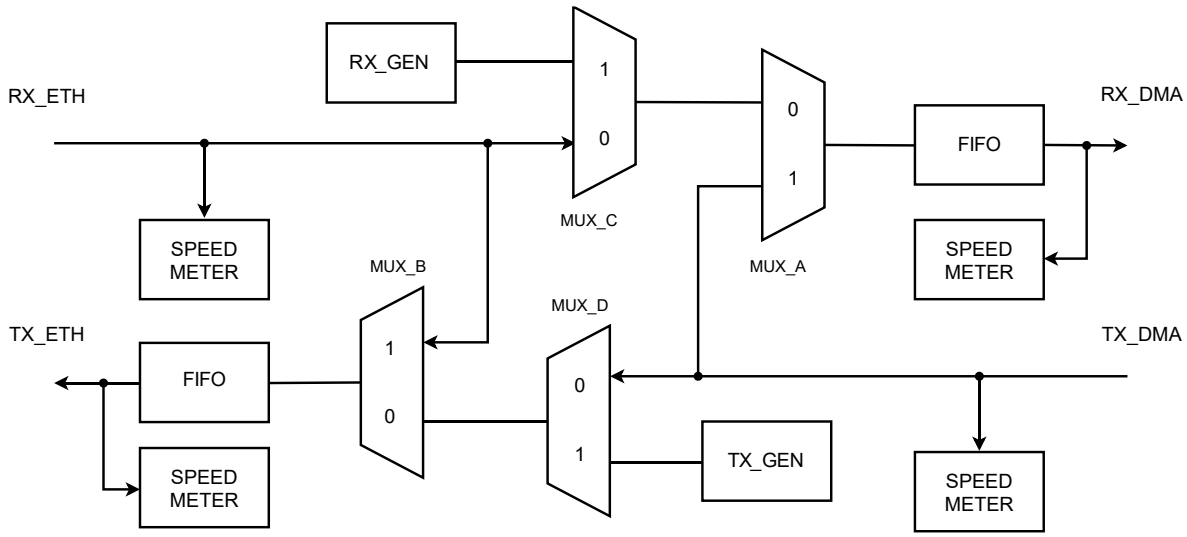


Fig. 4.3 Structure of the Generator Loopback Switch

In order to control individual registers in an automated way, there is a Python script available in OFM that allows the user to measure throughput with a single command. This script is called *gls_mod.py* and is able to output the measurement data into a .csv file. The results of this measurement are summarized in the following chapter.

5 RESULTS

The last chapter of this thesis summarizes the results of the tests presented in the previous chapter. In addition to the functional test, the results of the performance within the verification and hardware are presented as well.

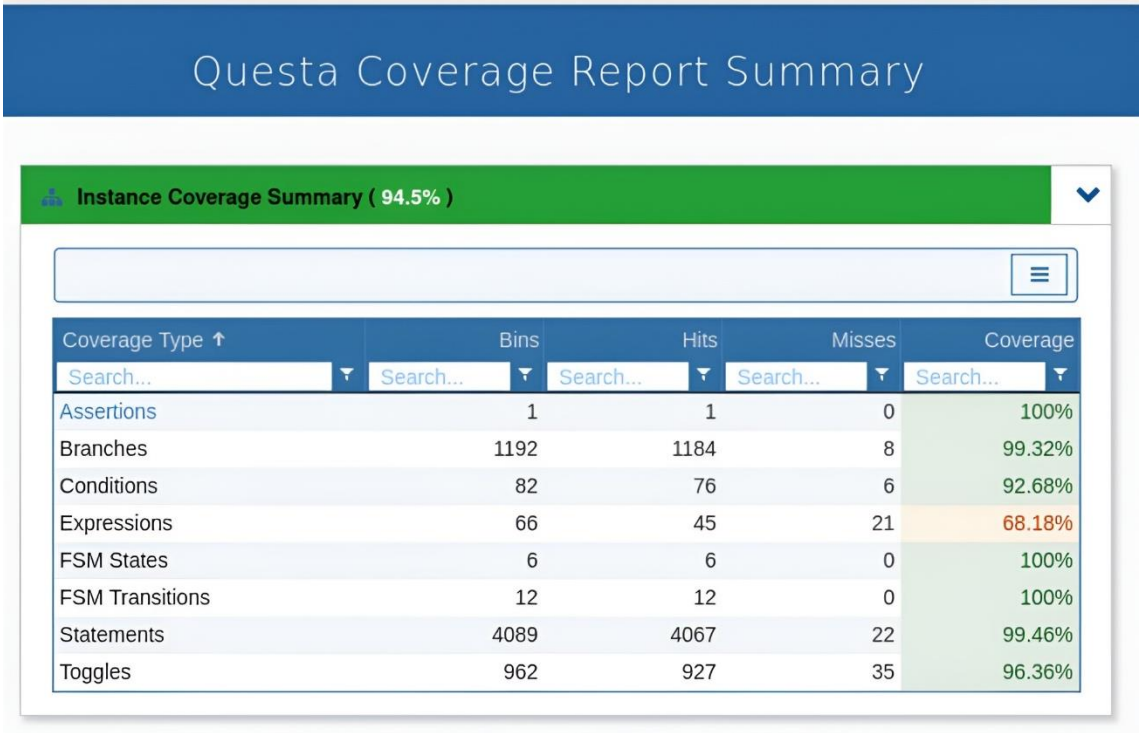
5.1 Verification Results

The functional test of the proposed design has met all the requirements described in the previous chapter. These requirements are listed in Appendix A in the form of a table. **Table 5.1** summarizes the results of the testing including the method used for testing the requirement.

Table 5.1 Results of the Verification

REQ ID	REQ Name	Test (Chapter 4.1.3)	Status
REQ_0	Packet Concatenation	test::ex_test	PASS
REQ_1	Packet Alignment	test::ex_test	PASS
REQ_2	Maximum Packet Size	test::ex_test	PASS
REQ_3	Optimal Packet Size	test::ex_test	PASS
REQ_4	Multiple Channels	test::ex_test	PASS
REQ_5	Channel Coherence	test::ex_test	PASS
REQ_6	Channel Order	test::ex_test	PASS
REQ_7	Timeout	test::ex_test + TIMEOUT = 32, 1024, 4096	PASS
REQ_8	Small Packet Test	test::ex_test + MAX_SIZE = 128	PASS
REQ_9	Big Packet Test	test::ex_test + MIN_SIZE = 2**13	PASS
REQ_10	Random Size	test::ex_test	PASS
REQ_11	One Region	test::ex_test + REGIONS = 1	PASS
REQ_12	Four Regions	test::ex_test + REGIONS = 4	PASS

Although the verification has covered all test cases, it does not necessarily mean the design is bug-free. There may still be combinations that weren't tested or thought of. For this reason, a "test of the test" was performed in the form of code coverage. This method can give feedback on the quality of the verification by returning a percentage of the code checked during the verification runs. This percentage includes covered branches, conditions, signal toggles, statements, or even FSM transitions.



Coverage Type	Bins	Hits	Misses	Coverage
Assertions	1	1	0	100%
Branches	1192	1184	8	99.32%
Conditions	82	76	6	92.68%
Expressions	66	45	21	68.18%
FSM States	6	6	0	100%
FSM Transitions	12	12	0	100%
Statements	4089	4067	22	99.46%
Toggles	962	927	35	96.36%

Fig. 5.1 Report of the Code Coverage – Questa Sim

The report shown in **Figure 5.1**, generated by Questa Sim, has returned a coverage of 94.5%. This result was obtained after excluding all coverage types that are not relevant to the verification of the Frame Packer. For example, this was a generic input of the subcomponents (such as the DEVICE name) or the toggle of the reference signals (SOF_POS(0)). The remaining 5% might appear like a big number, but it is (ironically) caused by parts of the code that were added to fix errors found by the *ex_test* with different seeds. In conclusion, there is still room for improvement in the verification, but it would be overwhelmingly time-consuming, as these cases were detected in specific scenarios that are not so easy to replicate.

Note that the coverage is merged from multiple test runs. This was performed using *multi_ver.py*, a Python script for running multiple tests with different generic parameters. In this case, the number of regions of the MFB word was changed, along with the different timeout length and Super-Packet size.

To evaluate the performance within the verification, a speed-meter located in the scoreboard was used. This speed-meter is a simple construct that counts the number of bytes received

in the specified period of the simulation time. At the end of the verification, the function returns a set of statistical values such as minimum (min), maximum (max), average (avg) and standard deviation (st_dev). These results are given for the output speed and length. For testing purposes, the value of the generic parameters was selected as follows. The FIFO accumulating Super-Packets in each channel is set to 512 items. This is 131072 bytes of memory per channel in the case of 4 regions and 32768 bytes for 1 region. This FIFO depth was chosen as it is the most efficient for the utilization of the BRAM blocks. The TIMEOUT was set to 4096 clock cycles as this resulted in best performance in previous tests. The number of channels was chosen based on the real application requirements for the given throughput.

The **Table 5.2** and **Table 5.3** summarize the performance results for the 100G and 400G versions in the *speed* test verification. The tables show the statistical value for the output speed and packet length for the given parameters SPKT_SIZE_MIN and FRAME_SIZE_MAX. The SPKT_SIZE_MIN parameter represents the length of the Super-Packet that is being targeted. FRAME_SIZE_MAX represents the maximum length of the INPUT packets. The minimum value is set to 64B as this is the minimum length of the Ethernet packet. The length of the input packets is randomly distributed within the set range. The purpose of this parameter is to give an insight into the performance in different length ranges. Note that the throughput value is only a hint to whether the designed architecture can handle the target speed or not. The actual throughput can be obtained from the hardware performance test, the results of which are presented in the following section.

Table 5.2 Results of Verification performance test – 100 G

REGIONS = 1, TIMEOUT = 4096, FIFO_DEPTH = 512, CHANNELS = 16									
TX_Speed [Gbps]				**TX_Length [B]**					
min	max	avg	st_dev	min	max	avg	st_dev	SPKT_SIZE_MIN [B]	FRAME_SIZE_MAX [B]
3	98	96	9	96	1016	967	40	1024	128
0	100	99	8	104	2040	1987	87	2048	128
5	102	99	9	72	4088	4025	201	4096	128
7	102	99	11	168	8184	8066	656	8192	128
0	97	94	9	72	1024	780	165	1024	1024
1	100	98	9	456	2040	1742	215	2048	1024
5	102	100	8	744	4088	3700	307	4096	1024
7	102	101	9	392	8184	7799	446	8192	1024
17	102	98	11	72	8160	3673	2177	1024	8192
3	102	98	12	88	8184	3771	2068	2048	8192
1	102	99	10	96	8168	4625	1974	4096	8192
6	102	99	14	592	8192	5947	1354	8192	8192

Table 5.3 Results of Verification performance test – 400 G

REGIONS = 4, TIMEOUT = 4096, FIFO_DEPTH = 512, CHANNELS = 32									
TX_Speed [Gbps]				TX_Length [B]					
min	max	avg	st_dev	min	max	avg	st_dev	SPKT_SIZE_MIN [B]	FRAME_SIZE_MAX [B]
8	332	271	116	248	1200	862	113	1024	128
16	365	209	174	248	2232	1813	284	2048	128
12	388	307	136	496	4088	3837	440	4096	128
2	401	333	130	408	8184	7427	1599	8192	128
18	321	309	12	128	1208	775	176	1024	1024
12	365	337	80	904	2040	1736	249	2048	1024
9	389	364	81	944	4088	3707	357	4096	1024
5	404	376	87	800	8184	7712	887	8192	1024
26	398	366	74	88	8176	4074	2620	1024	8192
9	398	355	97	96	8080	4877	1945	2048	8192
29	399	385	32	88	8192	4718	1711	4096	8192
30	398	360	90	1616	8176	6002	1181	8192	8192

The output throughput of the Frame Packer is given by the effectiveness of the merging mechanism. If the final Super-Packet is smaller, the MERGER must read from each channel more often. This creates overhead because there are empty spaces between Super-Packets. This effect can be reduced by making the Super-Packet larger, as the results from the previous tables (**Table 5.2**, **Table 5.3**) suggest. Especially for smaller input packets.

5.2 Implementation Results

Since the theoretical functionality of the Frame Packer has been verified, it would make sense that it would work in hardware. In reality, however, it may not be that simple, as the resources and maximum frequency on the FPGA are limited. The Frame Packer is configurable, so two versions were implemented: 100G and 400G with the required number of channels for each speed. Results for the 100G version are available for both Intel and AMD FPGAs. The 400G version is only available for Intel's Agilex family of FPGAs. This part of the thesis is dedicated to the implementation of the mentioned versions on the FPGA with and without NDK minimal design. The version of the design with the Frame Packer is called NDK-APP-FramePacker to distinguish it from the NDK-APP-Minimal. The last part of this chapter is dedicated to testing NDK-APP-FramePacker on real hardware to evaluate its real performance. The following tables present FPGA resource usage for the 100G version (Intel, AMD) of the Frame Packer.

For such speed, at least 16 Channels are required.

Table 5.4 Frame Packer-100G & 16 Channels - Intel Stratix 10 1SD280PT2F551VG

	Used	Available	Percentage
ALM	19352	933120	2%
ALUT	29376	N/A	N/A
FF	17207	N/A	N/A
M20K (BRAM)	225	11721	2%
DSP	0	5760	0%
Memory ALUT	256	N/A	N/A
LAB	2647	93312	3%

Table 5.5 Frame Packer-100G & 16 Channels - AMD Virtex UltraScale+ xcvu7p

	Used	Available	Percentage
LUT	26245	788160	3.33%
LUTRAM	160	394560	0.04%
FF	15983	1576320	1.01%
BRAM	128.5	1440	8.92%

The 100G version of the Frame Packer on the Intel device requires about 2% of the Adaptive Logic Modules (ALMs), 30,000 Adaptive Look-Up Tables (ALUTs), and about 17,000 Flip-Flops (FFs). Similar results for LUTs and FFs are also found on the AMD FPGA. These results are positive when compared to the minimal NDK design (more to that later) and when considering the impact on performance of the NDK system. While Intel's BRAM utilization is around 2%, AMD's is close to 9%. This is due to differences in available BRAMs on AMD FPGA and the current optimization of Frame Packer FIFOs for Intel's M20K BRAMs. BRAMs are a critical resource for NDK application development, and maximizing their efficiency is important. In terms of timing, the maximum frequency that can be reached on the Stratix 10 FPGA is 199.4 MHz, while the implementation on the UltraScale+ FPGA can reach up to 213.17 MHz. Therefore, both implementations can run on the FPGA because the frequency required for 100G transfer is 200 MHz.

The resources required for the 400G version are shown in the following tables. There are three examples of resource usage for three different configurations: 8 channels, 16 channels, and 32 channels. As you can see, as the number of channels in the design increases, so does the number of resources and with that the timing requirements as well.

Table 5.6 Frame Packer-400G & 8 Channels - Intel Agilex AGIB027R29A1E2VR0

	Used	Available	Percentage
ALM	84964	912800	9%
ALUT	131869	N/A	N/A
FF	65110	N/A	N/A
M20K (BRAM)	426	13272	3%
DSP	0	8528	0%
Memory ALUT	960	N/A	N/A
LAB	10876	91280	12%

Table 5.7 Frame Packer-400G & 16 Channels - Intel Agilex AGIB027R29A1E2VR0

	Used	Available	Percentage
ALM	139194	912800	16%
ALUT	223091	N/A	N/A
FF	76247	N/A	N/A
M20K (BRAM)	850	13272	6%
DSP	0	8528	0%
Memory ALUT	1024	N/A	N/A
LAB	18774	91280	21%

Table 5.8 Frame Packer-400G & 32 Channels - Intel Agilex AGIB027R29A1E2VR0

	Used	Available	Percentage
ALM	253479	912800	28%
ALUT	409099	N/A	N/A
FF	127722	N/A	N/A
M20K (BRAM)	1698	13272	13%
DSP	0	8528	0%
Memory ALUT	1088	N/A	N/A
LAB	34083	91280	37%

The first thing to notice is that the FPGA used is much larger than that used in the 100G version. The 32-channel version of the Frame Packer would require 28% of the ALMs, 409,099 ALUTs, and almost 13% of the BRAMs. The timing requirements for this version are so high

that it would only run at 92.36 MHz. The 16-channel version requires 16% of the ALMs, 223,091 ALUTs and 6% of the BRAMs. The maximum frequency goes up to 156.99 MHz. Still not enough, but it shows a trend that can be used for the next generation of Frame Packer. The last version with 8 channels needs 9% of the ALMs, 131,869 ALUTs, and 3% of the BRAMs. The maximum frequency is 207.81 MHz, which meets the timing requirements. These results point to the main problem with the Frame Packer - the channels. Each of them consumes a significant number of resources, which leads to problems when more channels are used. This brings us to the idea of a next generation Frame Packer where a number of channels are set to be shared or dynamically allocated. This would reduce the resources needed and improve timing requirements.

The final part, which involves building the design, are the tables summarizing the resource usage of NDK-APP-Minimal and NDK-APP-FramePacker. The following table shows the 100G build for the SmartNIC N6010 FPGA card. The FPGA on this card (Intel(R) Agilex ™ AGF014) is slightly smaller than the one used in the previous builds but has better timing characteristics. The aim of this part is to provide a context of the size of the Frame Packer within the NDK design.

Table 5.9 NDK-APP-Minimal - 100G SmartNIC N6010

	Used	**Available**	**Percentage**
ALM	198706	487200	41%
ALUT	193792	N/A	N/A
FF	332198	N/A	N/A
M20K (BRAM)	1140	7110	16%
DSP	16	4510	<1%
Memory ALUT	25066	N/A	N/A
LAB	24960	48720	51%

Table 5.10 NDK-APP-FramePacker - 100G SmartNIC N6010

	Used	**Available**	**Percentage**
ALM	227819	487200	47%
ALUT	237332	N/A	N/A
FF	364616	N/A	N/A
M20K (BRAM)	1372	7110	19%
DSP	16	4510	<1%
Memory ALUT	25808	N/A	N/A
LAB	28913	48720	59%

The 100G version of the NDK-APP-Minimal requires at least 41% of the ALMs, 193,792 of the ALUTs, and 16% of the BRAMs. With the addition of the Frame Packer, these numbers jump to 47% for ALMs, 237,332 for ALUTs, and 19% for BRAMs. These numbers suggest the Frame Packer is suitable for the 100G transfer rate.

The last build would be for the 400G design, but unfortunately this build failed due to the high demands on the FPGA resources. Quartus was unable to route the design within the NDK. For context, only the minimal version of the 400G NDK design is presented. The only board available for this build is the XpressSX AGI-FH400G with Intel Agilex I-Series FPGA which is proportionally bigger than FPGAs used for 100G transfer rate. Similar to the 100G version of the NDK, the 400G design uses about 50% of the logic resources and 20% of the BRAMs.

Table 5.11 NDK-APP-Minimal - 400G XpressSX AGI-FH400G

	Used	**Available**	**Percentage**
ALM	457655	912800	50%
ALUT	445394	N/A	N/A
FF	739740	N/A	N/A
M20K (BRAM)	2458	13272	19%
DSP	20	8528	<1%
Memory ALUT	50951	N/A	N/A
LAB	58481	91280	64%

5.3 Hardware Test Results

The last part of this section is dedicated to performance testing using the previously created builds for the N6010 (Intel) and NFB-200G2QLP (AMD). Note that the card with the AMD FPGA only runs at 100G, even though it is marked as 200G. Due to the high resource usage of the 400G Frame Packer version, only the 100G version is tested.

The first figure (**Figure 5.2**) shows the throughput of the system versus the length of the packets sent by the NDK design. This design was uploaded to the N6010 and tested on machines with different processor types. From left to right is the performance of the NDK-APP-Minimal on the machine with an Intel processor. The performance on the smaller packets is low due to the handling of PCIe credits and DMA descriptors. The same applies to the minimal design on the machine with an AMD processor. The difference between them is due to the nature of handling communication. The red line in the graph represents the ideal Ethernet throughput, and just above it is the line showing the throughput with the NDK-APP-FramePacker design. The reason it performs better than the theoretical Ethernet is due to the way the NDK measures the data flow. The speed meter does not unpack the Super-Packets, so the empty spaces within the Super-Packet are also counted. These empty spaces have a greater effect on the smaller

packets, while the large packets are able to suppress this effect. Just for a reminder, these spaces are there because of the block alignment.

Note that these tests were executed on host PCs named Sevar and Laurot. The Sevar is equipped with the Intel(R) Xeon(R) CPU E5-2630 and 64GB of DDR4 RAM, while the Laurot is equipped with the AMD EPYC 7402 24-core processor and 128 GB of DDR4 RAM. Both use the PCIe 4.0 to communicate with the FPGA card. The Frame Packer was tested on both host PCs and performed equally well.

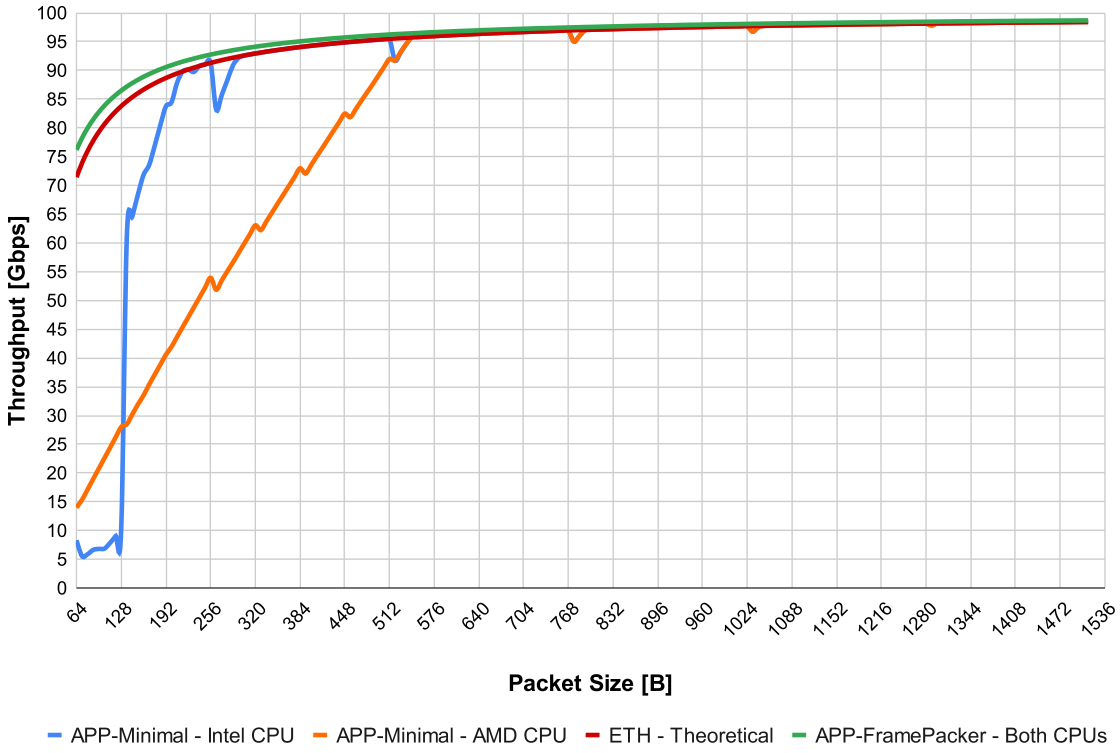


Fig. 5.2 Hardware performance test – Intel FPGA (N6010)

The second measurement (**Figure 5.3**) is performed on the NFB-200G2QLP FPGA card on a host PC called Emilion. This machine is equipped with an Intel(R) Xeon(R) CPU E5-2620 and 64GB DDR4 of memory. In contrast to the previous test, this card did not perform so well compared to the Agilex family of FPGAs. This is unfortunate for the NDK, but great for the Frame Packer, as the measurement proves that the designed component is able to boost the performance even if the target card/system is not as powerful. Another difference between these graphs is that the throughput of the 64B packets starts at 90 Gbps. This indicates that there may be a problem with this measurement, such as an incorrect working frequency. This could affect the measurement as it is the software that calculates the speed, and it uses an application core frequency to do so. This measurement should therefore be taken with a grain of salt.

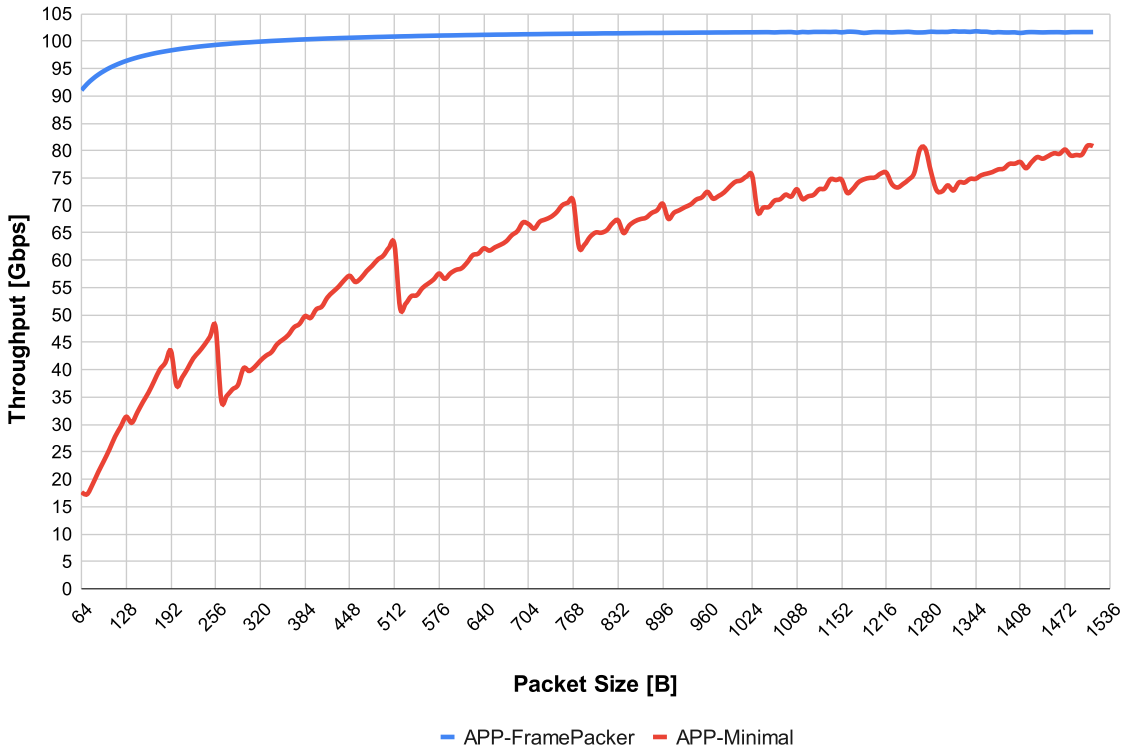


Fig. 5.3 Hardware performance test – AMD FPGA (200G2QLP)

CONCLUSION

The goal of this thesis was to gain a full understanding of the NDK framework and its specific communication protocols in order to create a compatible digital circuit. The purpose of this digital circuit was to accumulate packets from the same virtual lanes at transfer rates up to 400 Gbps. The reason for creating such a digital circuit is that due to the transfer overhead on the PCIe and inefficient descriptor usage by the DMA controller, the throughput over the PCIe (DMA) is low when transferring small packets. Both beforementioned issues can be reduced by creating larger units from multiple smaller packets. The digital circuit capable of creating such units from the accumulated packets is called the Frame Packer.

The final design was achieved after creating several prototypes that helped uncover the problem and take them into account when designing the current version. The obstacles present in most of the prototypes were the control mechanism of the Frame Packer and the amount of FPGA resources required. Problems with the management of the packet alignment were solved by dividing the Frame Packer into several stages and a control unit that performed a simple calculation at the end. The issues with the resources were mostly solved by using integrated memory blocks (BRAMs) instead of common logic resources.

The final goal was to verify the functionality of the designed architecture using the UVM methodology. For this purpose, a verification plan was created based on the requirements listed in Chapter 3. These represent the minimum the digital circuit must meet to function properly within the NDK. As concluded in the last chapter, the design has met all the requirements stated in the verification plan. The last chapter also summarizes the code coverage and the performance tests within the verification and hardware. The final result of the code coverage is 94.5%. The verification performance test showed that the design is able to handle high-speed transfers without slowing down the traffic. The performance test that used the real network card proved the 100G version is able to increase the transfer rate for small packets significantly.

The 400G version of the Frame Packer also passed the verification. As for the implementation, it was possible to implement an 8-channel variant that could operate at 200 Mhz. When implementing the design in the NDK, a 400G design expects at least 32 channels (due to the DMA requirements). This makes it quite difficult to implement the Frame Packer in the available FPGA because it requires many resources. Therefore, optimization is needed to reduce the number of physical channels. This could be done by sharing a physical channel with multiple virtual lanes.

In terms of real-world applications, based on the results, this component could be used in monitoring high-speed networks, particularly networks up to 100 Gbps. That is because the 100G version of the Frame Packer requires an adequate number of resources. On the other hand, this component is not suitable for applications such as High Frequency Trading (HFT), as this component increases the transmission latency. This is due to the nature of the component, which is based on buffering and concatenating packets.

LITERATURE

[1] CESNET Z.S.P.O. *Network Development Kit* [online]. [cit. 2023-11-17]. Dostupné z: https://www.liberouter.org/ndk/

[2] CESNET Z.S.P.O. *CORE of Network Development Kit* (NDK) [online]. 2023 [cit. 2023-11-17]. Dostupné z: https://github.com/CESNET/ndk-core

[3] CESNET Z.S.P.O. *OFM repository* [online]. 2023 [cit. 2023-11-17]. Dostupné z: https://github.com/CESNET/ofm

[4] CESNET Z.S.P.O. *FPGA cards files with open support for the NDK* [online]. 2023 [cit. 2023-11-17]. Dostupné z: https://github.com/CESNET/ndk-cards-open

[5] CESNET Z.S.P.O. *NDK architecture* [online]. 2023 [cit. 2023-11-17]. Dostupné z: https://cesnet.github.io/ndk-app-minimal/devel/ndk_core/intel/readme.html

[6] CESNET Z.S.P.O. *The Network Module* [online]. 2023 [cit. 2023-11-17]. Dostupné z: https://cesnet.github.io/ndk-app-minimal/devel/ndk_core/intel/doc/eth.html

[7] CESNET Z.S.P.O. *The PCIe module* [online]. 2023 [cit. 2023-11-17]. Dostupné z: https://cesnet.github.io/ndk-app-minimal/devel/ndk_core/intel/doc/pcie.html

[8] CESNET Z.S.P.O. *The Memory Controller* [online]. 2023 [cit. 2023-11-17]. Dostupné z: https://cesnet.github.io/ndk-app-minimal/devel/ndk_core/intel/doc/mem.html

[9] CESNET Z.S.P.O. *The DMA module* [online]. 2023 [cit. 2023-11-17]. Dostupné z: https://cesnet.github.io/ndk-app-minimal/devel/ndk_core/intel/doc/dma.html

[10] CESNET Z.S.P.O. *Time Stamp Unit* [online]. 2023 [cit. 2023-11-17]. Dostupné z: https://cesnet.github.io/ndk-app-minimal/devel/ndk_core/intel/doc/tsu.html

[11] CESNET Z.S.P.O. *MI specification* [online]. 2023 [cit. 2023-11-17]. Dostupné z: https://cesnet.github.io/ndk-app-minimal/devel/ofm_doc/comp/mi_tools/readme.html

[12] CESNET Z.S.P.O. *MVB Specification* [online]. 2023 [cit. 2023-11-17]. Dostupné z: https://cesnet.github.io/ndk-app-minimal/devel/ofm_doc/comp/mi_tools/readme.html

[13] CESNET Z.S.P.O. *MFB specification* [online]. 2023 [cit. 2023-11-17]. Dostupné z: https://cesnet.github.io/ndk-app-minimal/devel/ofm_doc/comp/mfb_tools/readme.html

[14] CESNET Z.S.P.O. MI *Transaction Controller* [online]. 2023 [cit. 2023-11-17]. Dostupné z: https://cesnet.github.io/ndk-app-minimal/devel/ofm_doc/comp/pcie/mtc/readme.html

[15] CESNET Z.S.P.O. *N_LOOP_OP* [online]. 2023 [cit. 2023-11-17]. Dostupné z: https://cesnet.github.io/ndk-app-minimal/devel/ofm_doc/comp/base/logic/n_loop_op/readme.html#n-loop-op

[16] KEKELY, Lukáš, Jakub CABAL, Viktor PUŠ a Jan KOŘENEK. *Multi Buses: Theory and Practical Considerations of Data Bus Width Scaling in FPGAs* [online]. Kranj, Slovenia: IEEE, 2020 [cit. 2023-11-17]. Dostupné z: https://ieeexplore.ieee.org/document/9217811

[17] ARM LTD. *AMBA® AXI-Stream: Protocol Specification* [online]. [cit. 2023-11-17]. Dostupné z: https://documentation-service.arm.com/static/64819f1516f0f201aa6b963c?token=

[18] INTEL. *Avalon® Interface Specifications* [online]. [cit. 2023-11-17]. Dostupné z: https://cdrdv2-public.intel.com/667068/mnl_avalon_spec-683091-667068.pdf
[19] SCIENCE DIRECT. *Ethernet overview* [online]. [cit. 2023-11-17]. Dostupné z: https://www.sciencedirect.com/topics/computer-science/minimum-frame-size
[20] AMD/XILINX. *Understanding Performance of PCI Express Systems* [online]. https://docs.xilinx.com/v/u/en-US/wp350 [cit. 2023-11-18].
[21] WHANGBO, Joonho, Kevin ANDERSON a Coleman HOOPER. *Exploring Industry Standard System Integration for Hardware-Software Co-Design of PCI Express* [online]. 2022 [cit. 2023-11-18]. Dostupné z: https://people.eecs.berkeley.edu/~kubitron/courses/cs262a-F22/projects/reports/project5_report_ver2.pdf
[22] MCLELLAN, Paul. CADENCE. *The History of PCIe* [online]. 2021 [cit. 2023-11-18]. Dostupné z: https://community.cadence.com/cadence_blogs_8/b/breakfast-bytes/posts/pcie-the-next-generation
[23] XDA-DEVELOPERS. *PCI Express 5 (PCIe 5.0)* [online]. 2023 [cit. 2023-11-18]. Dostupné z: https://www.xda-developers.com/pcie-5/
[24] VERIEN DESIGN GROUP. *PCI Express Topology* [online]. 2018 [cit. 2023-11-18]. Dostupné z: http://www.verien.com/pcie-primer.html
[25] GLOBALSPEC. *Data Link Layer Architecture* [online]. 2003 [cit. 2023-11-18]. Dostupné z: https://www.globalspec.com/reference/66552/203279/chapter-7-data-link-layer-architecture
[26] SOUTHWELL, Simon. *PCI Express Primer #2: Data Link Layer* [online]. 2022 [cit. 2023-11-18]. Dostupné z: https://www.linkedin.com/pulse/pci-express-primer-2-data-link-layer-simon-southwell
[27] SOUTHWELL, Simon. *PCI Express Primer #1: Overview and Physical Layer* [online]. 2022 [cit. 2023-11-18]. Dostupné z: https://www.linkedin.com/pulse/pci-express-primer-1-overview-physical-layer-simon-southwell/
[28] OPEN4TECH. *Direct Memory Access (DMA) in Embedded Systems* [online]. [cit. 2023-12-24]. Dostupné z: https://open4tech.com/direct-memory-access-dma-in-embedded-systems/
[29] AMD/XILINX. *DMA/Bridge Subsystem for PCI Express Product Guide (PG195)* [online]. [cit. 2023-12-24]. Dostupné z: https://docs.xilinx.com/r/en-US/pg195-pcie-dma/Descriptors
[30] AMD/XILINX. *Stream Mode DMA* [online]. [cit. 2023-12-26]. Dostupné z: https://docs.xilinx.com/r/en-US/pg347-cpm-dma-bridge/Stream-Mode-DMA?tocId=ozsDQ5~waYmCNHjO7tyQFQ
[31] KUBÁLEK, Jan, Jakub CABAL, Martin ŠPINLER a Radek IŠA. IEEE. *DMA Medusa: A Vendor-Independent FPGA-Based Architecture for 400 Gbps DMA Transfers* [online]. 2021 [cit. 2023-12-26]. Dostupné z: https://ieeexplore.ieee.org/document/9444087

[32] CESNET Z.S.P.O. *DMA Calypte* [online]. [cit. 2023-12-26]. Dostupné z: https://cesnet.github.io/ndk-app-minimal/devel/ofm_doc/comp/dma/dma_calypte/readme.html

[33] CESNET Z.S.P.O. *RX DMA Calypte* [online]. [cit. 2023-12-26]. Dostupné z: https://cesnet.github.io/ndk-app-minimal/devel/ofm_doc/comp/dma/dma_calypte/comp/rx/readme.html

[34] INTEL. *DMA Operation Flow* [online]. [cit. 2023-12-26]. Dostupné z: https://www.intel.com/content/www/us/en/docs/programmable/683390/quartus-prime-pro-v17-0-arria-10/dma-operation-flow.html

[35] OPEN4TECH. *Direct Memory Access (DMA) in Embedded Systems* [online]. [cit. 2023-12-26]. Dostupné z: https://open4tech.com/direct-memory-access-dma-in-embedded-systems/

[36] CESNET Z.S.P.O. *Metadata Insertor* [online]. [cit. 2024-04-09]. Dostupné z: https://github.com/CESNET/ofm/tree/main/comp/mfb_tools/flow/metadata_insertor

[37] CESNET Z.S.P.O. *Auxiliary signals* [online]. [cit. 2024-04-09]. Dostupné z: https://github.com/CESNET/ofm/tree/main/comp/mfb_tools/logic/auxiliary_signals

SYMBOLS

Abbreviations:

ACK	Acknowledgement
AVMM	Avalon Memory-Mapped Interface
AXI	Advanced Extensible Interface
BRAM	Block Random Access Memory
BUT	Brno University of Technology
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DC	Direct Current
DDR	Double Data Rate
DLL	Data Link Layer
DLLP	Data Link Laye Packet
DMA	Direct Memory Access
DUT	Device Under Test
FIFO	First In First Out
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GLS	Generator Loopback Switch
IP	Intellectual Property
ISO	International Organization for Standardization
MFB	Multi-Frame Bus
MI	Memory Interface
MPS	Maximum Payload Size
MVB	Multi Value Bus
NACK	Negative Acknowledgement
NDK	Network Development Kit
NDP	Netcope Data Plane
NFB	Netcope FPGA Board
OFM	Open FPGA Modules
OSI	Open Systems Interconnection
PC	Personal Computer
PCIe	Peripheral Component Interconnect Express
PL	Physical Layer
RAM	Random Access Memory
TL	Transaction Layer
TLP	Transaction Layer Packet
UVM	Universal Verification Methodology

Appendix A – Requirement Specification

ID	Name	Description
REQ_0	Packet Concatenation	Several packets behave as one. This requires correct set of SOF and EOF flags.
REQ_1	Packet Alignment	The packets within the Super-Packet should be aligned to the block.
REQ_2	Maximum Packet Size	The size of the Super-Packet should not exceed the length set by a parameter USR_PKT_SIZE_MAX.
REQ_3	Optimal Size	The average size of the Super-Packet should be around set value.
REQ_4	Multiple Channels	The component should be able to handle packets destined for multiple channels. The number of channels is generically set.
REQ_5	Channel Coherence	The Super-Packet is made of packets with the same Channel ID.
REQ_6	Channel Order	The packets from the same channel should be in the same order in which they were sent to the DUT. Overtake is not permitted.
REQ_7	Timeout	The component should be able to handle different durations of timeout.
REQ_8	Small Packet Test	The component should handle stress tests when only small packets are sent to the DUT. This test confirms that the internal counters have correct length.
REQ_9	Big Packet Test	The component should handle stress tests when only big packets are sent to the DUT.
REQ_10	Random Size	The component should handle the combination of small and big packets.
REQ_11	One Region	The component should handle previous tests (REQ_0 – REQ_10) with configuration MFB#(1,8,8,8)
REQ_12	Four Regions	The component should handle previous tests (REQ_0 – REQ_10) with configuration MFB#(4,8,8,8)

Appendix B – Structure of the FP Directory

```
├── frame_packer/
│   ├── Modules.tcl
│   ├── synth/
│   │   └── Makefile
│   ├── uvm/
│   │   ├── tbench/
│   │   │   ├── env/
│   │   │   │   ├── env.sv
│   │   │   │   ├── generator.sv
│   │   │   │   ├── meter.sv
│   │   │   │   ├── model.sv
│   │   │   │   ├── pkg.sv
│   │   │   │   ├── scoreboard.sv
│   │   │   │   ├── scoreboard_cmp.sv
│   │   │   │   ├── sequence_tb.sv
│   │   │   │   └── sequencer.sv
│   │   │   ├── info/
│   │   │   │   ├── agent.sv
│   │   │   │   ├── config.sv
│   │   │   │   ├── monitor.sv
│   │   │   │   ├── pkg.sv
│   │   │   │   ├── sequence.sv
│   │   │   │   ├── sequence_item.sv
│   │   │   │   └── sequencer.sv
│   │   │   ├── tests/
│   │   │   │   ├── pkg.sv
│   │   │   │   ├── sequence.sv
│   │   │   │   ├── speed.sv
│   │   │   │   └── test.sv
│   │   │   ├── dut.sv
│   │   │   ├── property.sv
│   │   │   └── testbench.sv
│   │   ├── Modules.tcl
│   │   ├── signals.fdo
│   │   ├── signals_sig.fdo
│   │   ├── top_level.fdo
│   │   └── ver_settings.py
│   ├── fp_auxiliary_gen.vhd
│   ├── fp_block_vld.vhd
│   ├── fp_bs_calc.vhd
│   ├── fp_bs_ctrl.vhd
│   ├── fp_bs_per_packet.vhd
│   ├── fp_channel.vhd
│   ├── fp_channel_demux.vhd
│   ├── fp_data_sel.vhd
│   ├── fp_dropper.vhd
│   ├── fp_fifo_ctrl.vhd
│   ├── fp_merger.vhd
│   ├── fp_meta_concatenate.vhd
│   ├── fp_mux_ctrl.vhd
│   ├── fp_ptr_ctrl.vhd
│   ├── fp_spkt_lng.vhd
│   ├── fp_timeout_ext.vhd
│   ├── fp_tmp_reg.vhd
│   ├── fp_ver_module.vhd
│   └── frame_packer.vhd
```