Ethernet DMA Datapath Performance Optimization for 5G Radios

SARANYA BALATHANDAPANI & SUNIL NANJIANI
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Ethernet DMA Datapath Performance Optimization for 5G Radios

Saranya Balathandapani
saranya.balathandapani.4464@student.lu.se
Sunil Nanjiani
sunil.nanjiani.7228@student.lu.se

Department of Electrical and Information Technology
Lund University

Academic Supervisor: Steffen Malkowsky

Supervisor: Aravind Annavaram (Ericsson)

Examiner: Erik Larsson

June 24, 2021

Printed in Sweden
Tryckeriet i E-huset, Lund

Acknowledgements

We would like to thank our supervisor Aravind Annavaram (ASIC Architect, Ericsson) for his constant support, encouragement and guidance during this thesis project. We thank our academic supervisor Steffen Malkowsky (Postdoctoral Fellow, Lund University) for his guidance, supervision and valuable feedback during the project.

We are grateful to our manager Fredrik Angsmark (Manager, Ericsson) for giving us the opportunity to work on this thesis project at Ericsson in his team. He has always been there to support us and tracking the progress of the project. We would also like to thank JayaKrishna Gundala (Ericsson) for his guidance during the thesis, which helped us gain more knowledge about industrial designs.

Finally, we express our gratitude to our families for their support in our degree at Lund University.

Abstract

Direct Memory Access (DMA) is a feature of computer systems that allows hardware subsystems to access the main memory of the system independent of the Central Processing Unit (CPU). With the rise of big data transfers from/to different I/O devices, the use of DMA controllers has increased significantly. The work of DMA is not limited to only offloading processor data transfer tasks, but it can transfer data at much higher rates than processor reads and writes. Scatter-Gather DMA further enhances this technique by providing data transfers from one non-contiguous block of memory to another by means of a series of smaller contiguous-block transfers unlike normal DMA.

This thesis project explores Ericsson's Ethernet DMA, which is used in the 5G radios for high speed Ethernet data transfer. The ASIC hardware design was synthesized and programmed on Intel's Agilex development board. A test case has been written to measure the performance of Ethernet DMA's datapath. The test case was run first to check the functionality of the design in a loop-back scenario. A packet generator module was integrated to generate ethernet packets in the Ethernet DMA and the packets were sent through the datapath to be written to the memory. Besides, ethernet packets were read from the memory and transmitted from memory-mapped to streaming path. The performance of Ethernet DMA datapath was measured for both streaming to memory-mapped and memory-mapped to streaming paths. To get more reliable results, performance was measured directly from the design hardware using oscilloscope. The obtained results are analyzed and some suggestions are proposed to optimize the performance of Ethernet DMA.

Popular Science Summary

With the advent of 5G wireless communication, the data rate will increase potentially (Gbps order), with low latency, and better quality of service (QoS) to the users compared to previous generation cellular networks. CPU performances and storage capacity has also increased with the miniaturization of the devices and technological advancement. We are transferring more and more data in the system, which is taking a big percentage of CPU usage. In modern processors, more energy is consumed in moving data than on computational tasks.

To lessen the workload of the CPU, some tasks can be performed with the help of supporting controllers. For example, we are using more and more Graphic Processing Units (GPUs) for the graphics processing tasks, and Google launched Tensor Processing Unit (TPU) in 2016 for the dedicated computations involving machine learning applications. Similarly, to transfer the data between the peripherals and the memory, Direct Memory Access Controller (DMAC) was introduced to offload the CPU from data transfer tasks. DMAC helps the system by doing the dedicated task of the data transfer so that the CPU can focus on other tasks.

Ethernet is a wired computer networking technology which is commonly used in local area networks (LAN), metropolitan area networks (MAN) and wide area networks (WAN). Owing to its adaptability to new requirements, Ethernet succeeded to become dominant LAN technology and its data transfer rate has increased significantly over the years, starting from 2.94 Mbps to 400 Gbps.

The advancement in chip design has integrated the system with the network and all the peripherals on a single chip. More and more dedicated hardware is being developed for application specific tasks to increase the performance of the modern systems. Ethernet DMA controller is a dedicated hardware used to transfer Ethernet data packets with much higher speed than a normal processor.

This thesis work studies Ericsson's Ethernet DMA controller and measures its performance. The Ethernet DMA uses scatter-gather direct memory access (SGDMA) technique which allows data transfer of data that can be written to non-contiguous areas of memory. With this mechanism, it can perform transaction using buffer descriptors which can be placed in memory. As a result, high performance is achieved

using scatter-gather DMA. A test case written in C language is used to measure the performance of Ethernet DMA's datatapath by sending 10,000 ethernet packets of different packet sizes. The measurements are taken from real hardware using oscilloscope and the results presented show that the Ethernet DMA is configured to work at optimal performance. Based on the results obtained for the performance of Ethernet DMA controller, improvements have been suggested to further increase its performance.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ARCNET Attached Resource Computer Network
ASIC Application Specific Integrated Circuit
AXI Advanced eXtensible Interface
BBM Baseband Module
CCU Cache Coherency Unit
CPU Central Processing Unit
DDR Double Data Rate
DMA Direct Memory Access
DMAC Direct Memory Access Controller
DSP Digital Signal Processor
EOP End of Packet
EPG Ethernet Packet Generator
FDDI Fibre Distributed Data Interface
FIFO First In First Out
FPGA Field Programmable Gate Array
FSM Finite State Machine
Gbps Gigabits Per Second
GDMA Graph DMA
GPU Graphic Processing Units
HPS Hard Processor System
I/O Input/Output
IPG Inter Packet Gap
LAN Local Area Network
MAN Metropolitan Area Networks
Mbps Megabits Per Second
MTU Maximum Transmission Unit
MM2S Memory-mapped to Streaming
MPSoCs Multiprocessor System on Chip
QoS Quality of Service
RX DMA Receiver DMA
S2MM Streaming to Memory-mapped
SGDMA Scatter Gather Direct Memory Access
SFD Start Frame Delimiter
SoC System on Chip

SOP Start of Packet
TCI Tag Control Information
TPID Tag Protocol Identifier
TPU Tensor Processing Unit
TRX Transceiver
TX DMA Transmitter DMA
VLAN Virtual LAN
WAN Wide Area Network

Chapter 1
Introduction

The evolution of cellular networks over the last few decades have made significant improvements in data rates, latency and quality of service for the users. At present, the work on the development for the 5G has paved the way for very high data rates i.e. it offers data rates in orders of Gbps [1], [2]. To handle the high data rate, we need hardware infrastructure to efficiently transfer data without burdening the main processor with it. Ethernet is an IEEE standard for communication between different electronic devices connected through the internet or local area network. It is one of the oldest, but still the largest form of technology being used for networking and it has proven itself as a very popular, fast and relatively inexpensive LAN technology [3]. The data transfer using Ethernet happens frequently in the processor subsystem for communication. Direct Memory Access is the hardware mechanism that allows peripheral components to transfer their I/O (input/output) data directly to and from main memory without the need to involve the system processor. Direct memory access mechanism offloads the processor by taking the bus control between memory and I/O, thereby, increasing the throughput of a system [4].

This thesis project focuses on a DMA engine and try to measure the performance of existing Ethernet DMA controller's datapath at Ericsson, which is used in company's 5G radio systems.

1.1 Background & Motivation

With the huge growth in ASIC technology, memory has been playing a crucial role in the performance of SoC (System on Chip) design systems. Data is written to and read from memory quite frequently from different peripherals of the computer system. The processor of a computer usually performs a wide range of tasks and it is occupied with some task at hand all the time. In order to speedup and improve the performance of overall system, it is better to use dedicated hardware for different tasks. Data transfer is one such task which happens quite often in a computer system. If the processor dedicates itself to data transfer tasks, it would not get enough time to perform other tasks. As a result the performance may get affected, and the system would be slow. In recent years, the use of DMA controllers (DMAC) have played an important role in the performance of multipro-

cessor system-on-chip (MPSoCs) designs that require a transfer of large data sets. The main job of a DMA controller is to offload the processor from data movement tasks. When the processor needs data, it issues command to the DMA controller by specifying the source address, the destination address and the amount of data, then the DMA controller takes charge of the transfer. After handing over the necessary information to the DMAC, the processor is free to do other tasks while the DMAC transfers the data. Thus, a DMA controller improves the performance of a system by taking control of the data transfer tasks.

Ethernet is a wired computer networking technology which is commonly used in local area networks, metropolitan area networks and wide area networks. It is a dominant LAN technology and its data transfer rates have increased significantly with the evolution of cellular networks, i.e. in orders of Gbps. To process data at such a high speed, dedicated DMA controllers need to be designed to handle it and write to the memory. Different types of DMA controllers connect DSPs (Digital Signal Processors) to a DDR (Double Data Rate) memory controller, a major hurdle is the rate at which DMA blocks transfer Ethernet packets between DSPs and DDR memory. This thesis project tries to look into this problem for Ericsson's Ethernet DMA controller and its datapath, which is a part of the company's ASIC targeting 5G Mid-band and High-band radio products.

The main motivation behind this thesis project is to understand the full architecture of Ethernet DMA and work to improve performance of the datapath of the system. We would analyze the results of different performance parameters and would propose improvements to enhance the performance of Ethernet DMA controller. This thesis will pave a way to improve the performance of the DMA controller and increase the rate of ethernet packet transmission to DDR memory by analyzing the DMA path and applying different solutions. This thesis would briefly describe the challenges that occurred and their possible solutions along with results that can show how our proposed improvements can help.

1.2 Previous Work

Different designs of DMA controllers have been proposed in the literature and they have been tested on different peripherals for the performance measurements. Some relevant previous works will be introduced below.

A design of DMA controller for SoC based embedded systems[5] which increases the data transfer rate compared to conventional DMA methods. Asynchronous FIFO (First In First Out) is used for buffering the data and an FSM (Finite State Machine) is implemented based on the FIFO signals. This approach achieves a maximum frequency of 306.24 MHz and it gives better performance than polling method. A DMA controller[6] using AMBA (Advanced Microcontroller Bus Architecture) is proposed, which can be used in SoC based design. The proposed design supports memory to memory, memory to peripherals and peripheral to memory

transfer and it achieves a maximum frequency of 476 MHz and it data transfer rate of 1904 MB/s. This work has significant improvements in data transfer rate compared with embedded processors used in SoC, which we can take as an insight to get some improvements with our ethernet DMA where we are using AXI (Advanced eXtensible Interface) interconnect. An optimized PCIe DMA control flow[7] proposes timing optimizations by using FIFO cascade connection method. It uses cascading FIFOs with pipeline to replace a large FIFO to improve timing without errors. FIFO can be a bottleneck to our system which needs improvement. The DMA engine[8] uses a ring buffer to write data and adopts a handshake sequence between FPGA and Linux driver to avoid data loss and throughput reduction. PCIe Avalon MMDMA bridge has been used with Avalon FIFO memory and tri-speed ethernet which can be used in high throughput ethernet applications[9]. These solutions could avoid data loss and achieve high throughput, which are also concerns in datapath optimization in our project. A hardware controlled graph DMA (G-DMA) based on scatter-gather DMA engine[10] fetches new data based on memory request. It decreases the processing time greatly over using standard CPU and standard AXI memory requests.

This thesis will explore Ericsson's ASIC design which uses Ethernet DMA whose performance has not been measured before. The above mentioned previous works make a good ground for the motivation to measure the performance of Ethernet DMA's datapath and propose improvements to optimize it.

1.3 Thesis Outline

This thesis attempts to analyze and measure the performance of an existing Ethernet DMA's datapath. Based on the results, some solutions to improve the performance are proposed. The data rate improvement of ethernet packets transferred between baseband module (BBM) and memory can improve the performance of 5G radios. The rest of the thesis is organized in the following way.

- Chapter 2: Background and theory
- Chapter 3: Hardware Design and Test Case
- Chapter 4: Implementation
- Chapter 5: Results
- Chapter 6: Improvements & Future Work
- Chapter 7: Conclusion

Chapter 2
Background & Theory

A computer consists of a CPU, memory, I/O interfaces and peripherals. A CPU usually performs two kinds of tasks, computations and communication. Memory in a computer is used to store data and programs. I/O interface helps in transferring data between peripherals and the CPU.

2.1 Data Transfer Methods

In a computer system, data transfer is one of the main tasks which takes a good amount of computer resources and time. Different peripherals attached to the system need to access the data in the memory, that is, either it is read from the memory or it is written to the peripherals attached. In all cases, the CPU needs to allocate the resources for these tasks to be completed. Some well known approaches of data transfer are discussed below.

2.1.1 Polling Based Transfer

In the polling based transfer method, CPU continuously checks the peripheral status whether it is ready to transmit or receive data or it is busy. Figure 2.1 presents a system with three peripherals. The CPU will check each peripheral repeatedly in loops if it needs to be served. If there is a data at Peripheral 1, it will serve it until it finishes the task. Peripheral 2 and peripheral 3 would be waiting while the CPU serves peripheral 1. Next it will check peripheral 2 if there is any service request. In polling method, CPU wastes a lot of time monitoring peripherals to check if there is any service request. This method transfers data at a high rate, but it has a major drawback, that is, processor cannot perform any other activity during the data transfer.

2.1.2 Interrupt Based Transfer

The second method of data transfer is interrupt based data transfer mode, which is shown in Figure 2.2. In this data transfer mode, whenever the device is ready for data transfer, it raises interrupt to processor. Once the interrupt for the data transfer is raised (CPU sends the Interrupt Ack signal to the Interrupt Controller), the processor completes its ongoing instruction and saves its current status. Then

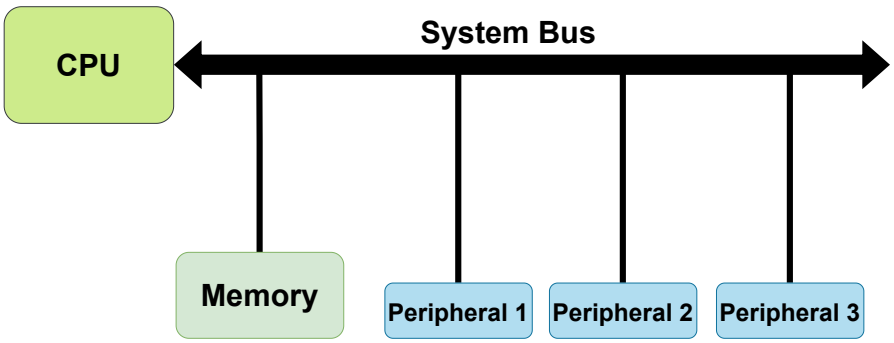

Figure 2.1: Polling Based Data Transfer

it switches to the data transfer mode to serve the interrupt, thus, starts data transfer with a delay. In this method, the processor does not keep scanning for peripherals for data transfer, but it is fully involved in the data transfer process. Therefore, this method is also not an effective way of data transfer.

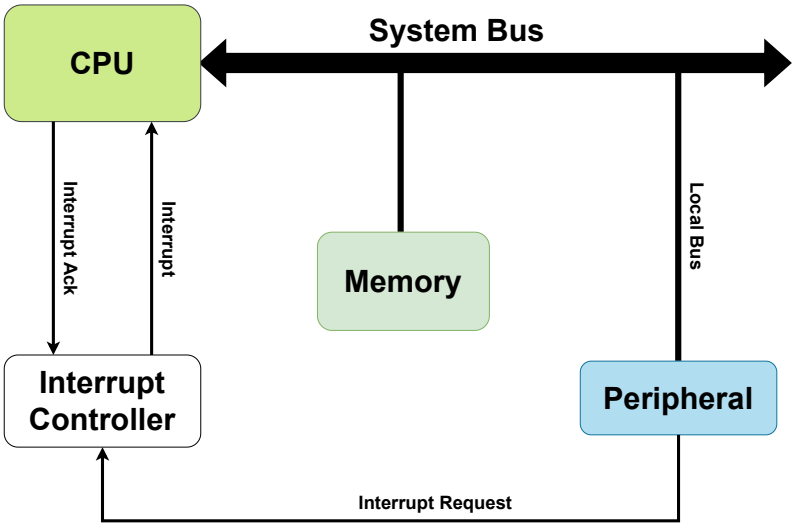

Figure 2.2: Interrupt Based Data Transfer

2.1.3 DMA Based Transfer

Both polling and interrupt based data transfer methods involve CPU directly for the data transfer. To offload the CPU from the data transfer tasks, direct memory access mechanism is used, which helps in data transfer directly from peripherals to the memory and vice versa. A DMA controller as shown in the Figure 2.3 is used to transfer data between peripherals and memory. A more detailed overview

of DMA controller is given in the following section.

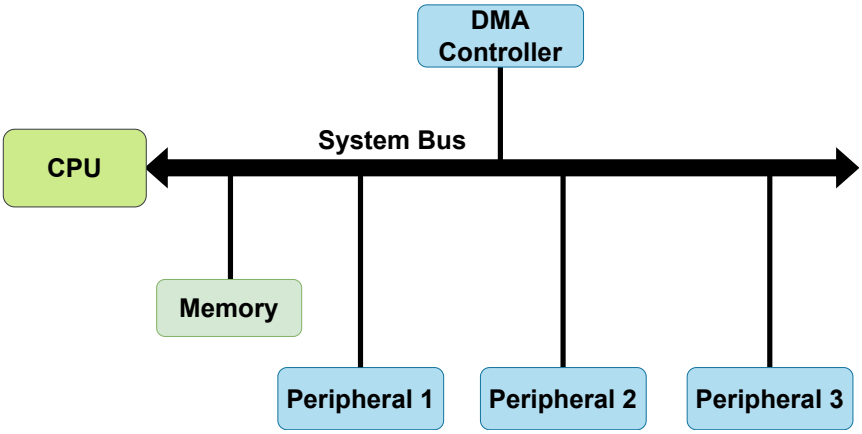

Figure 2.3: DMA Based Data Transfer

2.2 DMA Controller

The term DMA stands for Direct Memory Access. It is a hardware mechanism that allows peripheral components to transfer their I/O data directly to and from main memory without the need to involve the system processor. The hardware device used for direct memory access is called the DMA controller which allows I/O devices to directly access the memory with minimal participation of the processor. The job of a DMA controller is to transfer/copy data from one memory location to another without involving the processor. When the processor needs data, it issues command to the DMA controller by specifying the source address, the destination address and the amount of data, then the DMAC takes charge of the transfer. After handing over the necessary information to the DMAC, the processor is free to do other tasks while the DMAC transfers the data. Even though the DMA controller transfers data without the involvement of processor, it is controlled by processor. A DMA controller provides an interface between peripherals and the system bus. More than one peripherals can be connected to a DMA controller.

A DMA controller can transfer the data in three modes:

Burst Mode: In this mode, once the DMA controller gets the charge of the system bus, it releases the system bus only after completion of data transfer. The CPU needs to wait for the system bus until DMA controller releases it.

Cycle Stealing Mode: DMA controller forces the CPU to halt its operation and abandon the control over the system bus for a short duration to give it to DMA controller. In this mode, DMA controller steals the clock cycle for transferring every byte, that is why it is called cycle stealing mode. DMA controller

releases the bus after every byte transfer and requests it again for the next transfer.

Transparent Mode: In the transparent mode, the DMA controller takes charge of system bus only when the processor does not need it.

2.2.1 Working of DMA Controller

A DMA controller acts as an interface between the system bus and the peripheral. It transfers data without the intervention of the main processor, but it is controlled by the processor. The main processor initiates the operation of the DMA controller by sending the addresses, the size of data to be transferred and the direction of transfer i.e. from peripheral to memory or from memory to peripheral. A DMA controller can have more than one peripherals connected.

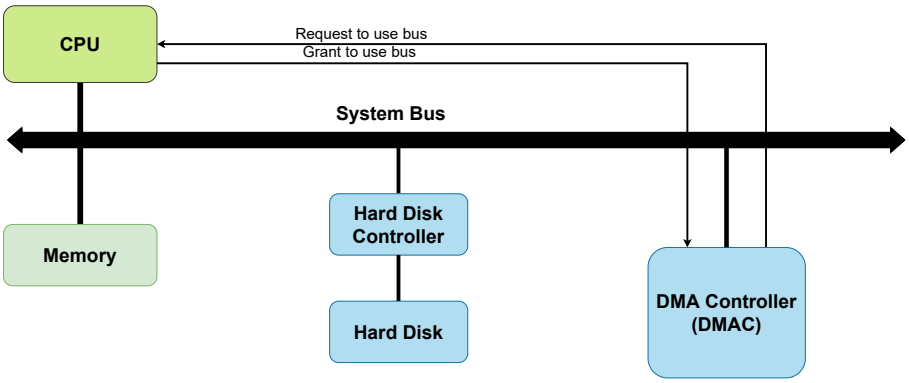

Figure 2.4: Direct Memory Access Controller

DMA controller contains an address unit, for generating addresses and selecting I/O device for transfer. It also contains the control unit and data count for keeping counts of the number of blocks transferred and indicating the direction of transfer of data. When the transfer is completed, DMA informs the processor by raising an interrupt. DMA controller has to share the bus with the processor to make the data transfer. The device that holds the bus at a given time is called bus master. When a transfer from I/O device to the memory or vice versa has to be made, the processor stops the execution of the current program, increments the program counter, moves data over stack then sends a DMA select signal to DMA controller over the address bus. If the DMA controller is free, it requests the control of the bus from the processor by raising the bus request signal. Processor grants the bus to the controller by raising the bus grant signal, now the DMA controller is the bus master. The processor initiates the DMA controller by sending the memory addresses, number of blocks of data to be transferred and direction of data transfer. After assigning the data transfer task to the DMA controller, instead of waiting ideally till completion of data transfer, the processor resumes the execution of the program after retrieving instructions from the stack.

A DMAC contains several registers like address register, byte count register, and control register. The address register specifies the start address of memory. A byte count register is used which specifies the number of bytes to be transferred. The control registers are used to specify the I/O port and the direction of the transfer (reading from the I/O device or writing to the I/O device). The CPU can write data to these registers. Once the DMAC is initialized, it sends a signal to the CPU to request the memory bus. The CPU acknowledges the request by sending a grant signal to DMAC and releasing the bus (floating the output signals). Then the bus control is taken by DMA controller for data transfer. Bus is released back to the CPU by the DMA controller once the specified amount of data is transferred [11].

2.2.2 Scatter Gather DMA

Scatter-Gather Direct Memory Access (SGDMA) is a DMA data transfer technique which allows the transfer of data from/to multiple memory areas in a single DMA transaction. This mechanism is equivalent to having multiple simple DMA requests together. Thus, it can off-load multiple input/output interrupt and data transfer tasks from the CPU. Scatter/gather is supported through the use of linked lists, which means that the source and destination areas do not have to occupy contiguous areas in memory. Therefore, scatter/gather is used to do DMA data transfers of data that is written to non-contiguous areas of memory [12]. With the scatter gather mode, a DMA can perform transaction and management using buffer descriptors which can be placed in any storage like DDR. As a result high performance can be achieved by using the DMA in scatter gather mode [13]. SGDMA controller core implements high-speed data transfer between two components and it can be used in the following modes to transfer data from:

- Memory to memory
- Data stream to memory
- Memory to data stream

The SGDMA controller core transfers and merges non-contiguous memory to a continuous address space, and vice versa [14]. The core reads a series of descriptors that specify the data to be transferred. SGDMA controller has its own series of descriptors specifying the data transfers. Let's take an example of descriptor used by Xilinx's AXI DMA IP when it is configured for scatter-gather mode. The descriptor is made up of eight 32-bit base words and 0 or 5 User Application words as shown in Figure 2.5. Buffer address field gives the address of data which needs to be processed. The descriptor has future support for 64-bit addresses and support for user application data. These descriptors can be described for data transfer in streaming to memory direction and vice versa.

Address Offset	Descriptor Field
0x0	Next Descriptor Pointer
0x4	Upper 32 bits of Next Descriptor Pointer
0x8	Buffer Address
0xC	Upper 32 bits of Buffer Address
0x10	Reserved
0x14	Reserved
0x18	Control
0x1C	Status
0x20	APP0
0x24	APP1
0x28	APP2
0x2C	APP3
0x30	APP4

Figure 2.5: Descriptor Example

2.3 Ethernet Protocol

Ethernet is a wired computer networking technology which is commonly used in LAN, MAN and WAN networks. Ethernet was first commercially introduced in 1980 when Xerox, Intel and DEC together published a modified Ethernet version, which came to be known as DIX Ethernet II version, and it was standardized in 1983 as IEEE 802.3. Ever since it has been standardized and commercially available, Ethernet has evolved to support higher bit rates, backward compatibility, longer link distances and higher number of nodes. Owing to its adaptability to new requirements, Ethernet succeeded to become dominant LAN technology and replaced competing wired LAN technologies such as Token Ring, fibre distributed data interface (FDDI) and attached resource computer network (ARCNET) [15] . Ethernet data transfer rate has increased significantly over the years, starting from 2.94 Mbps to 400 Gbps. In the following section, Ethernet frame will be discussed.

2.3.1 Ethernet Frame

Ethernet is an IEEE standard which works in the data link layer and physical layer of the OSI model. An Ethernet frame is a data link layer protocol data unit and it uses the underlying Ethernet physical layer transport mechanisms. In other words, a data unit on an Ethernet link transports an Ethernet frame as its payload. IEEE 802.3 defines different Ethernet frame formats and this section gives an overview of most common Ethernet frame formats.
An Ethernet frame consists of 3 major fields; namely, header, payload and frame check sequence (FCS). Each Ethernet frame starts with an Ethernet header, which

contains destination MAC address and source MAC address as its first two fields. The second part of the Ethernet frame contains payload data, and the last part of the frame is called frame check sequence which is a 32-bit cyclic redundancy check used to detect any in-transit corruption of data.

2.3.2 Ethernet Frame Structures

IEEE 802.3 specifies the internal structure of an Ethernet frame. In today's LANs, the most dominant Ethernet version is Ethernet II with the type field indicating the payload type as shown in the Figure 2.6. An Ethernet packet starts with a seven-octet preamble and one-octet start frame delimiter (SFD). Then it is followed by destination MAC address and Source MAC address fields. Depending on the type of the Ethernet frame, some bytes are used to specify the type before the start of the payload. The type of Ethernet is defined by 4 bytes field in which the first 2 bytes represent the tag protocol identifier (TPID) and the second 2 bytes indicate tag control information (TCI) field. The length of the payload is indicated by 2 bytes.
For different types and tags, there are different Ethernet frame formats, which are discussed below. Ethernet II packet size varies based on tagging. There are 3 types of tagging; namely, Untagged, Single VLAN and double VLAN.

Figure 2.6: Ethernet II Frame Structure

Virtual LAN (VLAN) Tagging – In this tagging structure, also known as Single VLAN, 2 bytes are used for each TPID and TCI fields. The frame format for Single VLAN is shown in the Table 2.1. The size of header is 18 bytes which is fixed for single VLAN. The minimum packet size is 18 bytes.

Table 2.1: VLAN Ethernet Frame Structure

Destination MAC	Source MAC	TPID	TCI	Payload Size	Payload
6 bytes	6 bytes	2 Bytes	2 Bytes	2 Bytes	0-1500 Bytes

Let's take an example ethernet frame with fields defined as below:
Destination MAC Address: 0xAA0475C82865
Source MAC Address: 0xAA0102FA70AA
TPID: 0x8100
TCI: 0x0000A
Payload Size: 0x0004
Payload: 0xFFFFFFFF
Using these fields values, an ethernet frame can be formed as shown in Figure 2.7.

This frame uses 4 bytes for tag fields and 2 bytes for payload size field, thus, it is a Single VLAN frame structure.

Destination MAC Address	Source MAC Address	Ethernet Type (Tag Fields - Payload Size)			Payload
0xAA0475C82865	0xAA0102FA70AA	0x8100	0x000A	0x0004	0xFFFFFFFF

Figure 2.7: Ethernet Frame Example

Double VLAN Tagging – In the double VLAN tagging, 4 more bytes are added to the Ethernet type field for TPID and TCI. The frame size is 4 bytes greater than single VLAN structure as shown in the Table 2.2. The minimum packet size is 22 bytes.

Table 2.2: DVLAN Ethernet Frame Structure

Destination MAC	Source MAC	TPID + TCI	TPID	TCI	Payload Size	Payload
6 Bytes	6 Bytes	4 Bytes	2 Bytes	2 Bytes	2 Bytes	0-1500 Bytes

Untagged VLAN – In this frame format, there is no tagging used and the structure is shown in the table 2.3. The minimum packet size is 14 bytes which includes source and destination addresses and payload size.

Table 2.3: Untagged VLAN Ethernet Frame Structure

Destination MAC	Source MAC	Payload Size	Payload
6 Bytes	6 Bytes	2 Bytes	0-1500 Bytes

Jumbo Frame – Some variants of Ethernet support larger frames which are known as jumbo frames. The payload size for the jumbo frames is greater than the maximum transmission unit (MTU), an example frame structure of jumbo frame is shown in the table 2.4.

Table 2.4: Jumbo Frame Structure

Destination MAC	Source MAC	TPID	TCI	Payload Size (>1500)	Payload
6 Bytes	6 Bytes	2 Bytes	2 Bytes	2 Bytes	>1500 Bytes

Chapter 3
Hardware Design & Test Case

In this chapter, the hardware design of the project and initial test case will be discussed. In the second chapter, an overview of the background knowledge of the subject was provided and using the basics from it, this chapter discusses design setup and the test case.

The main goal of this thesis project is to measure the performance of Ethernet DMA which is a part of Ericsson's ASIC design, that targets 5G Mid-band and High-band radio products. As a part of testing the design, Intel's Agilex FPGA Dev-Kit has been used to measure the performance and a test case was developed in C programming language. The following sections go through the design specifications and test case used to achieve the results.

3.1 Hardware Design of Ethernet DMA Datapath

In the first step of the project, the Ethernet DMA design was synthesized using Intel Quartus Prime Tool. Then, the FPGA image was loaded onto the FPGA Dev-Kit. Figure 3.1 presents an overview of the system which shows the full datapath from DDR memory to BBM devices and vice versa. The design consists of a Hard Processor Subsystem (HPS) which contains CPU, Cache Coherency Unit (CCU), DDR Controller and interconnects. The design implements two separate Ethernet links at 10.3125 Gbps towards each BBM. One of the links is used for standard Ethernet packages, which connects to DDR controller through a subsystem of two scatter-gather DMA controllers, known as Ethernet DMA. There are two Ethernet DMA controllers in the system, one for each BBM device. The other link is intended for BBM trace data, which connects to the DDR through Trace DMA block. Trace DMA block is a simple Ethernet packet inspector and DMA controller, connecting a BBM device to the DDR controller of the processor subsystem. Similar to Ethernet DMA, there are two instances of Trace DMA, each connecting to one of the two BBM devices. As shown in the datapath, the Ethernet data follows from DDR memory to BBM devices via interconnects, Ethernet DMA and transceiver (TRX) modules, and vice versa. The system design has been created integrating the third party design IPs. The datapath from BBM devices to DDR memory is called Streaming to Memory-mapped (S2MM) path. The datapath from DDR memory to BBM devices is called Memory-mapped to

Streaming (MM2S) path. In the next section, more detailed view of Ethernet DMA is given.

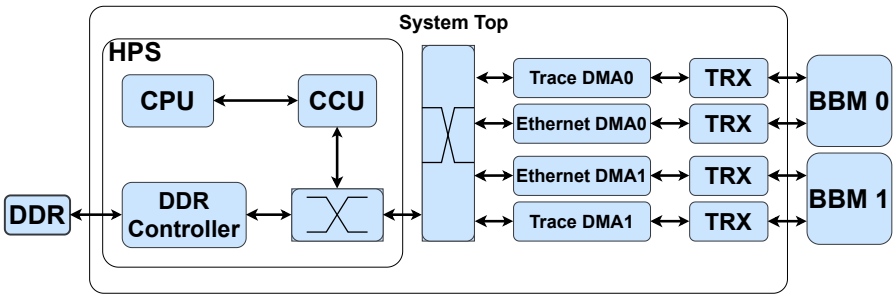

Figure 3.1: System Overview

3.1.1 Ethernet DMA

The Ethernet DMA controller offloads the HPS subsystem from the data transfer tasks. Each Ethernet DMA subsystem consists of two instances of DMA cores. The top level view of Ethernet DMA is presented in Figure 3.2. Ethernet DMA consists of two DMA cores; one instance is for receiving data streams, that is, streaming data from BBM to memory mapped data to DDR; the other instance is for transmitting data streams, that is, memory mapped data from DDR to streaming data to BBM. The Ethernet DMA works at a frequency of 156.25 MHz. The description of main components of Ethernet DMA is given below.

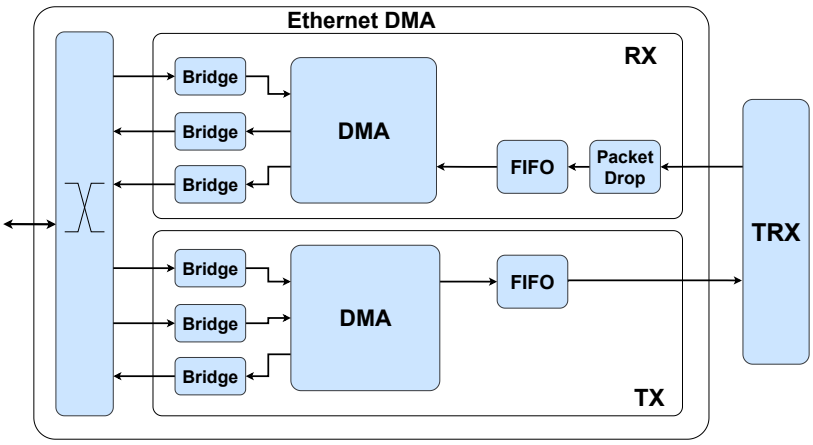

Figure 3.2: Ethernet DMA Datapath

FIFO – FIFO register is used to hold the packet until the previous packet is getting processed by the DMA. FIFO stores a complete ethernet packet and it has a

size of 2 KB.

Packet Drop Block – Packet Drop block in the Ethernet DMA uses a mechanism to drop ethernet packets depending on the following three reasons.

1. Based on the FIFO availability, either packets are transferred to FIFO or dropped. When there is no sufficient space in the FIFO, then the incoming packet is dropped.
2. A packet is dropped if there is an error in the packet, i.e. packet is corrupt or not in proper format.
3. Transmitter Ready error, i.e. problem at the transmitter side.

Bridges – Bridges and inter-connectors are used to convert AXI to streaming and streaming to AXI. Each port from Ethernet DMA to DDR controller is connected through individual bridge. These bridges can be individually run-time configured by SW to decide if the data stream over that specific interface shall be cache coherent, i.e. routed through the CCU unit of HPS or non-coherent, i.e. routed directly to/from the DDR controller. Each bridge contains one SW control register. Bridges are also used to improve the performance through pipelining.

DMA – Ethernet DMA employes two SGDMA cores, one is used to receive the data and the other is used to transmit the data.

3.1.2 DMA Core

Ethernet DMA consists of two SGDMA cores, one is used for streaming to memory mapped data transfer (Figure 3.3) and the other is used for memory mapped to streaming data transfer (Figure 3.4). Each SGDMA core consists of three main blocks; namely, Prefetcher, Dispatcher and Read/Write Master. The Prefetcher fetches a series of descriptors from memory before passing them to the Dispatcher. The series of descriptors in memory can be linked together to form a descriptor list. This allows the SGDMA to execute multiple descriptors in a single run, thus enabling transfers to/from non-contiguous memory spaces and thereby improving system performance. Each descriptor specifies a data transfer to be performed. The dispatcher receives and decodes the descriptors and dispatches instructions to the Read Master or Write Master blocks for further operation. Each time a descriptor has been processed, number of bytes transferred, status and control fields of the descriptor are updated and written back to memory. The details of DMA core blocks is discussed below.

Receiver DMA – Receiver DMA (RX DMA) is used to receive data as input from the streaming side and send it to the memory, hence, it is used in streaming to memory mapped datapath. The block diagram of RX DMA core is shown in the Figure 3.3, which consists of three blocks; Prefetcher, Dispatcher and Write Master block. Prefetcher reads the descriptors from the memory and then they are sent to Dispatcher block which decodes them and gives write command to the Write Master block. As the new data (Data In) comes in to Write Master block,

it writes data to memory and sends write response signal to the Dispatcher block once the data is written to the memory.

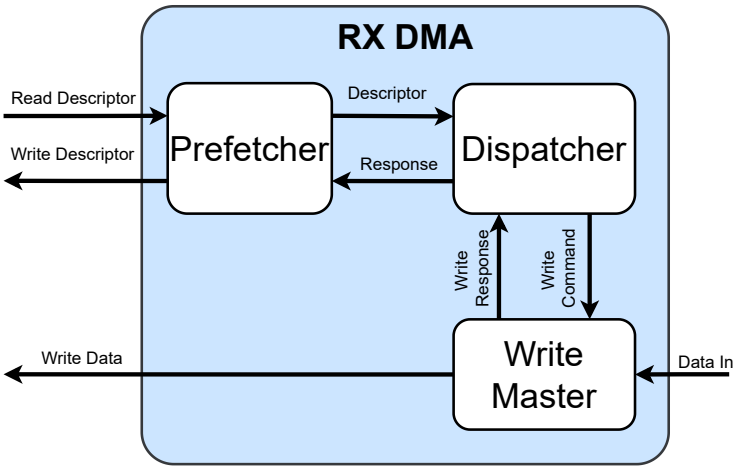

Figure 3.3: RX DMA Core

Transmitter DMA – Transmitter DMA (TX DMA) is used to transmit data from the memory to streaming side, i.e. it is used to read ethernet data packets from memory and send them out to transmit. Figure 3.4 shows the block diagram of TX DMA core and it consists of three blocks; Prefetcher, Dispatcher and Read Master. Prefetcher reads the descriptors from the memory and then they are sent to Dispatcher block which decodes them and gives read command to the Read Master block. As the new data (Read Data) comes in to Read Master block, it reads data out to transmit and sends read response signal to the Dispatcher block once the data is read from the memory.

Prefetcher – The Prefetcher block is used to fetch the descriptors from the memory and have it as a linked list before it gets processed by the next stages. With the Prefetcher block, addressing of next descriptors becomes simpler and faster as it works as a linked list.

Dispatcher – The Dispatcher receives one descriptor at a time and dispatches the instruction to the Read/Write master block. It understands the descriptor by opening it and sends the instruction to Read/Write Master modules.

Write Master – Write Master receives the instruction from the dispatcher and executes it. i.e, it receives a packet and writes it into the memory. The size of the packet and memory location where it needs to be written are specified by dispatcher's instruction. It will write the response back to the descriptor upon completion of execution of the instruction specified by the dispatcher based on the descriptor.

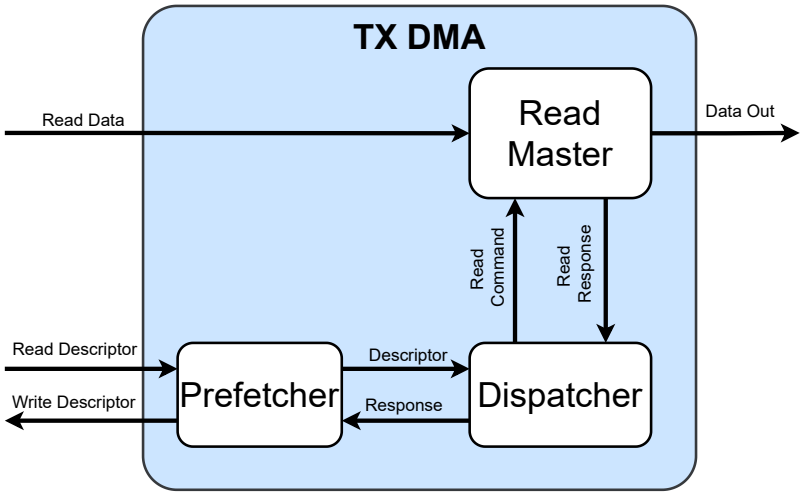

Figure 3.4: TX DMA Core

Read Master – Read Master receives the instruction from the dispatcher and executes it. i.e, it reads a packet from the memory and sends it to the FIFO. The size of the packet and memory location from where it needs to be read are specified by dispatcher's instruction. It writes the response back to the descriptor upon completion of execution of the instruction specified by the dispatcher based on the descriptor.

3.2 Test Case Structure

Once the design is synthesized and FPGA is programmed using Intel Quartus Prime tool, the next step is to test the working of the design. For this purpose, the test case has been written using C programming language. The test case verifies the working of the datapath of the design and it follows a sequence of steps to run the design on the FPGA Dev-Kit HW. The following sections talk more about the essential parts of the test case and the sequence of steps it follows.

3.2.1 Ethernet Packets

Initially, the ethernet packets were created from the software code written in the test case. The ethernet packets created used Single VLAN ethernet frame structure which was introduced in Chapter 2.

3.2.2 Descriptors

Descriptors can be considered as the instruction packages containing instructions about the data to be transferred. Descriptors are created in a way that it specifies the size and memory location of the packets where it needs to be read from or

written to. The descriptors are connected with one another using a linked list structure, which means that each descriptor has an address field which points to address of the next descriptor. Figure 3.5 gives the structure of a descriptors which consists of the following fields.

Address Offset	Descriptor Field
0x0	Read Address [31-0]
0x4	Write Address [31-0]
0x8	Length [31-0]
0xC	Next Descriptor Pointer [31-0]
0x10	Actual Bytes Transferred [31-0]
0x14	Status [15:0]
0x3C	Control Register [31-0]

Figure 3.5: Descriptor Format

Read Address – This field specifies the address of the memory location where the packets need to be read from.

Write Address – It specifies the address of the memory location from where the packets need to be written to.

Length – This field gives the size of the Ethernet packet.

Next Descriptor Pointer – This field gives the address of the next descriptor.

Actual Bytes Transferred – Specifies the actual number of bytes written to the memory and gets updated by the write master at the end of write operation. This field is applicable only for Streaming to Memory-Mapped transfer configuration.

Status – This field is used to give the status of streaming to memory-mapped data transfer. It is not applicable to memory-mapped to streaming configuration.

Control Register – Control Register is a 32 bit register which sets some useful bits and interrupts like, Start of Packet (SOP), End of Packet (EOP), Transfer complete Interrupt. A bit called 'Owned by Hardware' determines whether hardware or software has write access to the current descriptor.

The test case has been written to accommodate different configurations of the Ethernet packets and descriptors, which include:

- **Single DMA Descriptor and Single Packet:** In this configuration,

descriptors are created in such a way that each descriptor carries only one Ethernet packet.

- **Multiple DMA Descriptors and Single Packet:** In this configuration, the descriptors are created in such a way that multiple descriptors can carry one Ethernet packet. In this configuration, an ethernet packet header can be carried with one descriptor and its payload can be carried with other descriptor.
- **Multiple DMA Descriptors and Multiple Packets:** In this configuration, multiple descriptors are created to carry multiple Ethernet packets.

3.3 Loop-back Test Case Scenario

In order to measure the performance of the Ethernet DMA datapath, the first step is to verify the functionality of the design. Different approaches can be used to check the working of different modules. As introduced in Section 3.1, there are two paths in the design which need to be tested, one is from streaming to memory-mapped and the other is from memory-mapped to streaming. The best way to verify the functionality of the datapath is to perform a loop-back test, which allows one test to verify the functionality of both paths. For that purpose, a test case has been written to read and transmit the data from the memory and receive the same data from the streaming side and write it back to the memory. Thus, the loop-back test helps in verification of both datapaths.

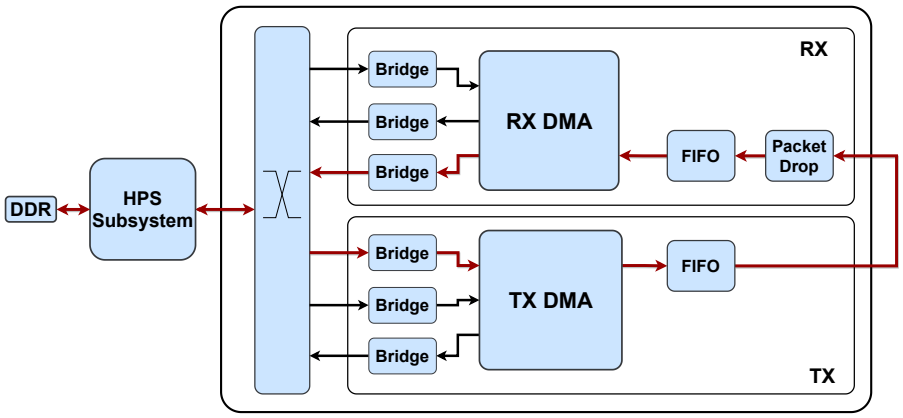

Figure 3.6: Loop-back Test Datapath

The working of Ethernet DMA datapath is confirmed with successful transmission and reception of the same packets through loop-back test. Figure 3.6 presents the block diagram of the loop-back test, where the Ethernet packets are created and stored in the memory. The packets are then transmitted from the memory to streaming path through the TX DMA. Then the packets are sent back to be written to the memory following the streaming to memory-mapped path through

the RX DMA. The datapath of the this loop-back test case scenario is highlighted in the brown in Figure 3.6. The descriptors and Ethernet packets are are created using the functions defined in the test case.

Using the test case for the loop-back, a number of different Ethernet packets and descriptors were processed. The test case follows a series of steps to configure and check certain registers to run. Here, we take an example of test case where 'Single DMA Descriptor and Single Packet' configuration has been used to follow all the important steps, which are discussed below.

Listing 3.1 shows the log output of the initial setup of hardware while running the test case. The test case starts with checking PLL locking status and release reset to initialize the modules of the design.

```
1  int_loopback = 1 num_pkts = 1 payload_size = 128 perf = 0 dma
       = 0 cbuff = 1 setuptrx = 0 fpgadownload = 0 firewall = 0
       initbbms = 0 setup_switch = 0 tdmaenable = 0 seqmemaddr = 0
        value = 3c008008 dump_ddr=0 init_ccr=1 untag_vlan_dvlan=1
       cyclic_bd_enable=0
2  IO_BASE_CCR:0xf919f000 IO_BASE_CCR+CCR_TEST_RD_OFFSET:0
       xf919f144
3
4  read_value:0x55555555
5
6  Checking writing to CCR test register
7
8  Poll PLLs B and C some time and report error if not all of
       them are locked.
9
10 For max R/W and also to check PLL reset, perform reset (should
        be set for at least 1us!) and then check for lock again
11
12 CCR_PLL_RESET_CTRL address:0xf919f15c read_value:0x333
       exp_value:0x33
13
14 ERR - CCR Max RW Value: 0x333 expected: 0x33
15 ADDING SOME OF THE OTHER ACCESSES HERE TO MAKE SURE WE GET AT
       LEAST 1us RESET ASSERTED
16 Read CCR_DEBUG_STATUS
17 address:0xf919f164  read_value:0x0  exp_value:0x22200
18
19 ERR - CCR Max RO Value: 0x0 expected: 0x22200
20 Release Reset
21 Poll for Lock again
22 INFO - Poll for PLLs B and C Lock after soft reset
23
24 After PLL are Locked, Release clock gating of B and C, writing
        0x3 and then 0x0 at address: 0xf919f16c
25 Release TRX reset, writing 0x0 at address: 0xf919f170
26 Release all blocks from reset, writing 0x3FFF and then 0x0 at
```

```
    address: 0xf919f160
27 Check PLL locked again, and IC reset control partly to pass
    time to make sure all resets are fully released at end of
    this function, reading value at 0xf919f17c
28 INFO - Read out at address:0xf919f148 LOCKSTATUS:0x7 exp_value
    :0x7
29 INFO - No switch setting provided
30 # NOTE: Test has ended with 2 error(s) and 0 warning(s) in
    test
```

Listing 3.1: Loop-back Test Initial Setup

Ethernet Packet Creation: After starting the system, the test case works with the creation of ethernet packets, which are generated using a function in the test case. The memory is cleared before populating it with the ethernet packets. The header of ethernet packet is created with Single VLAN tag structure which takes 18 bytes and a payload of 128 bytes is created. Thus, Ethernet packets of size 146 bytes are created and stored in the memory location specified through the test case. The log output of the test case in showing these steps is presented in Listing 3.2.

```
1 INFO - FUNC_DMA_ETH_Test 235 int_loopback = 1 num_pkts = 1
    payload_size = 128 dma = 0 check_buff = 1 setuptrx=0
    vlantag=1 tdmaenable=0
2
3 INFO - clear_buffer 158
4 INFO - clear_buffer 158
5 INFO - Clear TX buffer memory 4000030e  before populating it
6 INFO - Create 2 tx buffers
7  19 : set_eth_type_ii_frame_header a_frame_addr=x40000280
    payload_size=128 payload_size=x80
8  139 : set_eth_type_ii_frame_payload a_frame_addr=x4000028e
    a_size=128 a_size=x80
9 dump_buffer_contents 147 sizeof(buff)=8 buff=0x40000280##
    index=0 read_byte=0xaa :
10 ## index=1 read_byte=0x4 : ## index=2 read_byte=0x75 : ##
    index=3 read_byte=0xc8 : ## index=4 read_byte=0x28 : ##
    index=5 read_byte=0x65 : ## index=6 read_byte=0xaa : ##
    index=7 read_byte=0x1 : ## index=8 read_byte=0x2 :
11 ## index=9 read_byte=0xfa : ## index=10 read_byte=0x70 : ##
    index=11 read_byte=0xaa : ## index=12 read_byte=0x81 : ##
    index=13 read_byte=0x0 : ## index=14 read_byte=0x0 : ##
    index=15 read_byte=0xa : ## index=16 read_byte=0x0 :
12 ## index=17 read_byte=0x80 : ## index=18 read_byte=0xa : ##
    index=19 read_byte=0xa : ## index=20 read_byte=0xa : ##
    index=21 read_byte=0xa : ## index=22 read_byte=0xa : ##
    index=23 read_byte=0xa : ## index=24 read_byte=0xa :
13 ## index=25 read_byte=0xa : ## index=26 read_byte=0xa : ##
    index=27 read_byte=0xa : ## index=28 read_byte=0xa : ##
    index=29 read_byte=0xa : ## index=30 read_byte=0xa : ##
```

```
    index=31 read_byte=0xa : ## index=32 read_byte=0xa :
```

Listing 3.2: Loop-back Test Packet Creation

Descriptor Creation: Once the Ethernet packets are created, the next step is to create descriptors with information about the data to be transferred. Descriptors are created using the functions in the test case, which allow to create descriptors based on the configurations through which the packets need to be send. As an example, one descriptor created for memory-mapped to streaming transaction is shown in Listing 3.3 of log output. The descriptor address is 0x50000280. The first field in the descriptors is the read address (0x40000280) and it is the address at which Ethernet packet data starts. The second field at address 0x50000284 indicates the write address which is not applicable to memory-mapped to streaming transactions. Third field of descriptor shows the length of the ethernet frame, which is 0x92 (146 bytes). The fourth field shows the next descriptor pointer address is 0x50000980 which indicates that each descriptor has been allocated 0x100 (256 bytes). Next two fields which give actual bytes transferred and status value are are not applicable for memory mapped to streaming transaction. The last field shows control register and has a value of 0xC200D300.

```
1  INFO - Dumping MM2S and S2MM descriptors before DMA has
       started operation
2  INFO - Dumping mm2s descriptor number 0 contents at address 0
       x50000280
3  DEBUG : dump_mm2s_scatter_gather_descriptor line no 219
4  DEBUG : Address = 0x50000280 Value = 0x40000280
5  DEBUG : Address = 0x50000284 Value = 0x0
6  DEBUG : Address = 0x50000288 Value = 0x92
7  DEBUG : Address = 0x5000028c Value = 0x50000380
8  DEBUG : Address = 0x50000290 Value = 0x0
9  DEBUG : Address = 0x50000294 Value = 0x0
10 DEBUG : Address = 0x500002bc Value = 0xc200d300
11 DEBUG : dump_mm2s_scatter_gather_descriptor line no 301
       completed
```

Listing 3.3: MM2S Descriptor Creation Log Output

DMA Configuration: Once the ethernet packets and descriptors are created, the next step is to configure the Ethernet DMA. Prefetcher block is configured with descriptors' physical addresses as shown in the line 11 and 12 in log output in Listing 3.4. Dispatcher block's control register is configured by setting the bit number 4, which enables global interrupt as shown in the lines 9 and 10 in the log output. Once all the modules complete their operations in the DMA, then the prefetcher blocks are reset, dispatchers are stopped and then reset.

```
1 INFO - clear_buffer 158
2 IO_BASE_ETH_DMA0_TX_PREFETCHER_CONFIG is f91a7920
3 IO_BASE_ETH_DMA0_RX_PREFETCHER_CONFIG is f91a7960
4 IO_BASE_ETH_DMA_0_MM2S_CSR_BASE is f91a7900
5 IO_BASE_ETH_DMA_0_S2MM_CSR_BASE is f91a7940
```

```
IO_BASE_ETH_DMA0_RX_PREFETCHER_CONFIG is f91a7960
IO_BASE_ETH_DMA0_TX_PREFETCHER_CONFIG is f91a7920
start_dma_and_wait_for_done 870 s2mm_desc_phyaddr_current
    =50003800 mm2s_desc_phyaddr_current=50000280
IO_BASE_ETH_DMA_0_MM2S_CSR_BASE + 0x4=0xf91a7904 value=0x10
IO_BASE_ETH_DMA_0_S2MM_CSR_BASE + 0x4=0xf91a7944 value=0x10
IO_BASE_ETH_DMA0_TX_PREFETCHER_CONFIG + 0x4=0xf91a7924 value=0
    x50000280
IO_BASE_ETH_DMA0_RX_PREFETCHER_CONFIG + 0x4=0xf91a7964 value=0
    x50003800
IO_BASE_ETH_DMA0_RX_PREFETCHER_CONFIG=0xf91a7960 value=0x8
IO_BASE_ETH_DMA0_TX_PREFETCHER_CONFIG=0xf91a7920 value=0x8
INFO - HL1_DMA_wait_dma_done 900
INFO : MM2S IRQ at IO_BASE_ETH_DMA_0_MM2S_CSR_BASE + 0x0
    received: 200

INFO : S2MM IRQ at IO_BASE_ETH_DMA_0_S2MM_CSR_BASE + 0x0
    received: 200

INFO : MM2S IO_BASE_ETH_DMA0_TX_PREFETCHER_CONFIG + 0x10 at
    IO_BASE_ETH_DMA0_TX_PREFETCHER_CONFIG + 0x10 received: 1

INFO : MM2S IO_BASE_ETH_DMA0_RX_PREFETCHER_CONFIG + 0x10 at
    IO_BASE_ETH_DMA0_RX_PREFETCHER_CONFIG + 0x10 received: 1

INFO - get_status_dma 970
INFO - HL1_DMA0_reset 982
Reset Prefetcher: 0
Stop Dispatcher: 17
Reset Dispatcher: 0
```

Listing 3.4: DMA Configurations Log Output

Test Completion: The test case completes with the transfer of ethernet packets from memory to outside Ethernet DMA through the transmit DMA path, and then the same packets are sent back to be written to the memory. To check if the test worked, buffer values of the ethernet frames transmitted and received are printed in the log. Listing 3.5 shows log output comparing the contents of an ethernet frame transmitted and received, indicating that the packet was transmitted and received properly.

```
Comparing 146 values:eth_frame_size
TX Buffer Value: 0xaa, RX Buffer Value: 0xaa
TX Buffer Value: 0x4, RX Buffer Value: 0x4
TX Buffer Value: 0x75, RX Buffer Value: 0x75
TX Buffer Value: 0xc8, RX Buffer Value: 0xc8
TX Buffer Value: 0x28, RX Buffer Value: 0x28
TX Buffer Value: 0x65, RX Buffer Value: 0x65
TX Buffer Value: 0xaa, RX Buffer Value: 0xaa
TX Buffer Value: 0x1, RX Buffer Value: 0x1
TX Buffer Value: 0x2, RX Buffer Value: 0x2
```

```
TX Buffer Value: 0xfa, RX Buffer Value: 0xfa
TX Buffer Value: 0x70, RX Buffer Value: 0x70
TX Buffer Value: 0xaa, RX Buffer Value: 0xaa
TX Buffer Value: 0x81, RX Buffer Value: 0x81
TX Buffer Value: 0x0, RX Buffer Value: 0x0
TX Buffer Value: 0x0, RX Buffer Value: 0x0
TX Buffer Value: 0xa, RX Buffer Value: 0xa
TX Buffer Value: 0x0, RX Buffer Value: 0x0
TX Buffer Value: 0x80, RX Buffer Value: 0x80
TX Buffer Value: 0xa, RX Buffer Value: 0xa
TX Buffer Value: 0xa, RX Buffer Value: 0xa
TX Buffer Value: 0xa, RX Buffer Value: 0xa
TX Buffer Value: 0xa, RX Buffer Value: 0xa
TX Buffer Value: 0xa, RX Buffer Value: 0xa
TX Buffer Value: 0xa, RX Buffer Value: 0xa
```

Listing 3.5: Ethernet Frame Comparison

Listing 3.6 shows one of the updated descriptors for streaming to memory-mapped transaction. The second field the write address (0x50000500) where the first packet has been written in the memory. Descriptor field at address 0x50003808 gives the length of the ethernet packet, which is 0x92 (146 bytes). Actual bytes transferred field at address 0x50003810 gives the total total bytes written to the memory, in this case it is 0x92 (146 bytes) which is the size of the ethernet packet. Status field at address 0x50003814 gives 0x100, which shows that there is no error and Early Termination bit (bit 8) is set indicating that all the bytes are transferred successfully and the operation has been completed. Control register field at address 0x5000383C generated a value of 0x82FFD300. This register sets some useful bits and IRQs, like GO bit (bit 31) which indicates that this particular descriptor has been written successfully, Transfer Complete IRQ Enable (bit 14) indicates that the transfer has completed, i.e. write master has finished its job.

```
DEBUG : dump_s2mm_scatter_gather_descriptor line no 474
DEBUG : Address = 0x50003800 Value = 0x0
DEBUG : Address = 0x50003804 Value = 0x50000500
DEBUG : Address = 0x50003808 Value = 0x92
DEBUG : Address = 0x5000380c Value = 0x50003900
DEBUG : Address = 0x50003810 Value = 0x92
DEBUG : Address = 0x50003814 Value = 0x100
DEBUG : Address = 0x5000383c Value = 0x82ffd300
```

Listing 3.6: S2MM Descriptor Updated

Chapter 4
Implementation

In this chapter, the implementation of design and test case setup for measurements will be discussed. The following sections explain the approaches and implementations used to achieve the performance measurements.

4.1 Test Case Scenario

In the previous chapter, the test case structure was discussed. A loop-back test case scenario which was used to verify the functionality of the design was presented. However, the loop-back test can not be used to measure the performance of the design because: (1) Both the data links are active in the loop-back test case scenario. (2) Packets may drop due to different processing times for S2MM and MM2S paths. (3) Ethernet packets were created using the C language function in the test case, not from real hardware.

Therefore, in order to measure the performance of the Ethernet DMA datapath for S2MM and MM2S, both the hardware design and the test case was modified. The following sections talk more about the implementation of design and test case.

4.1.1 Ethernet Packet Generator

The Ethernet Packet Generator (EPG) is a hardware module which has been written in SystemVerilog, and it generates ethernet packets. It creates ethernet packets which are composed of two parts, header and payload. Ethernet Packet Generator creates a packet with dedicated 50 bytes for the header, while the payload of the packet may vary. The structure of Ethernet header is shown in Figure 4.1. EPG module is designed to generate ethernet packet of specific header structure, which uses 6 words of 8 bytes and 2 bytes, making a total of 50 bytes. Header consists of 6 bytes of destination MAC address, 6 bytes of source MAC address, 4 bytes are dedicated to the Ethernet type, 2 bytes are used for the length field. The next 4 bytes are not used which are followed by 4 bytes of start address. The remaining bytes are unused in our case. An example ethernet header generated from EPG module is shown in Figure 4.2. EPG module uses a parameter called Inter-Packet Gap (IPG), which gives the time gap between two ethernet packets.

Byte	Ethernet Header Structure	Bit Length
0 - 7	Destination MAC [47:0] + Source MAC [47:32]	64
8 - 15	Source MAC [31:0] + Ethernet Type [31:0]	64
16 - 23	Length[15:0] + Unused [31:0] + Start Address [31:16]	64
24 - 31	Start Address [15:0] + Unused	64
32 - 39	Unused	64
40 - 47	Unused	64
48 - 49	Unused	16

Figure 4.1: Ethernet Header Structure

Destination MAC	Source MAC	Ethernet Type	Length	Unused	Start Address	Unused
6 Bytes	6 Bytes	4 Bytes	2 Bytes	4 Bytes	4 Bytes	24 Bytes
0x111111111111	0x222222222222	0xEBC01110	0x040C	0x0	0x40000100	0x0

Figure 4.2: Ethernet Header Example

4.1.2 Test Case using Ethernet Packet Generator

Ethernet Packet Generator module, which produces ethernet packets is connected directly to the DMA via packet drop checker and the new design is synthesized and built. The test case was modified to run on the new synthesized design. Now, the test case uses the EPG module to create ethernet packets to send them through the Ethernet DMA datapath. To get the proper performance measurements, first the packets need to be received at the receiver without any packet drop. A test is carried out for a fixed IPG to find the number of packets dropped for different numbers of packets received. The test case is repeated with different IPGs to find the best IPG for which packet loss is zero for different packet sizes.

Ethernet Packet Generator has a number of registers that are used to configure the ethernet packet size and header. A list of useful registers is shown in Figure 4.3. Each register is of size 32 bits. First 2 registers are used to control the operation of it. Only one bit is used in both the registers. Data setting register is used to set the data length, i.e., payload size. Transfer setting register is used to set IPG value and number of packets. Address setting register represents the address of DDR where the ethernet packet needs to be written.

Figure 4.4 shows the design of Ethernet DMA after integrating Ethernet Packet Generator module. A number of tests were carried out to understand the behaviour of the Ethernet DMA when it receives different numbers of packets with different packet sizes. Using the register configurations for the packet generator, the test case was run to observe the number of packets dropped while sending the packets to RX DMA. The number of packets dropped were read from the Packet

Address Offset	Register
0x0	Control/Status Register
0x4	Loopback Control
0x8	Data Setting
0xC	Transfer Setting
0x10	Address Setting

Figure 4.3: Ethernet Packet Generator Registers

Drop module register present in the Ethernet DMA. The test case was run to create 10000 ethernet packets from Ethernet Packet Generator. A number of tests were performed for different ethernet packet sizes to observe the packets dropped and keeping the inter packet gap value as 1. It was found that number of packets dropped for different packet sizes.

Test case was run again by changing the IPG values for different packet sizes and it was observed that as the IPG values were increased, the number of packets dropped were reduced. A number of tests were performed to find the best IPG values for different packet sizes where the number of packets dropped reduced to zero.

Once the best IPG values for different number of packet sizes were found, the next step was to measure the performance by writing the ethernet packets to memory. A number of different tests were run to measure the performance for both the streaming to memory-mapped and the memory-mapped to streaming paths. The test case steps for these tests are explained in the next sections.

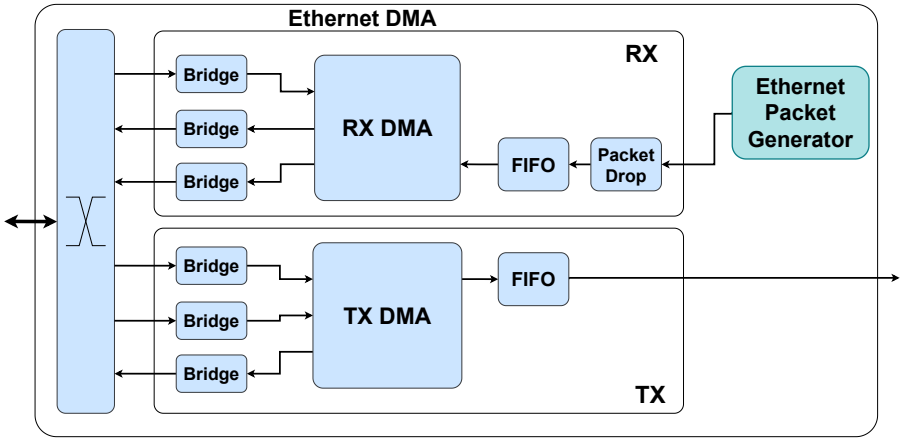

Figure 4.4: Ethernet DMA with Packet Generator

4.1.3 Streaming to Memory Mapped Test

Streaming to memory-mapped path starts with the Ethernet Packet Generator module which creates ethernet packets which are sent to be written to the DDR memory. The full datapath of the scenario is given in Figure 4.5, which uses RX DMA of the Ethernet DMA to process ethernet packets generated from EPG module. The packets are first fed into the Packet Drop module and then to the FIFO. The DMA core, which is a scatter-gather DMA is configured with the descriptors which are fetched in the Prefetcher module of the RX DMA. Then the descriptors are sent to the Dispatcher module which decodes them and issues the write command to the Write Master module. Following the command, Write Master sends Ethernet packets arriving at its input from FIFO towards the HPS subsystem to write them to the DDR. When the Write Master finishes writing a packet to the DDR, it sends the Write Response signal to the Dispatcher, which then sends a response signal to Prefetcher module. Prefetcher then updates the descriptor confirming the write operation. The datapath of the test is highlighted in brown in the Figure 4.5.

The test case follows a series of steps to run properly. Some important steps to run the test case used are given below:

- In this first step, it is made sure the FPGA sleeps for some time before starting it. It is also made sure that it completes the previous operations, if it is running any.
- The test case starts with the initialization of the clocks and reset configuration.
- Before starting the Ethernet Packet Generator module to generate the ethernet packets, the DDR memory is cleared according to the size of the packets and number of packets.
- Once the DDR memory is cleared, the Ethernet Packet Generator is configured with required settings using its registers.
- After configuring the Ethernet Packet Generator, the next step is to create the descriptors. The descriptors are created according to the configuration provided in the test case. The configuration used in the test case is multiple descriptors and multiple packets. Multiple descriptors are created wherein, each descriptor carries one packet. Hence, the descriptors are written in a way that the number of packets is equal to the number of descriptors.
- Once the descriptors are created, the next step is to configure the Prefetcher and Dispatcher of the Ethernet DMA.
- After configuring all the modules, Ethernet DMA is started.
- A timestamp is created, once Ethernet DMA starts. The test case uses a timestamp to measure the time with software.
- Ethernet Packet Generator starts generating packets and send them to the Receiver DMA.

- Ethernet Packet Generator completes generating all the packets.
- Once all the packets have been transferred, Prefetcher and Dispatcher of RX DMA complete their operation.
- Create one more timestamp to measure time after completion of operation.
- Check Packet drop registers if there are any packets dropped.
- Check DDR values with manually calculated values using a function which follows same logic as Ethernet Packet Generator does.

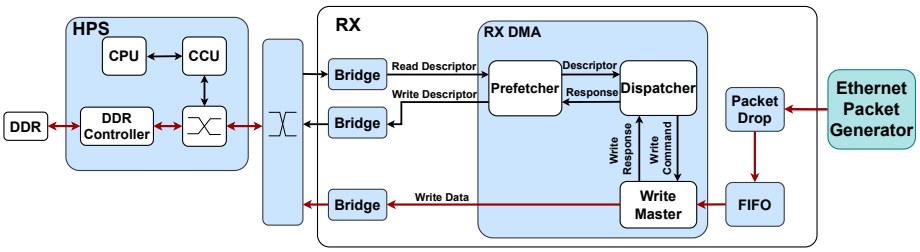

Figure 4.5: Streaming to Memory Mapped Datapath

4.1.4 S2MM Test Example

The test case as explained in the previous section follows steps to send ethernet packets to write to the memory. The test case was run to transfer a number of different ethernet packet sizes. The test case uses multiple descriptors and multiple packets configuration, i.e. each descriptor will have the instructions to transfer one ethernet packet. Let's take an example run of the test case, where we run the test to generate 10,000 ethernet packets of size 450 bytes each from the EPG module and send them through Ethernet DMA to write to the memory.

Initial Setup: The test case starts with initial setup of design. A screenshot of the log file showing the initial setup of hardware while running the test case is shown in Listing 4.1. The test case starts with checking PLL blocks and locking them and all the resets are released to initialize the modules of the design.

```
1 autorun=0 num_pkts = 10000 ebcom_payload_size = 400 eth_dma =
      1 cbuff = 1 dump_ddr=0 ipg_length=152
2 INFO - run_ethdma_test  has started
3
4 DEBUG - IO_BASE_CCR:0xf919f000 IO_BASE_CCR+CCR_TEST_RD_OFFSET
      :0xf919f144
5
6 DEBUG - read_value:0x55555555
7
8 Checking writing to CCR test register
9
```

```
Poll PLLs B and C some time and report error if not all of
    them are locked.

For max R/W and also to check PLL reset, perform reset (should
     be set for at least 1us!) and then check for lock again

DEBUG - CCR_PLL_RESET_CTRL address:0xf919f15c read_value:0x333
      exp_value:0x33

WARN - CCR Max RW Value: 0x333 expected: 0x33
DEBUG - ADDING SOME OF THE OTHER ACCESSES HERE TO MAKE SURE WE
     GET AT LEAST 1us RESET ASSERTED
DEBUG - Read CCR_DEBUG_STATUS
address:0xf919f164  read_value:0x0  exp_value:0x22200

WARN - CCR Max RO Value: 0x0 expected: 0x22200
DEBUG - Release Reset
DEBUG - Poll for Lock again
INFO - Poll for PLLs B and C Lock after soft reset

DEBUG - After PLL are Locked, Release clock gating of B and C,
     writing 0x3 and then 0x0 at address: 0xf919f16c
DEBUG - Release TRX reset, writing 0x0 at address: 0xf919f170
DEBUG - Release all blocks from reset, writing 0x3FFF and then
     0x0 at address: 0xf919f160
DEBUG - Check PLL locked again, and IC reset control partly to
     pass time to make sure all resets are fully released at
    end of this function, reading value at 0xf919f17c
DEBUG - INFO - Read out at address:0xf919f148 LOCKSTATUS:0x7
    exp_value:0x7
DEBUG - RD_CCR_TRX_RST_CTRL_ETH10G_TRX_PHY_RST=0x0
DEBUG - RD_CCR_BLOCK_RESET_CTRL_TRACE_DMA0_RST=0x0
DEBUG - Enabling requested blocks address=0xf919f000
INFO - run_ethdma_test 506 num_pkts = 10000 ebcom_payload_size
     = 400 dma = 1 check_buff = 1
```

Listing 4.1: S2MM Test - Initial Setup

Ethernet Packet Generation: After starting the system, the Ethernet Packet Generator module is configured to create ethernet packets. The packets with the frame format explained in Section 4.1.1 are created by configuring the registers of EPG module as explained in Section 4.1.2. The packet consists of 50 bytes of header and 400 bytes of payload size. Thus, a toal of 10,000 Ethernet packets of size 450 bytes are created from EPG module. The log output showing the configuration of registers for generating the packets is shown in Listing 4.2.

```
DEBUG - configure_bridge 450 address=0xf91a79c0 initial
    read_value=0x60f82f02
DEBUG - configure_bridge 454 address=0xf91a79c0 write_value=0
    x3c008008
DEBUG - configure_bridge 458 address=0xf91a79c0 final
```

```
    read_value=0x3c008008
INFO - prim_header_include=0  buffer_size=0
DEBUG - disable_ebcom_generator_loopback 413 returning
    read_value=0x0
DEBUG - disable_ebcom_generator_loopback 415 returning
    read_value=0x1
DEBUG - data_seed=0x7b data_length=0x190 writing data_settings
    =0x7b0190 at memlocation=0xf91a0008
DEBUG - Reading EBCOM DATA SETUP Address=f91a0008 Value=1
DEBUG - error, write_value=0x7b0190 is not equal to read_value
    =0x1 at address=0xf91a0008
DEBUG - Reading 2nd time EBCOM DATA SETUP Address=f91a0008
    Value=7b0190
DEBUG - transfer_type=0x0 ipg_length=0x98 transfer_length=0
    x2710 writing transfer_settings=0x982710 at memlocation=0
    xf91a000c
DEBUG - Reading EBCOM TRANSFER SETUP Address=f91a000c Value=7
    b0190
WARN - write_value=0x982710 is not equal to read_value=0
    x7b0190 at address=0xf91a000c
DEBUG - Reading 2nd time EBCOM TRANSFER SETUP Address=f91a000c
     Value=982710
DEBUG - writing start_ddr_address=0x40000100 at memlocation=0
    xf91a0010
DEBUG - Reading EBCOM ADDR SETUP Address=f91a0010 Value=982710
WARN - write_value=0x40000100 is not equal to read_value=0
    x982710 at address=0xf91a0010
DEBUG - Reading 2nd time EBCOM ADDR SETUP Address=f91a0010
    Value=40000100
DEBUG - Returning from configure_ebcom_generator
```

Listing 4.2: S2MM - Ethernet Packet Creation

Descriptor Creation: Once the Ethernet packets are created after configuring the EPG module, the next step is to create descriptors. In our case, 10,000 descriptors are created for 10,000 ethernet packets with packet size of 450 bytes. Listing 4.3 shows the log output displaying the function call where the descriptors are being created.

DMA Configuration: After the EPG module is configured to generate packets and descriptors are created, the next step is to configure the SGDMA. Listing 4.3 shows the log file output displaying configuration of Prefetcher and Dispatcher block of RX DMA. Once all the blocks of RX DMA are configured to transfer the ethernet packets, the Ethernet DMA starts its operation and the data is sent to be written to the memory.

```
INFO -Create s2mm desc
create_s2mm_descriptors 1192 payload_size=400 length_buff=450
     header_size=36 is_ebcom_test=0
create_s2mm_descriptors 1215 first descriptor i=0 next_id=1
create_s2mm_descriptors 1227 last descriptor i=9999 next_id
    =9999
```

```
start_dma_and_wait_for_done 870 s2mm_desc_phyaddr_current
    =50003800 mm2s_desc_phyaddr_current=50000280
IO_BASE_ETH_DMA_0_S2MM_CSR_BASE + 0x4 = 0xf91a7944 value=0x10
IO_BASE_ETH_DMA0_RX_PREFETCHER_CONFIG + 0x4 = 0xf91a7964 value
    =0x50003800
IO_BASE_ETH_DMA0_RX_PREFETCHER_CONFIG = 0xf91a7960 value=0x9
INFO - Starting timer and the EBCom packet generator
DEBUG - start_ebcom_generator 421 Writing value=0x1 at
    memlocation=0xf91a0000
DEBUG - write_value=0x1 NOT MATCHING read_value=0x1 at address
    =0xf91a0000
DEBUG - Returning from start_ebcom_generator reading value at
    memlocation=0xf91a0000 value=0x1
INFO - HL1_DMA_wait_dma_done 900
INFO : S2MM IRQ at IO_BASE_ETH_DMA_0_S2MM_CSR_BASE + 0x0
    received: 200

INFO : S2MM IO_BASE_ETH_DMA0_RX_PREFETCHER_CONFIG + 0x10 at
    IO_BASE_ETH_DMA0_RX_PREFETCHER_CONFIG + 0x10 received: 1

DEBUG - poll_ebcom_generator 441 returning state=0x1
DEBUG - poll_ebcom_generator 443 returning state=0x1
INFO - HL1_DMA_wait_dma_done 900
INFO : S2MM IRQ at IO_BASE_ETH_DMA_0_S2MM_CSR_BASE + 0x0
    received: 200

INFO : S2MM IO_BASE_ETH_DMA0_RX_PREFETCHER_CONFIG + 0x10 at
    IO_BASE_ETH_DMA0_RX_PREFETCHER_CONFIG + 0x10 received: 1

DEBUG - poll_ebcom_generator 441 returning state=0x1
DEBUG - poll_ebcom_generator 443 returning state=0x1
INFO - HL1_DMA_wait_dma_done 900
INFO : S2MM IRQ at IO_BASE_ETH_DMA_0_S2MM_CSR_BASE + 0x0
    received: 200

INFO : S2MM IO_BASE_ETH_DMA0_RX_PREFETCHER_CONFIG + 0x10 at
    IO_BASE_ETH_DMA0_RX_PREFETCHER_CONFIG + 0x10 received: 1
```

Listing 4.3: S2MM - Descriptor Creation & DMA Configuration

Completion of Test: The test case completes with the transfer of 10,000 ethernet packets from the EPG module to DDR memory. As shown in the log output in Listing 4.4, the test took 0.026095 seconds time to write 10000 packets of size 450 bytes. To verify that all the packets were written properly in the memory, we read the registers for packet drop. As the log output printing the register values shows that there were no packet dropped, and all the packets sent and received in the memory are compared and they match as shown in the log file (here, showing first two packets only).

```
DEBUG - elapsed=0.026095 secs

DEBUG - Reading Packet drop count due to error Address=
```

```
    f919f184 Value=0
DEBUG - Reading Packet drop count due to no space Address=
    f919f188 Value=0
DEBUG - Reading Packet drop count due to Transmitter ready
    Address=f919f18c Value=0
DEBUG - Resetting ETH DMA
INFO - HL1_DMA0_reset 983

##########################PACKET NO
    0000################################
DEBUG - Compare a_tx_buffer = 0xaaaad6347310,a_size:450
    a_rx_buffer = 0x40000100 for packet 0

DEBUG - Compare a_tx_buffer = 0xaaaad6347310 a_rx_buffer = 0
    x40000100 for packet 0

##########################PACKET NO
    0001################################
DEBUG - Compare a_tx_buffer = 0xaaaad6347310,a_size:450
    a_rx_buffer = 0x400002c2 for packet 1

DEBUG - Compare a_tx_buffer = 0xaaaad6347310 a_rx_buffer = 0
    x400002c2 for packet 1
```

Listing 4.4: S2MM - Test Completion

Finally, the descriptors contents are printed to verify the operation. Listing 4.5 presents updated descriptors after the ethernet packets are written to the memory. As shown in the log output, the read address field (0x50003800) of of the first descriptor is 0x0 as it is not applicable for S2MM case. Write address field (0x50003804) has the starting address (0x40000100) where the ethernet packet has been written to. Address field 0x50003808 shows the size of the ethernet packet, that is 0x1C2 (450 bytes). Next descriptor pointer points to the address 0x50003E00 as each descriptor occupies a size of 0x600. Actual bytes transfer field gives the total bytes transferred, which in this case is 450 bytes as each descriptor has one ethernet packet. Status field at address 0x50003814 gives 0x100, which shows that there is no error and Early Termination bit (bit 8) is set indicating that all the bytes are transferred successfully and the operation has been completed. Control register field at address 0x5000383C has generated a value of 0x82FFD300. This register sets some useful bits and IRQs, like GO bit (bit 31) which indicates that this particular descriptor has been written, Transfer Complete IRQ Enable (bit 14) indicates that the transfer has completed, i.e. write master has finished its job. Early Termination IRQ Enable (bit 15) is set which signals an interrupt as the transfer completes early. Similar to the first descriptor, all other descriptors are updated for all the packets transferred.

```
DEBUG : dump_s2mm_scatter_gather_descriptor line no 474
DEBUG : Address = 0x50003800 Value = 0x0
DEBUG : Address = 0x50003804 Value = 0x40000100
DEBUG : Address = 0x50003808 Value = 0x1c2
```

```
DEBUG : Address = 0x5000380c Value = 0x50003e00
DEBUG : Address = 0x50003810 Value = 0x1c2
DEBUG : Address = 0x50003814 Value = 0x100
DEBUG : Address = 0x5000383c Value = 0x82ffd300
DEBUG : dump_s2mm_scatter_gather_descriptor line no 555
    completed
DEBUG : dump_s2mm_scatter_gather_descriptor line no 474
DEBUG : Address = 0x50003e00 Value = 0x0
DEBUG : Address = 0x50003e04 Value = 0x400002c2
DEBUG : Address = 0x50003e08 Value = 0x1c2
DEBUG : Address = 0x50003e0c Value = 0x50004400
DEBUG : Address = 0x50003e10 Value = 0x1c2
DEBUG : Address = 0x50003e14 Value = 0x100
DEBUG : Address = 0x50003e3c Value = 0x82ffd300
DEBUG : dump_s2mm_scatter_gather_descriptor line no 555
    completed
DEBUG : dump_s2mm_scatter_gather_descriptor line no 474
DEBUG : Address = 0x50004400 Value = 0x0
DEBUG : Address = 0x50004404 Value = 0x40000484
DEBUG : Address = 0x50004408 Value = 0x1c2
DEBUG : Address = 0x5000440c Value = 0x50004a00
DEBUG : Address = 0x50004410 Value = 0x1c2
DEBUG : Address = 0x50004414 Value = 0x100
DEBUG : Address = 0x5000443c Value = 0x82ffd300
DEBUG : dump_s2mm_scatter_gather_descriptor line no 555
    completed
```

Listing 4.5: S2MM - Updated Descriptors

4.1.5 Memory-mapped to Streaming Test

Memory-mapped to streaming test attempts to measure the performance of the Ethernet DMA with the datapath starting from the DDR memory to the FIFO output as shown in the datapath in Figure 4.6. Ethernet packets are read from the memory and transmitted to the BBM of the system top through the TX DMA present in the Ethernet DMA. The DMA core, which is a scatter-gather DMA is configured with the descriptors which are fetched in the Prefetcher module of the TX DMA. Then the descriptors are sent to the Dispatcher module which decodes them and issues the read command to the Read Master block. Following the command, Read Master sends ethernet packets arriving at the input of Read Master from DDR via interconnects and bridges to FIFO to transmit them to the outside to BBM device. When the Read Master reads a packet from the DDR, it sends the Read Response signal to the Dispatcher, which then sends a response signal to Prefetcher module. Prefetcher then updates the descriptor confirming the read operation. The path highlighted in brown in Figure 4.6 shows the datapth of MM2S test case. MM2S test case follows a series of steps to run properly which are explained below:

- To read the packets from the memory, first the Ethernet packets are gen-

erated with the Ethernet Packet Generator module present in the Ethernet DMA. Then the ethernet packets are written to the memory via S2MM datapath as explained in the previous section.

- Once, it is made sure that all the packets generated have been written to the memory, the FPGA is kept on sleep mode for some time.
- Next, the test case for MM2S path starts with initialization of the clocks and reset configuration.
- MM2S descriptors are created according to the configuration provided in the test case. The test case uses multiple descriptors and multiple packets configuration in which multiple descriptors are created wherein, each descriptor carries one ethernet packet.
- The Prefetcher and Dispatcher blocks are configured in the TX DMA of Ethernet DMA.
- A timestamp is created, once Ethernet DMA's modules are configured. The test case uses a timestamp to measure the time with software
- Ethernet DMA starts its operation.
- Once all the packets are read, then Ethernet DMA's modules complete their operation.
- Check the FIFO almost empty bit in FIFO_STATUS_LEVEL register to check if all the packets have been transmitted successfully.
- Create one more timestamp to measure time after completion of operation.

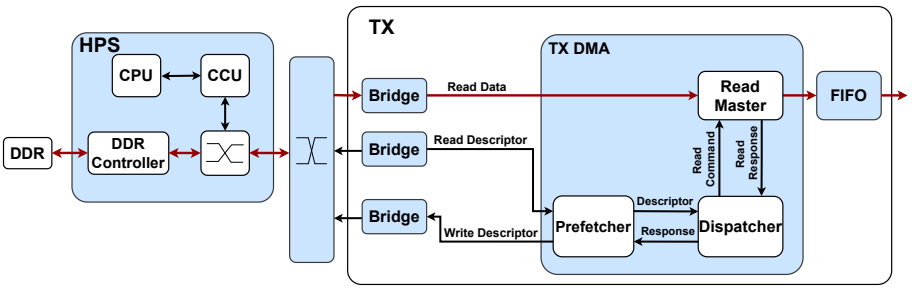

Figure 4.6: Memory Mapped to Streaming Datapath

4.1.6 MM2S Test Example

MM2S path can be tested by reading the ethernet packets from memory and sending them through the DMA to transmit. As explained in the test case flow for the memory-mapped to streaming datapath, we ran a test to read 10,000 ethernet packets of size 450 bytes from memory. The test case uses multiple descriptors and multiple packets configuration, i.e. each descriptor will have the instructions

to transfer one ethernet packet.

Initial Setup: The test case starts with initial setup of design. A log file showing the initial hardware setup while running the test case is shown in the Listing 4.6. The test case checks PLL blocks and locking is done and all the resets are released to initialize the modules of the design.

```
autorun=0 num_pkts = 10000 ebcom_payload_size = 400 eth_dma = 1 cbuff = 0 dump_ddr=0 ipg_length=1
INFO - run_ethdma_test  has started

DEBUG - IO_BASE_CCR:0xf919f000 IO_BASE_CCR+CCR_TEST_RD_OFFSET :0xf919f144
DEBUG - read_value:0x55555555
Checking writing to CCR test register
Poll PLLs B and C some time and report error if not all of them are locked.
For max R/W and also to check PLL reset, perform reset (should be set for at least 1us!) and then check for lock again
DEBUG - CCR_PLL_RESET_CTRL address:0xf919f15c read_value:0x333 exp_value:0x33

WARN - CCR Max RW Value: 0x333 expected: 0x33
DEBUG - ADDING SOME OF THE OTHER ACCESSES HERE TO MAKE SURE WE GET AT LEAST 1us RESET ASSERTED
DEBUG - Read CCR_DEBUG_STATUS
address:0xf919f164  read_value:0x0  exp_value:0x22200

WARN - CCR Max RO Value: 0x0 expected: 0x22200
DEBUG - Release Reset
DEBUG - Poll for Lock again
INFO - Poll for PLLs B and C Lock after soft reset

DEBUG - After PLL are Locked, Release clock gating of B and C, writing 0x3 and then 0x0 at address: 0xf919f16c
DEBUG - Release TRX reset, writing 0x0 at address: 0xf919f170
DEBUG - Release all blocks from reset, writing 0x3FFF and then 0x0 at address: 0xf919f160
DEBUG - Check PLL locked again, and IC reset control partly to pass time to make sure all resets are fully released
at end of this function, reading value at 0xf919f17c
DEBUG - INFO - Read out at address:0xf919f148 LOCKSTATUS:0x7 exp_value:0x7
DEBUG - RD_CCR_TRX_RST_CTRL_ETH10G_TRX_PHY_RST=0x0
DEBUG - RD_CCR_BLOCK_RESET_CTRL_TRACE_DMA0_RST=0x0
DEBUG - Enabling requested blocks address=0xf919f000
INFO - run_ethdma_test 505 num_pkts = 10000 ebcom_payload_size = 400 dma = 1 check_buff = 0

DEBUG - configure_bridge 450 address=0xf91a79c0 initial
```

```
    read_value=0x60f82f02
33 DEBUG - configure_bridge 454 address=0xf91a79c0 write_value=0
    x3c008008
34 DEBUG - configure_bridge 458 address=0xf91a79c0 final
    read_value=0x3c008008
35 INFO - prim_header_include=0  buffer_size=0
36 DEBUG - IO_BASE_CCR:0xf919f000 IO_BASE_CCR+CCR_TEST_RD_OFFSET
    :0xf919f144
```

Listing 4.6: MM2S Test - Initial Setup

Descriptor Creation: MM2S descriptors are created using the functions in the test case based on the configuration through which the packets need to be sent. As in this example, 10,000 descriptors are created for 10,000 ethernet packets with packet size of 450 bytes, which is shown as log output in the Listing 4.7.

```
1 DEBUG - configure_bridge 450 address=0xf91a79c0 initial
    read_value=0x60f82f02
2 DEBUG - configure_bridge 454 address=0xf91a79c0 write_value=0
    x3c008008
3 DEBUG - configure_bridge 458 address=0xf91a79c0 final
    read_value=0x3c008008
4 create_mm2s_descriptors 1178 payload_size=400 length_buff=450
     header_size=36 is_ebcom_test=0
5 create_mm2s_descriptors 1199 first descriptor i=0 next_id=1
6 initialize_mm2s_scatter_gather_descriptor 46 eth_frame_size
    =450, mm2s_desc_phyaddr=50000280, mm2s_buff_addr=40000100,
    descposition=3, mm2s_desc_phyaddr_next=50000880
7 create_mm2s_descriptors 1229 i=1 next_id=2
8 initialize_mm2s_scatter_gather_descriptor 46 eth_frame_size
    =450, mm2s_desc_phyaddr=50000880, mm2s_buff_addr=400002c2,
    descposition=3, mm2s_desc_phyaddr_next=50000e80
9 create_mm2s_descriptors 1229 i=2 next_id=3
10 initialize_mm2s_scatter_gather_descriptor 46 eth_frame_size
    =450, mm2s_desc_phyaddr=50000e80, mm2s_buff_addr=40000484,
    descposition=3, mm2s_desc_phyaddr_next=50001480
11 create_mm2s_descriptors 1229 i=3 next_id=4
12 initialize_mm2s_scatter_gather_descriptor 46 eth_frame_size
    =450, mm2s_desc_phyaddr=50001480, mm2s_buff_addr=40000646,
    descposition=3, mm2s_desc_phyaddr_next=50001a80
13 create_mm2s_descriptors 1229 i=4 next_id=5
14 initialize_mm2s_scatter_gather_descriptor 46 eth_frame_size
    =450, mm2s_desc_phyaddr=50001a80, mm2s_buff_addr=40000808,
    descposition=3, mm2s_desc_phyaddr_next=50002080
15 create_mm2s_descriptors 1229 i=5 next_id=6
16 initialize_mm2s_scatter_gather_descriptor 46 eth_frame_size
    =450, mm2s_desc_phyaddr=50002080, mm2s_buff_addr=400009ca,
    descposition=3, mm2s_desc_phyaddr_next=50002680
17 create_mm2s_descriptors 1229 i=6 next_id=7
18 initialize_mm2s_scatter_gather_descriptor 46 eth_frame_size
    =450, mm2s_desc_phyaddr=50002680, mm2s_buff_addr=40000b8c,
```

```
    descposition=3, mm2s_desc_phyaddr_next=50002c80
19 create_mm2s_descriptors 1229 i=7 next_id=8
20 initialize_mm2s_scatter_gather_descriptor 46 eth_frame_size
    =450, mm2s_desc_phyaddr=50002c80, mm2s_buff_addr=40000d4e,
    descposition=3, mm2s_desc_phyaddr_next=50003280
```

Listing 4.7: MM2S - Descriptor Creation

DMA Operation: Once descriptors are created, TX DMA is configured to read descriptors to transmit ethernet packets out. Listing 4.8 shows the log output displaying configuration of TX DMA and its operation. The DMA operation completes with 0.000344 seconds time elapsed calculated from the test case to transmit 10000 packets. Once the Ethernet DMA finishes its job, it gets reset.

```
1  initialize_mm2s_scatter_gather_descriptor 46 eth_frame_size
       =450, mm2s_desc_phyaddr=50ea5c80, mm2s_buff_addr=4044a95e,
       descposition=3, mm2s_desc_phyaddr_next=50000280
2  start_dma_and_wait_for_done_mm2s 896 mm2s_desc_phyaddr_current
       =50000280
3  IO_BASE_ETH_DMA_0_MM2S_CSR_BASE + 0x4=0xf91a7904 value=0x10
4  IO_BASE_ETH_DMA0_TX_PREFETCHER_CONFIG + 0x4=0xf91a7924 value=0
       x50000280
5  IO_BASE_ETH_DMA0_TX_PREFETCHER_CONFIG=0xf91a7920 value=0x9
6  INFO - HL1_DMA_wait_dma_done_mm2s 987
7  INFO : MM2S IRQ at IO_BASE_ETH_DMA_0_MM2S_CSR_BASE + 0x0
       received: 200
8
9  INFO : MM2S IO_BASE_ETH_DMA0_TX_PREFETCHER_CONFIG + 0x10 at
       IO_BASE_ETH_DMA0_TX_PREFETCHER_CONFIG + 0x10 received: 1
10
11 INFO - get_status_dma_mm2s 1071
12
13 DEBUG - elapsed=0.000344 secs
14
15 DEBUG - Resetting ETH DMA
16 INFO - HL1_DMA0_reset 1083
17 # NOTE: Test has ended with 0 error(s) and 4 warning(s) in
       test
```

Listing 4.8: MM2S - DMA Operation

4.1.7 Drawbacks of Software Testing

The test case was initially run to get the performance measurements using the software, wherein the particular registers were read to get the values and timestamp was used to measure the time. To get the more realistic and accurate measurements, the results obtained through software can't be relied upon. There is always a chance to get incorrect reading. Timestamp works based on the system clock. It may add some overhead to the timing which needs to be measured.

4.2 Performance Measurement using Hardware

The performance of the Ethernet DMA can be measured using both software and hardware approaches. Initially, the performance was measured using the test case by reading the relevant registers in the software. To get the real time more accurate numbers, the performance was measured from the hardware using an oscilloscope. To get the proper signal values and time from the design, the relevant signal outputs were taken from the FPGA board pins to connect to the oscilloscope. Considering both the streaming to memory-mapped and memory-mappped to streaming datapaths, it was decided to take particular signals which would define the test completion. The methodology of the measurement is described below.

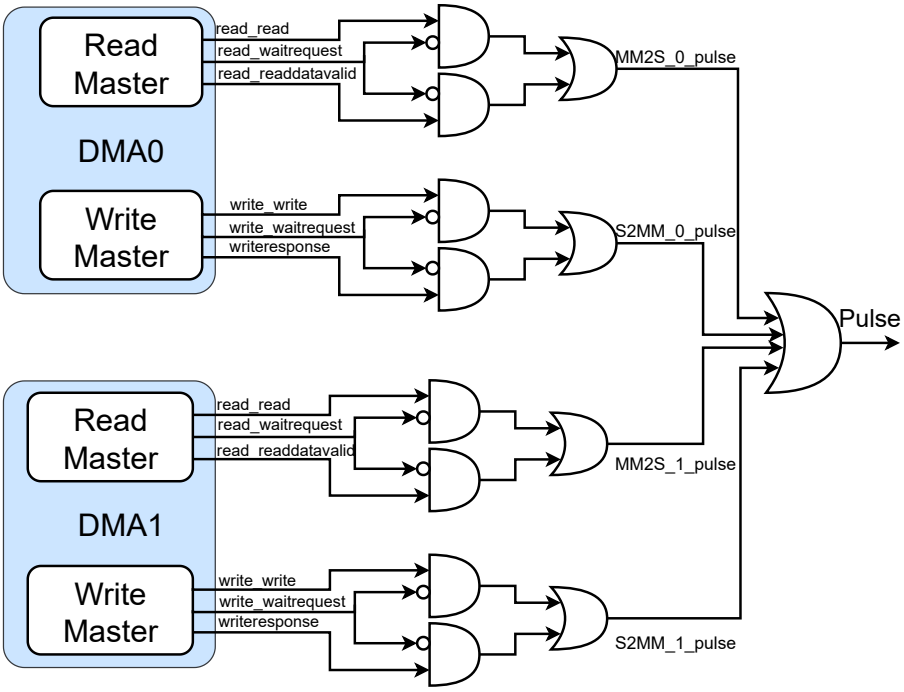

Figure 4.7: Pulse Signal Measurement Circuit

Figure 4.7 shows the implemented logic to get a pulse signal from the design to measure using the oscilloscope. As shown in the circuit, the relevant signals are taken from the DMA cores. The wait_request signal, which is active low (when it has value 1), indicates that DMA is not performing any operation and waiting for the data to be ready in FIFO or there is sufficient space available in FIFO to send in the packet. There are 2 signals connected to the master read/write modules of Ethernet DMA. Write Master uses two signals 'write' and 'writeresponse' in the streaming to memory-mapped path. Read Master uses 'read' and 'readdatavalid' signals in memory-mapped to streaming path. Whenever the DMA initiates a new transfer, read or write signal will become 1. When DMA completes an operation

based on the descriptor, it will be indicated by writeresponse and readdatavalid. So in order to get the total time period, the time between start of first packet transmission/reception and the response of the last packet transmission/reception needs to measured. The NOT of waitrequest indicates the DMA is active which is ANDed with the 2 signals from master and ORed to produce the pulse output as shown in Figure 4.7.
Pulse from each DMA is taken and ORed together which produces a single output. At a time only one DMA will be active to make sure that measurement is pertaining to that particular DMA operation. The pulse signal is taken out through FPGA's GPIO_TEST pin and connected to the oscilloscope shown as a block diagram Figure 4.8.

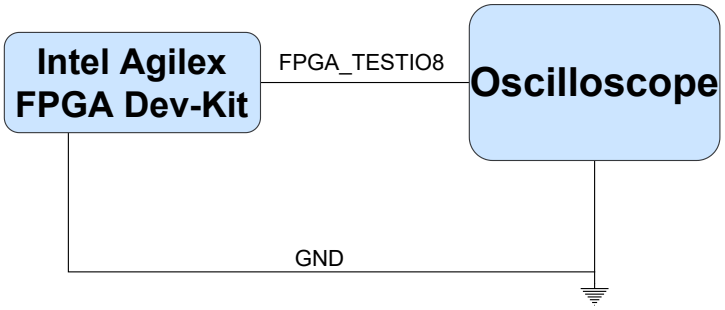

Figure 4.8: Lab Test Measurement Setup

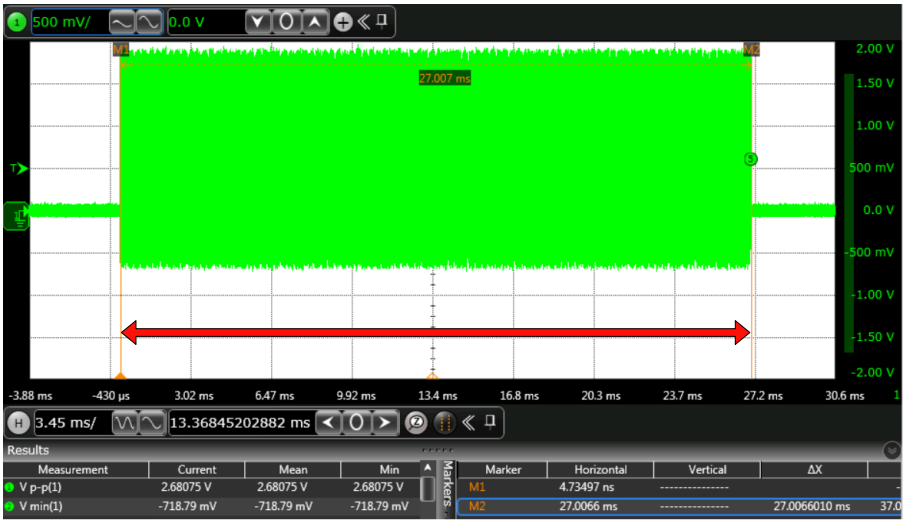

Figure 4.9: Oscilloscope Output

The oscilloscope output in Figure 4.9 shows the total time period calculated using two markers M1 and M2, when Ethernet packets were sent from the packet gener-

ator to be written to the memory. The difference between the two markers gives the time it takes to write total packets to the memory.

Figure 4.10: Oscilloscope Output - Start of First Packet Transfer

Figure 4.10 shows the first rising edge of the pulse signal which gives the start time when the first ethernet packet is written to the memory. In other words, it is the starting of the first packet from the FIFO to be sent to memory through the S2MM path.

Figure 4.11: Oscilloscope Output - Writing Last Packet

Figure 4.11 shows the pulse output which is measured at the falling edge and it gives the time when the last packet is written to the memory.

Chapter 5
Results

The implementation of the design and test case is presented in the last chapter. The results obtained from the performance measurement tests of Ethernet DMA are discussed in this chapter. The test case, which is written in C programming language, as explained in the previous chapters has been used to measure the performance of the design. Test case was run to measure certain performance parameters. These parameters include:

- Ethernet packets dropped
- Inter-Packet Gap
- Streaming to memory-mapped datapath performance
- Memory-mapped to streaming datapath performance

We will go through all the experimental results obtained for the performance measurement in the following sections.

5.1 Packet Drop

As discussed in the Chapter 3, there is a packet drop mechanism in the S2MM path which is handled by Packet Drop module. To get the number of packets dropped, we observe the packets dropped due to no space in the FIFO by reading the relevant register. After integrating Ethernet Packet Generator module to the design, the test case was run to check the number of packets dropped while sending the packets from the packet generator to write them to the memory. Table 5.1 presents the number of packets dropped when different numbers of packets were sent to the Ethernet DMA to write to the memory. The test case was run for multiple iterations to check and it was observed that for the same number of packets, the number of packets dropped remained almost same. However, it was found that the number of packets dropped increased considerably when the number of packets were increased. It can be observed from Table 5.1 that as the number of packets rose, the percentage of packet dropped rose significantly, starting with 10.5% drop for 80 packets to 47% drop for 10000 packets. The packets were sent through Ethernet DMA to write to the memory with a fixed packet size of 1050 bytes and IPG value of 1. The design uses a FIFO block and a Packet Drop module. While sending more number of packets, more packets are dropped because the

inter-packet gap value is fixed to 1. With more packets coming in, the FIFO does not have enough space to accommodate incoming new packets. Thus, the Packet Drop module drops the packets due to no space in the FIFO. The graph in Figure 5.1 presents the trend of dropped packets as the number of packets sent to the Ethernet DMA increase. As the number of packets increases, the number of packets dropped also increases and the percentage of packets dropping gets repeated from 1000 to 10000 packets so the curve starts to almost saturates from 3000 to 10000 packets. There is no linear relation for number of packets and percentage of packets dropped.

Table 5.1: Packet Drop for Different Number of Packets

No. of Packets	80	90	100	110	150	200	250	500	750	1000	3000	5000	10000
	Number of Packets Dropped												
Run 1	9	14	18	23	42	64	90	207	327	450	1400	2353	4733
Run 2	8	14	19	23	41	64	91	206	327	448	1399	2351	4732
Run 3	9	14	19	23	42	64	90	207	328	449	1400	2352	4734
Run 4	8	14	18	23	42	65	90	207	327	448	1400	2354	4733
Run 5	8	14	19	23	42	64	91	207	328	449	1399	2356	4733
Run 6	9	14	18	23	41	65	90	207	327	448	1400	2354	4735
Run 7	8	14	19	23	42	64	90	207	327	448	1399	2353	4734
Run 8	8	14	18	23	42	64	90	207	326	449	1399	2354	4733
Run 9	8	14	18	23	42	64	90	207	327	449	1401	2354	4735
Run 10	9	14	19	23	42	64	90	207	327	449	1398	2355	4735
Average	8.4	14	18.5	23	41.8	64.2	90.2	206.9	327.10	448.7	1399.5	2353.6	4733.7
Drop (%)	10.5	15.5	18.5	20.9	27.8	32.1	36.08	41.38	43.61	44.87	46.65	47.07	47.34

To check the effect of the packet size to the packets dropped, the tests were conducted on different packet sizes. Different numbers of packets (100 to 5000) were sent to the Ethernet DMA to be written to the memory. The purpose of measuring the packets dropped was to check the behaviour of Ethernet DMA whether it is dropping more packets as we increase packet size and number of packets. The test case was run with setting IPG value as 1. As presented in Table 5.2, the packets dropped have increased as we increase the number of packets and also more number of packets are dropped if we increase the packet size. However, the packets dropped are not linear with respect to the increase in number of packets. Let's take the number of packets dropped while processing 200 packets and 400 packets. The rise of packets dropped is not in proportion to the number of packets, because of many reasons. It can be affected by how the DMA is acting at that point, how much there is load on the CPU and whether the bus is busy. So, a relationship between increase of packets dropped and number of packets is difficult to find as a lot of other processes are running in the hardware at one point.

5.2 IPG Calculations

Different numbers of packets had a different number of packet drop when the inter packet gap value was set to 1. To make a zero packet drop, the best IPG value was found for each of different packet sizes by running the test multiple times. The packet sizes were chosen randomly and it was found that different packet sizes

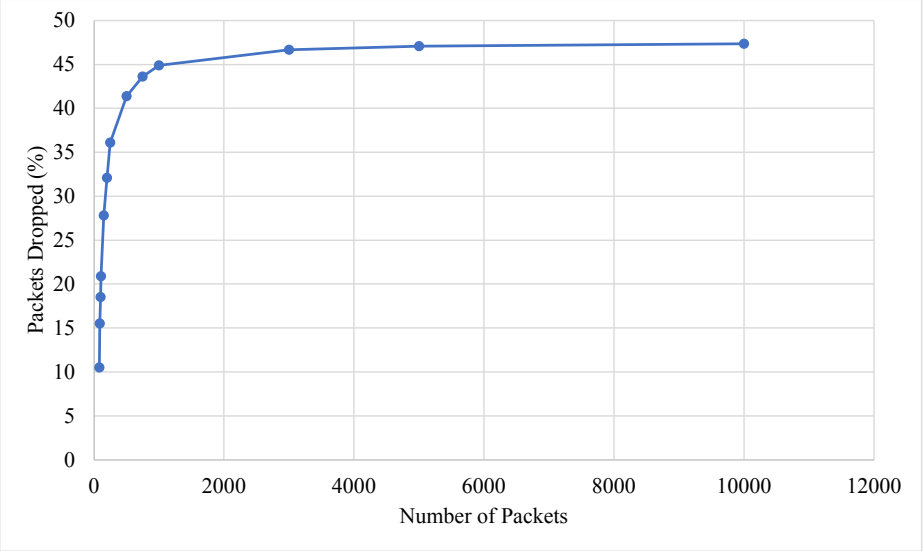

Figure 5.1: Relation of Packets Dropped vs. Number of Packets

Table 5.2: Packet Drop for Different Number of Packets & Sizes

Number of Packets	100	200	300	400	500	600	700	800	900	1000	3000	5000
Packet Size	Number of Packets Dropped											
50	0	0	0	0	0	54	148	242	336	430	2311	4192
150	0	4	91	178	265	352	439	526	614	700	2444	4186
250	0	35	116	198	279	361	442	523	605	686	2312	3941
350	0	68	145	221	298	374	451	528	604	682	2214	3745
450	9	80	152	223	295	367	439	512	583	654	2088	3523
550	12	80	144	216	284	352	420	489	555	624	1986	3346
650	17	81	144	208	272	336	399	463	527	591	1860	3131
750	16	78	139	199	259	319	380	440	501	561	1765	2972
850	20	76	133	188	244	299	355	411	467	523	1638	2750
950	19	71	124	176	230	283	335	388	440	493	1542	2593
1050	18	64	114	163	210	258	307	353	400	449	1399	2354

had different best IPG values. Table 5.3 presents best IPG values for randomly chosen different packet sizes between 50 bytes to 4096 bytes. Figure 5.2 shows the plot of the best IPG values against different packet sizes. The chart shows that as the packet size is increasing, the best IPG value is decreasing. The IPG value decreases till packet size of 2200 bytes, after which, an IPG value of 1 works as the best IPG for packet size higher than 2200 bytes. For a smaller packet size, the operation needs to be done faster which is not possible. The next packets in FIFO need to wait to get it processed by Ethernet DMA. Thus, there is a packet loss for smaller packet sizes ranging from 50 bytes to 2199 bytes. This can be addressed by adjusting the IPG value as best IPG values given in the Table 5.3.

Table 5.3: Best IPG Values for Different Packet Sizes

Packet Size (Bytes)	Best IPG	IPG Time (nsec)
50	136	870.4
250	151	966.4
450	152	972.8
650	148	947.2
850	139	889.6
1248	60	384.0
1448	64	409.6
1648	35	224.0
1848	28	179.2
2048	23	147.2
2200	1	6.4
2548	1	6.4
3048	1	6.4
3548	1	6.4
4096	1	6.4

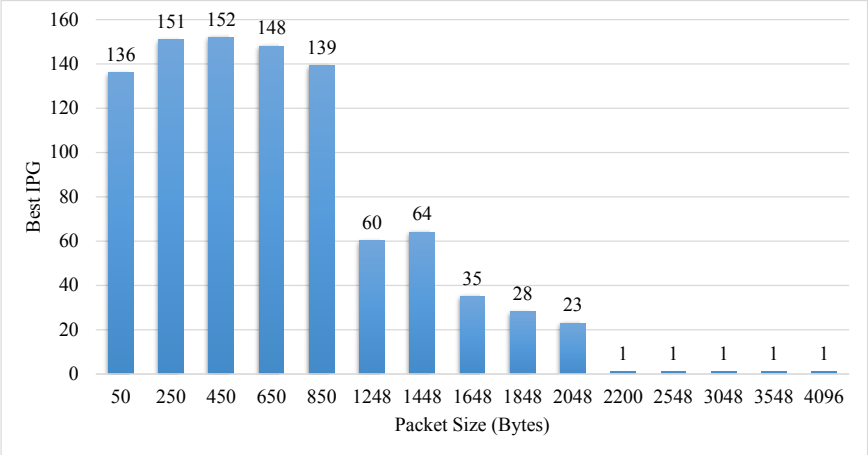

Figure 5.2: Best IPG Values for Different Packet Sizes

5.3 Ethernet DMA Datapath Performance

Once the best IPG values were measured for different packet sizes, the next phase was to measure the performance of the Ethernet DMA. Running the test case with the best IPG values ensure that there would be no loss of packets. To measure the performance of both S2MM and MM2S, EPG module was used to generate 10,000 ethernet packets to be sent to Ethernet DMA. The datapath from streaming to memory-mapped was used to write those packets to the memory. Once ethernet

packets were written to the memory, a test case was run to read 10,000 packets from memory and transmit them through memory-mapped to streaming datapath.

5.3.1 Performance for Writing Ethernet Packets

Table 5.4 presents the measurements made after processing 10000 ethernet packets of different sizes through S2MM datapath. The size of the packets were varied from 50 bytes to 4096 bytes randomly and the best IPG for the specific packet size was used to make sure that there is no packet drop. The measurements presented were taken once for each packet size and do not show the average. The performance of Ethernet DMA for writing ethernet packets (streaming to memory-mapped path) is plotted for different packet sizes in Figure 5.3. The minimum packet size is 50 bytes at which the Ethernet DMA takes 18.431 milliseconds to process 10,000 ethernet packets. For the packet size of 4 KB (4096 bytes), the Ethernet DMA takes 61.515 milliseconds to process 10,000 ethernet packets. The plot presents almost linear curve behaviour which indicates that the Ethernet DMA is functioning as it should. The performance depends on two factors, packet size and inter-packet gap. For the packet size of 1248 bytes, it takes 27.902 ms to process 10,000 packets as compared to packet size of 850 bytes which took 31.744 ms. This indicates that higher IPG value for packet size 850 bytes impacts its performance compared with packet size of 1248 bytes. This factor account for the glitch in the curve. Thus, it can be said that IPGs do not have a linear relation to packet size which is evident from the Figure 5.2.

Table 5.4: S2MM Performance of Ethernet DMA

Packet Size (Bytes)	S2MM Time (msec)
50	18.431
250	23.678
450	27.006
650	29.695
850	31.744
1248	27.902
1448	32.894
1648	31.102
1848	33.406
2048	35.966
2200	35.582
2548	41.213
3048	49.148
3548	57.212
4096	61.515

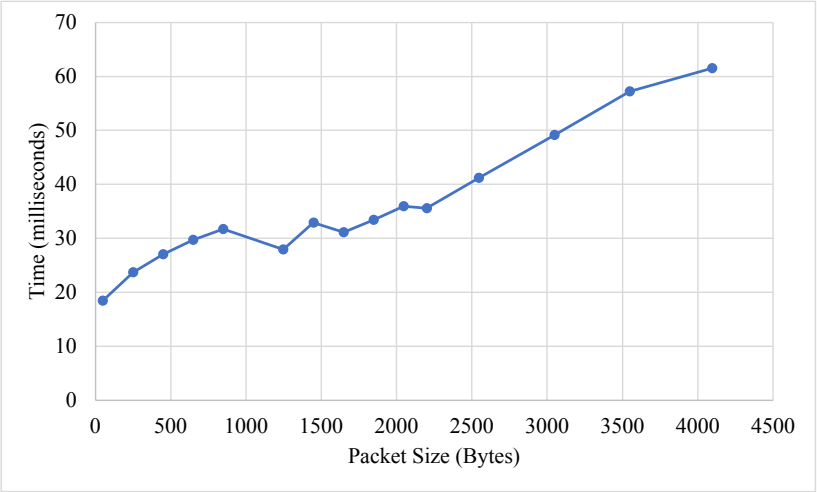

Figure 5.3: S2MM Performance

5.3.2 Performance for Reading Ethernet Packets

Table 5.5 presents the measurements made after processing 10000 ethernet packets of different sizes through MM2S datapath. The size of the packets were varied from 50 bytes to 4096 bytes randomly. The measurements presented were taken once for each packet size and do not show the average. The performance of Ethernet DMA while reading ethernet packets (memory-mapped to streaming path) is plotted for different packet sizes in Figure 5.4. For 50 bytes packet size, Ethernet DMA takes 12.753 milliseconds to read 10,000 ethernet packets and transmit them, while for a packet size of 4 KB (4096 bytes), it takes 65.563 microseconds to process 10,000 ethernet packets. The plot presents a linear curve behaviour as the packets size increases which indicates that the Ethernet DMA is performing as it should. However, from the plot it can be observed that for the packet sizes which are multiples of 1 KB (1024, 2048), there is a dip in the curve due to the boundary condition that we have in the SGDMA IP.

5.4 Effect of IPG on performance

Once the performance for writing and reading ethernet packets was measured, the next step was to measure the effect of IPG on the performance. For that purpose, the tests were run to measure the performance at IPG values higher than the best IPG values already calculated. A number of different packet sizes were selected to run the test case. The performance was measured to run tests by increasing the IPG values beyond the best values achieved for different packet sizes. Again, 10,000 ethernet packets were sent to write to the memory via streaming to memory mapped path. Table 5.6 shows the performance measurements with higher IPGs than the best IPG for different packet sizes. As it can be seen from the table,

Table 5.5: MM2S Performance of Ethernet DMA

Packet Size (Bytes)	MM2S Time (msec)
50	12.753
250	23.531
450	32.083
650	41.737
850	45.8
1024	16.402
1248	46.23
1448	49.61
1648	49.89
1848	51.72
2048	32.796
2200	53.493
2548	55.438
3048	57.405
3548	60.111
4096	65.563

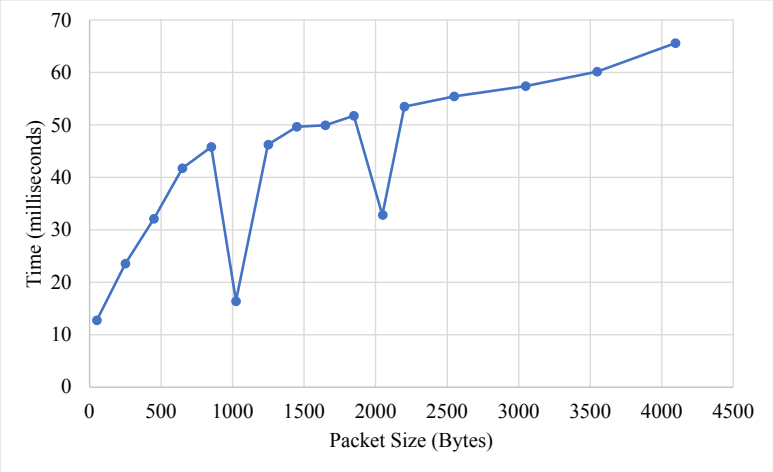

Figure 5.4: MM2S Performance

the time elapsed to write 10,000 ethernet packets increases as the IPG value is increased. Let's take a look at the performance numbers for packet size of 1248 bytes. It can be seen that for the measurements taken at IPG values higher than the best IPG (60), the time elapsed increases from 27.902 milliseconds to 33.022 milliseconds. Similar behaviour can be observed for all other packet sizes.
The resulting graph from Table 5.6 showing the effect of IPGs on performance is

plotted in Figure 5.5. It can be observed from the plots, there is a linear relationship between the IPG and time for different packet sizes. This shows that as we increase the IPG value beyond the best IPG value for a particular packet size, the performance gets affected. This may help us in saving the loss of packets by sending packets at higher inter-packet gaps, but it affects the performance adversely.

Table 5.6: Performance with Higher IPGs for Different Packet Sizes

		Time Elapsed (milliseconds)				
Packet Size	Best IPG	Best IPG	Best IPG + 10	Best IPG + 20	Best IPG + 30	Best IPG + 40
50	136	18.431	19.710	20.990	22.270	23.550
250	151	23.678	24.958	26.238	27.518	28.798
450	152	27.006	28.286	29.566	30.846	32.126
650	148	29.695	30.974	32.254	33.534	34.814
850	139	31.744	33.023	34.302	35.582	36.862
1248	60	27.902	29.182	30.462	31.742	33.022
1448	74	32.894	34.173	35.454	36.734	38.014
1648	35	31.102	32.383	33.662	34.942	36.222
1848	28	33.406	34.686	35.966	37.246	38.526
2048	23	35.966	37.246	38.526	39.805	41.085
2200	1	35.582	36.862	38.142	39.421	40.701
2548	1	41.213	42.493	43.773	45.053	46.333
3048	1	49.148	52.093	53.372	54.652	55.932
3548	1	57.212	58.492	59.772	61.052	62.332
4048	1	65.147	66.427	67.707	68.986	70.267
4096	1	65.915	67.195	68.475	69.755	71.034

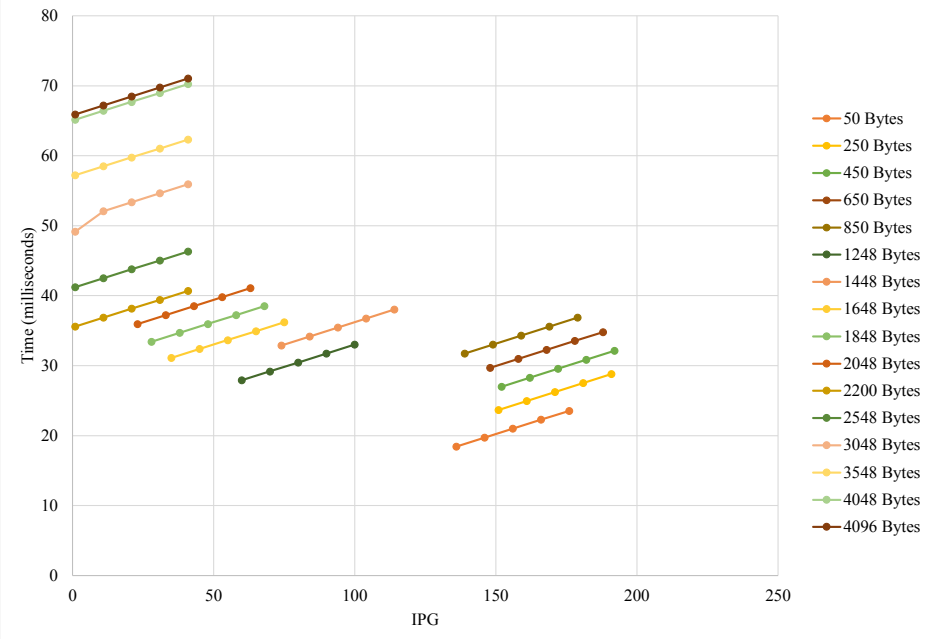

Figure 5.5: Effect of IPG on Performance

5.5 Performance for Different Number of Packets

The above mentioned performance results were measured by fixing the number of packets, that is, 10,000 ethernet packets were sent in all the tests. The performance for sending different number of packets was also measured by creating the ethernet packets in the range from 100 to 10,000 by EPG module. The packet size was fixed to 1050 bytes in all the cases.

Table 5.7: Performance for Different Number of Packets

No. of Packets	S2MM Time (ms)	MM2S Time (ms)
100	0.223	0.501
200	0.499	0.960
300	0.893	1.466
400	1.232	1.923
500	1.559	2.423
600	1.895	2.896
700	2.220	3.390
800	2.548	3.863
900	2.890	4.329
1000	3.211	4.818
3000	9.828	14.400
5000	16.446	23.996
10000	33.022	48.220

Table 5.7 presents the performance measurements for writing and reading different number of packets. As the number of packets are increased from 100 packets to 10,000 packets, it takes 0.0223 ms and 33.022 ms time to write packets, respectively. Similarly, for reading, it takes 0.501 ms to read 100 packets and 48.220 ms to read 10000 packets. The resulting plot for the S2MM path is presented in Figure 5.6 which shows the linear behaviour which is expected if the Ethernet DMA is functioning properly. The trend for reading different number of packets has been plotted in Figure 5.7. Similar to S2MM, the linear curve indicates that Ethernet DMA is behaving as it should.

Table 5.8 presents the data rates for S2MM and MM2S paths for different number of packets. The resulting plot for S2MM path, i.e. writing packets is shown in the Figure 5.8. The plot shows that there is an inverse relation to the data rate as the number of packets increase. This means that when small number of packets are sent, they are processed faster. When more packets are sent, they take more time to process, resulting in lower data rates. The trend of data rates for MM2S, i.e. reading different number of packets is shown in the Figure 5.9. Similar to data rate for writing, the data rate for reading also decreases as the number of packets is increased as we are processing more packets and they take more time.

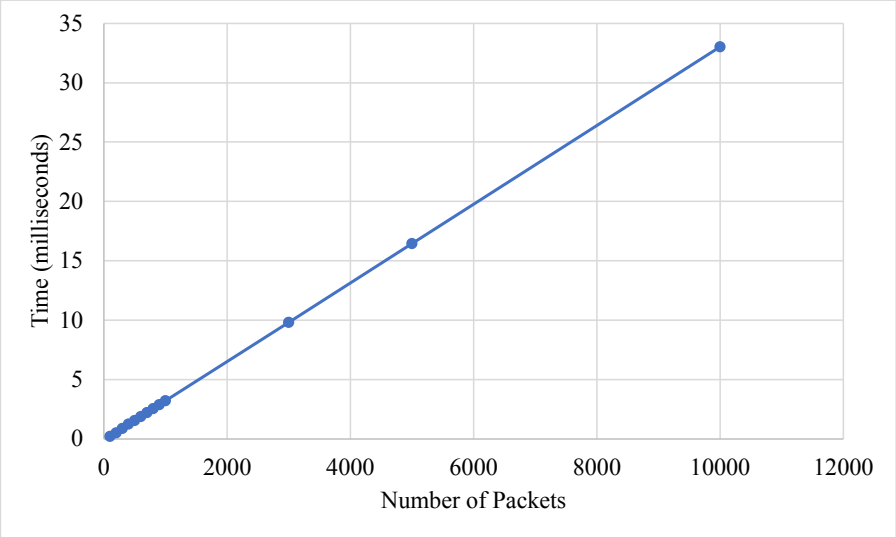

Figure 5.6: S2MM Performance for Different Number of Packets

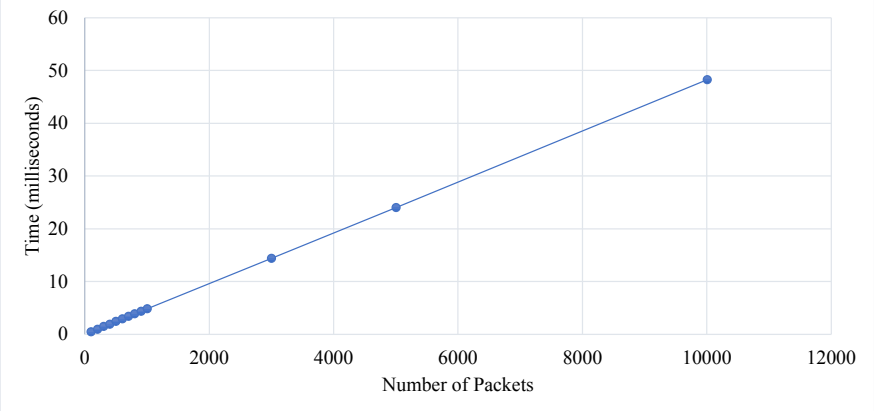

Figure 5.7: MM2S Performance for Different Number of Packets

Table 5.8: Data Rates for Different Number of Packets

No. of Packets	S2MM Data Rate (Mbps)	MM2S Data Rate (Mbps)
100	0.3766	0.168
200	8.41×10^{-2}	0.04375
300	3.13×10^{-2}	1.909×10^{-2}
400	1.70×10^{-2}	1.092×10^{-2}
500	1.07×10^{-2}	6.933×10^{-3}
600	7.38×10^{-3}	4.834×10^{-3}
700	5.40×10^{-3}	3.539×10^{-3}
800	4.12×10^{-3}	2.718×10^{-3}
900	3.23×10^{-3}	2.156×10^{-3}
1000	2.61×10^{-3}	1.743×10^{-3}
3000	2.85×10^{-4}	1.944×10^{-4}
5000	1.021×10^{-4}	7.000×10^{-5}
10000	2.54×10^{-5}	1.742×10^{-5}

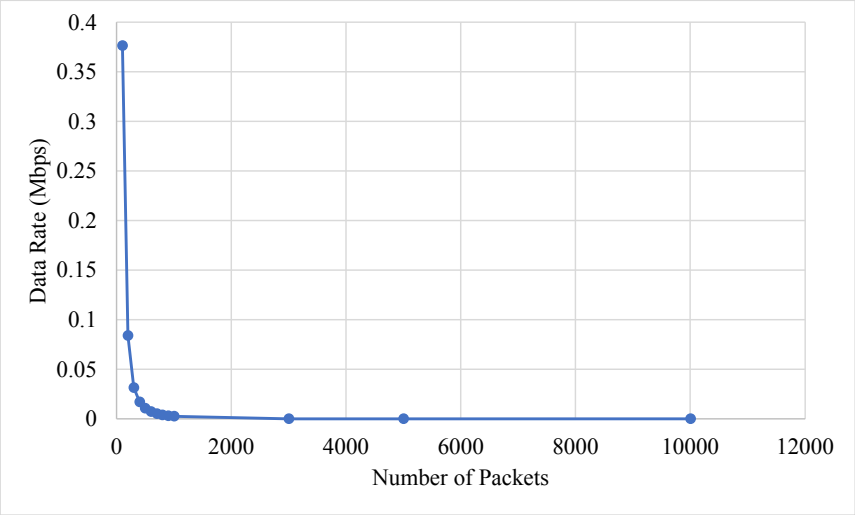

Figure 5.8: Data Rate for Writing Ethernet Packets

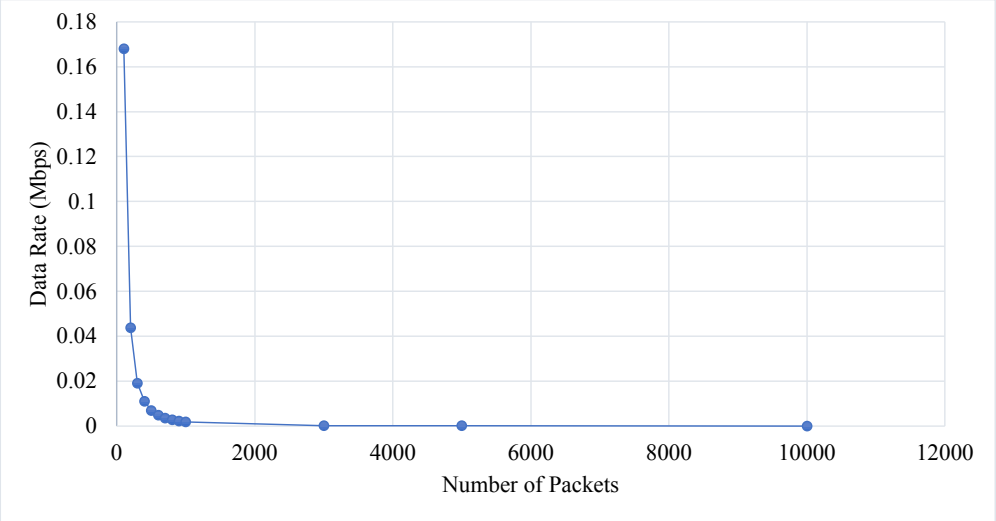

Figure 5.9: Data Rate for Reading Ethernet Packets

Chapter 6

Improvements & Future Work

The performance of Ethernet DMA datapath was measured and the results are presented in the previous chapter. The results presented clearly show that the Ethernet DMA is giving the optimal performance with the current configuration of the design IPs. A number of different factors were analyzed to optimize the design more to get improved performed. An analysis of solutions and some potential improvements are given in the following sections.

6.1 Analysis

The performance results presented in Chapter 5 were analyzed. After analyzing the measurements obtained, we could observe that there is a room for some improvements to the current design. The design of the Ethernet DMA uses SGDMA IP, which is a third-party IP where making improvements is difficult because we need to redesign the IP for it. In the current state, it is a black box, so there is not much we could do to modify the internals of the IP.

The second potential improvement that we could analyze in the design can be outside the SGDMA IP where some modifications in the design can be made to optimize it. The Ethernet DMA uses some interconnects between IPs which affect the performance of the datapath too. These interconnects have been designed and configured to suit for the specific data input modes and configurations. We can try to work on them to achieve optimization in performance of the design by implementing some modifications as discussed in the next section.

The FIFO used in the design was also analyzed to check if its size is a bottleneck to the performance. An option to increase the FIFO size can be omitted because there is already a lot of buffering in the path due to interconnects. Hence, additional buffers would not help in optimizing the design. It is concluded that increasing the FIFO size would not help in improving the performance, which is governed by the the ability of the receiver DMA.

6.2 Optimization

After the analysis of different design modules and its datapath, we come to a conclusion that there is a room for the optimization in the design to enhance its performance. Based on different aspects considered, it was decided that some changes to interconnects can be implemented to improve the performance.

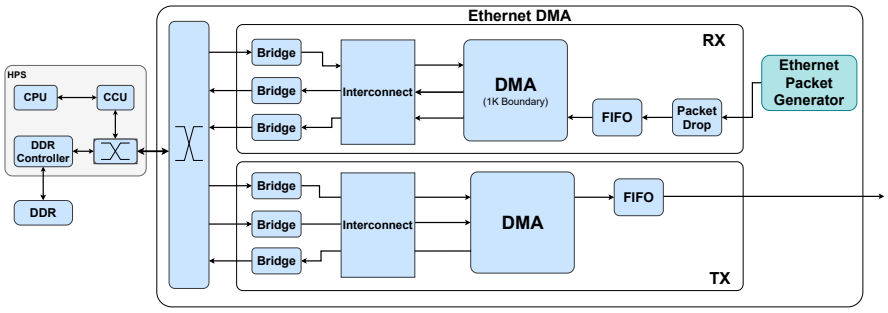

Figure 6.1: Design of Ethernet DMA Datapath

Interconnect: The system design presented in Figure 6.1 shows Ethernet DMA and its internal modules. SGDMA cores are connected to the bridges through interconnects. The SGDMA IP used from the third party company follows their own memory-mapped protocol for all the master interfaces towards the DDR. Since the end protocol at the input of the HPS is AXI, protocol conversion from third party protocol to AXI is also handled by the third party company's interconnect. The limitation of this interconnect is that it will not automatically take care of the 4K boundary crossing limitation that is part of the AXI protocol. For the AXI protocol, burst can not cross 4KB boundary, i.e. 4KB address boundary should not cross in AXI burst transaction. For example, if we have an address 0x0 and burst size of 4200 bytes, it should be split into two parts; 4096 bytes and the second part containing remaining bytes. Third party company's protocol does not have this limitation and interconnect does not take this boundary condition automatically. It needs to be made sure that 4K boundary condition should not be violated. For that purpose, in the IP settings are changed to comply to this boundary condition. To get around this problem, the SGDMA IP settings are chosen in such a way to always initiate a burst transfer at the 1k boundary crossing. For a given DMA transfer request, the IP will initiate single burst size transfers until it reaches the 1k boundary crossing and then initiate a burst transfer of size > 1, if possible, for the rest of the data transfer. If the interconnect does not have this 4KB boundary limitation, then we would have the best possible performance.

The second thing we can investigate is the 1k boundary setting of the IP. If the characteristics of the traffic are known, it should be possible to optimize this setting even further to reduce the number of single burst size transactions that are originated, so that they can be bundled into a single burst transfer.

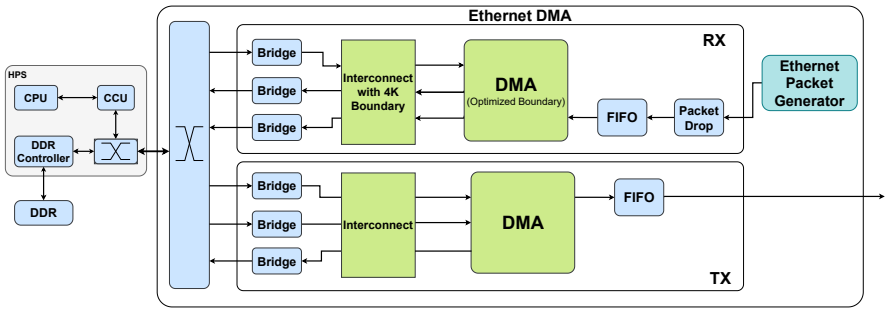

Figure 6.2: Proposed Improvement to Design

Therefore, the design can be optimized as shown in Figure 6.2 by changing configuration for the DMA IP and interconnects. The DMA module highlighted in green color is configured with optimized boundary and the interconnect too can be configured as highlighted in green. These proposed solutions can improve the performance of Ethernet DMA's datapath.

Chapter 7
Conclusion

The main objective of this thesis project was to measure and analyze the performance of Ethernet DMA controller datapath. The performance was measured and it was found that the Ethernet DMA was functioning to its optimal level, but still there was a room for some improvement. Based on the results obtained and the design analysis, some improvements were proposed to optimize the performance even more.

In the beginning, an ethernet packet generator was integrated to the design to generate ethernet packets. The packets generated were sent to measure the performance of streaming to memory-mapped path. Ethernet packets were dropped as the design was working with the inter-packet gap value 1, with which it is not possible to measure the Ethernet DMA datapath performance properly. So, the first task was to identify the best IPG values at which different packet sizes could be transferred without any packet loss.

Performance measurement using software timestamps would not produce reliable values, so the measurements were taken directly from hardware by taking a pin out from the design. The oscilloscope was used to measure the timings.

From the results obtained, we could see that there is a linear relationship between the ethernet packets and the timing performance, thus, there is less room for improvements. From the oscilloscope graphs, it can observed that the Ethernet DMA is waiting for some time and it is not processing packets continuously due to the packing of data. Interconnects have been used which can not handle burst of data coming through Ethernet DMA as they have been designed to process a certain number of bytes. This is an area where there is a room for optimizing the Ethernet DMA for better performance.

One more observation is that the packet sizes we used vary from 50 byte to 4096 bytes (4 KB), but in real scenario, the packets are only in the size ranging from 2048 (2 KB) to 4096 bytes (4 KB). Therefore, the design can be optimized based on the statistics of the incoming packet sizes.

Bibliography

[1] "Understanding 5G: Perspectives on future technological advancements in mobile, Whitepaper". In: *GSMA Intelligence* (2014).

[2] A. Gupta and R. K. Jha. "A Survey of 5G Network: Architecture and Emerging Technologies". In: *IEEE Access* 3 (2015), pp. 1206–1232. DOI: 10.1109/ACCESS.2015.2461602.

[3] C.E. Spurgeon. *Ethernet: The Definitive Guide: The Definitive Guide*. O'Reilly Media, 2000. ISBN: 9780596552824. URL: https://books.google.se/books?id=MRChaUQr0QOC.

[4] A. F. Harvey. "DMA Fundamentals on Various PC Platforms". In: 1994.

[5] A. Aljumah and M. Ahmed. "Design of High Speed Data Transfer Direct Memory Access Controller for System on Chip Based Embedded Products". In: *Journal of Applied Sciences* 15 (2015), pp. 576–581.

[6] Guoliang Ma and Hu He. "Design and implementation of an advanced DMA controller on AMBA-based SoC". In: *2009 IEEE 8th International Conference on ASIC*. 2009, pp. 419–422. DOI: 10.1109/ASICON.2009.5351258.

[7] Yingxiao Zhao et al. "Research on FPGA timing optimization methods with large on-chip memory resource utilization in PCIe DMA". In: *2016 CIE International Conference on Radar (RADAR)*. 2016, pp. 1–4. DOI: 10.1109/RADAR.2016.8059429.

[8] L. Rota et al. "A new DMA PCIe architecture for Gigabyte data transmission". In: *2014 19th IEEE-NPSS Real Time Conference*. 2014, pp. 1–2. DOI: 10.1109/RTC.2014.7097561.

[9] Kamlendra Chandra et al. “Design of PCIe-DMA bridge interface for High Speed Ethernet Applications”. In: *2019 Second International Conference on Advanced Computational and Communication Paradigms (ICACCP)*. 2019, pp. 1–5. DOI: 10.1109/ICACCP.2019.8882929.

[10] Andrew Bean, Nachiket Kapre, and Peter Cheung. “G-DMA: improving memory access performance for hardware accelerated sparse graph computation”. In: *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2015, pp. 1–6. DOI: 10.1109/ReConFig.2015.7393317.

[11] Y. Li and T.U. Press. *Computer Principles and Design in Verilog HDL*. Wiley, 2015. ISBN: 9781118841099. URL: https://books.google.se/books?id=wroyCgAAQBAJ.

[12] *Configurable, Multi-channel, WISHBONE-compliant DMA Controller*. URL: https://www.latticesemi.com/view_document?document_id=24824.

[13] *AXI DMA v7.1 LogiCORE IP Product Guide*. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf.

[14] *Scatter-Gather DMA Controller Core*. URL: https://www.intel.co.jp/content/dam/altera-www/global/ja_JP/pdfs/literature/hb/nios2/qts_qii55003.pdf.

[15] Jorg Sommer et al. “Ethernet – A Survey on its Fields of Application”. In: *IEEE Communications Surveys Tutorials* 12.2 (2010), pp. 263–284. DOI: 10.1109/SURV.2010.021110.00086.

Printed by Tryckeriet i E-huset, Lund 2021

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2021-833
http://www.eit.lth.se