Performance Optimization of a NIC for SoC Prototyping

Dissertation submitted in the fulfillment of

Master of Technology (M.Tech)
by

Bhupendra Sahu
Roll No. 22M1170

under the supervision of
Prof. Madhav P. Desai

Department of Electrical Engineering

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY
Powai, Mumbai - 400076
June 2025

Dissertation Approval

This dissertation entitled

Performance Optimization of a NIC for SoC Prototyping

by

Mr. Bhupendra Sahu
Roll No. 22M1170

is approved for the degree of
Master of Technology in Electrical Engineering

....	....
Prof. Madhav P. Desai	Prof. Virendra Singh
(Supervisor)	(Examiner)
....	....
Prof. Virendra Singh	Prof. Sachin B. Patkar
(Chairman)	(Examiner)

Date: 2025-06-30
Place: IIT Bombay

Declaration

I hereby declare that this written submission represents my own ideas, expressed in my own words. Where the ideas or words of others have been included, they have been appropriately cited and referenced. I affirm that I have adhered to the principles of academic honesty and integrity, and that I have not misrepresented, fabricated, or falsified any idea, data, fact, or source in this submission.

I understand that any violation of the above principles may result in disciplinary action by the institute, and may also lead to legal or penal consequences from the original sources if proper citation or necessary permissions have not been obtained.

....

Mr. Bhupendra Sahu
Roll No. 22M1170

Date : 2025-06-30

Acknowledgment

I would like to express my heartfelt gratitude to **Prof. Madhav P. Desai** for giving me the opportunity to work on this project and for his invaluable guidance throughout the course of this research. His insights and suggestions have greatly enhanced my understanding of the subject and have been instrumental in shaping this work.

I am also deeply thankful to my family and friends for their unwavering encouragement and support, without which this journey would not have been possible.

Abstract

This thesis focuses on optimizing the performance of a custom Network Interface Controller (NIC) within a Network-Centric System-on-Chip (N-SoC) platform built around the 32-bit AJIT processor and high-capacity DRAM. The platform, implemented on a Xilinx KC705 FPGA, builds on prior work by Harshad Ugale and Siddhant Singh Tomar, incorporating enhancements in NIC design and memory architecture. However, despite these improvements, system throughput remained under 10% of the Ethernet link speed.

To address this limitation, the primary objective was to achieve at least 60% of the link speed through targeted NIC optimizations. Key enhancements included transitioning to shared local packet memory, moving queue management to register-based FIFOs, and increasing MAC FIFO capacity from 4 KB to 16 KB. Multiple packet scheduling strategies were explored, culminating in a custom reliable protocol (`NIC_1.2`) featuring acknowledgments and retransmissions for robust data handling.

Performance was validated using Ping-based round-trip and throughput tests involving raw packets, files, and image transfers. Results demonstrated significant gains in throughput and reliability. A 32-bit hardware counter was also integrated to support precise timing, enabling future synchronization between the NIC and processor using the Precision Time Protocol (PTP).

This work lays the foundation for a scalable, high-performance NIC architecture capable of supporting future high-throughput applications.

Contents

List of Figures ix

List of Tables x

1 Introduction 1

1.1 Background and Motivation 1
1.2 Objectives 1
1.3 Introduction to the Network-Centric SoC 2
1.4 Packet Buffer Management Scheme 2
1.5 Organization of the Thesis 3

2 Architecture and Optimization of the Baseline SoC Platform 4

2.1 Introduction 4
2.2 Single-Board Computer (SBC) Architecture 5
2.2.1 SBC Core 5
2.2.1.1 AJIT (1x1x32) Processor Subsystem 6
2.2.1.2 NIC_1.1 Subsystem 7
2.2.1.3 Bus Complex: ACB and AFB 8
2.2.2 SBC Periphery 9
2.2.2.1 Xilinx Tri-Mode Ethernet MAC (TEMAC) IP 9
2.2.2.2 Xilinx MIG DRAM Controller IP 9
2.2.3 Clock Domains 10
2.3 System Organization: Processor and **NIC_1.1** Packet Processing 10
2.3.1 Packet Reception 10
2.3.2 Packet Transmission 11
2.3.3 Unified Buffer Queue Design 11
2.4 Performance Evaluation of the Baseline SoC 11
2.4.1 Average Round-Trip Time (RTT) 11

2.4.2 Raw Ethernet Throughput 12
2.5 Motivation for Optimization 12
2.6 Optimization A: Fast Local Packet Memory Shared Between **NIC_1.1** and Processor 13
2.7 Optimization B: Shared Inclusive Write-Back L2 Cache for **NIC_1.1** and Processor 14
2.8 Summary and Comparison of Baseline and Optimized Architectures . . 15
2.8.1 Round-Trip Time (RTT) Comparison 15
2.8.2 Raw Ethernet Throughput Comparison 15
2.9 Preliminary Performance Verification of Processor and **NIC** 16

3 Optimization 1: Relocation of Queue Structures to NIC_1.2 Hardware 17
3.1 Introduction 17
3.2 Queue Structure Relocation 17
3.2.1 Baseline Architecture Limitations 17
3.2.2 NIC_1.2 Architecture 18
3.2.3 On-Chip Queue Architecture and Management 18
3.2.4 Concurrency and Synchronization 19
3.2.5 Queue Operations and Register Interface 20
3.2.6 Register Map Overview 20
3.3 Simulation and Testing Infrastructure 21
3.3.1 Hardware Test Modules 21
3.3.2 C-Based Simulation Testbench 21
3.3.3 Coverage and Validation Scope 22
3.4 NIC_1.2 Test Modes 22
3.4.1 Normal Mode 23
3.4.2 NIC Loopback Mode 23
3.4.3 MAC Loopback Mode 23
3.4.4 Timing Breakdown Summary 23
3.5 Simulation Timing Results 24
3.6 FPGA-Based Testing with AJIT Processor 24
3.6.1 Normal Mode (`NORMAL_MODE`) 24
3.6.2 NIC Loopback Mode (`NIC_LOOPBACK`) 24
3.6.3 MAC Loopback Mode (`MAC_LOOPBACK`) 25
3.6.4 Summary 25

4 Optimization 2: Increasing MAC FIFO Depths (NIC_1.2) 26
4.1 Introduction 26
4.2 Increasing MAC FIFO Depths in NIC_1.2 26
4.2.1 Baseline Configuration 26
4.2.2 Design Modifications in NIC_1.2 27
4.2.3 FIFO Implementation Details 27
4.2.4 VHDL Wrapper and Parameterization 28
4.3 Summary 28

5 Optimization 3: Packet Scheduling Strategies (NIC_1.2) 29
5.1 Introduction 29
5.2 Packet Scheduling Strategies in NIC_1.2 29
5.2.1 Motivation 29
5.2.2 Implemented Scheduling Schemes 30
5.2.2.1 Scheme 1: FIFO Scheduling with Inter-Packet Delay . . 30
5.2.2.2 Scheme 2: Burst Transmission with Controlled Delay . . 30
5.2.2.3 Scheme 3: Continuous Transmission with FIFO Monitoring 30
5.2.2.4 Scheme 4: Reliable Packet Scheduling with Acknowledgment Protocol 30
5.2.3 Advantages of the Reliable Protocol 32
5.3 Summary 32

6 Evaluation and Testing of NIC_1.2 33
6.1 Overview 33
6.2 Simulation-Based Evaluation 33
6.2.1 RTT Measurements 33
6.2.2 Throughput Comparison 34
6.3 FPGA-Based Evaluation 34
6.3.1 RTT Measurements on Hardware 34
6.3.2 Throughput on FPGA 35
6.4 Image Transfer Performance 37
6.5 Summary 38

7 Conclusion & Future Work 39
7.1 Conclusion 39
7.2 Summary of Findings 42

7.3 Future Work 42

Appendices **44**

Appendix A Standard NIC_1.2 (Network Interface Controller) Design **45**

A.1 Design Decisions 45

A.1.1 NIC_1.2–MAC Interface 46

A.1.2 NIC_1.2–Memory Interface 47

A.1.3 NIC_1.2–Processor Interface 48

A.2 Interface Data Structures 49

A.3 Network Interface Controller (NIC_1.2) 50

A.3.1 Register File Map 50

A.3.2 Receive Engine 50

A.3.3 Transmit Engine 50

A.3.4 Register Access Daemon 51

A.4 Helper Modules 54

Appendix B Tri-mode Ethernet MAC Example Design on KC705 **55**

Appendix C Transmission Delay Measurement on Host-PC to FPGA Link **58**

C.1 Experiment Overview 58

C.2 Results 59

C.3 Key Observations 60

List of Figures

2.1 Architecture of single board computer with AJIT 1x1x32 6
2.2 Generic 32-bit AJIT processor core 7
2.3 NIC Subsystem 8
2.4 ACB and AFB bus complex 8
2.5 Architecture showing the NIC and processor sharing a 256KB fast local packet memory 13
2.6 NIC and processor sharing access to main memory via an L2 cache . . . 14

3.1 Queue structures implemented as hardware FIFOs inside **NIC_1.2** 18

4.1 TEMAC IP with Expanded RX/TX FIFOs (16kB each) in NIC_1.2 27

5.1 NIC_1.2 Implementation of Scheme 4: Acknowledgment-Based Reliable Packet Scheduling 31

7.1 RTT Comparison between NIC_1.1 and NIC_1.2 (Baseline and Optimization A) 40
7.2 Throughput Comparison between NIC_1.1 and NIC_1.2 (Baseline and Optimization A, with 16-kB MAC FIFO) 41

A.1 Top level with Interfaces for NIC_1.2 46

B.1 Block diagram of the Tri-mode MAC IP example design [1]. 56

List of Tables

2.1 Average Round-Trip Time (RTT) in microseconds for Various Payload Sizes 15
2.2 Raw Ethernet Throughput (Mbps) for Various Payload Sizes 16

6.1 RTT Comparison Across Simulation Modes (µs) 34
6.2 Simulation Throughput for 1024-byte Packet (Mbps) 34
6.3 RTT in MAC Loopback Mode (NIC_1.2 on FPGA, µs) 35
6.4 RTT Comparison: Baseline vs Opt A (Fast Local Memory) (Normal Mode, µs) 35
6.5 MAC Loopback Throughput (FPGA, Mbps) 36
6.6 Throughput with 16-kB MAC FIFOs (FPGA, Mbps) — Final Optimized Design 36
6.7 Image Transfer Performance for NIC_1.2 with Fast Local Memory — Small Image (Measured on FPGA) 37
6.8 Image Transfer Performance for NIC_1.2 with Fast Local Memory — Large Image (Measured on FPGA) 37

7.1 Average RTT (μs) Comparison: NIC_1.1 vs NIC_1.2 40
7.2 Throughput (Mbps) Comparison: NIC_1.1 vs NIC_1.2 41

A.1 NIC_1.2–MAC interface description 46
A.2 NIC_1.2 - Memory interface description 47
A.3 Memory–NIC_1.2 interface description 47
A.4 Processor to NIC_1.2 interface (AFB request) signal mapping 48
A.5 NIC_1.2 to Processor interface (AFB response) signal mapping 48
A.6 NIC_1.2 Control and Status Registers Map 51
A.7 NIC_1.2 Queue Registers Map 52
A.8 NIC_1.2 to Register File interface description 53
A.9 Register File to NIC_1.2 interface description 53

B.1 Average RTT for Various Payload Lengths (TEMAC Example Design) . . 56
B.2 Raw Ethernet Throughput (TEMAC Example Design) 57

C.1 Requested vs Actual Delays Using `time.sleep()` 59

Chapter 1

Introduction

1.1 Background and Motivation

This thesis focuses on optimizing the performance of a custom Network Interface Controller (NIC) within a Network-Centric System-on-Chip (N-SoC) platform built around the 32-bit AJIT processor and high-capacity DRAM. The platform, implemented on a Xilinx KC705 FPGA, builds on prior work by Harshad Ugale and Siddhant Singh Tomar [2], incorporating enhancements in NIC design and memory architecture. However, despite these improvements, system throughput remained under 10% of the Ethernet link speed.

1.2 Objectives

This thesis aims to address the throughput limitations by focusing on the following key objectives:

- Achieve NIC throughput of at least 60% of the link speed through performance optimizations.
- Develop a reliable and efficient scheme for packet transmission and reception to ensure data integrity and system stability.
- Build a scalable NIC platform capable of supporting future enhancements for higher data rates and advanced networking features.

1.3 Introduction to the Network-Centric SoC

The proposed Network-Centric System-on-Chip (N-SoC) integrates a 32-bit AJIT processor [3], a custom high-performance Network Interface Controller (NIC), and a DRAM-based memory subsystem. Implemented on a Xilinx KC705 FPGA, the system targets a NIC throughput of approximately 600 Mbps. Key architectural enhancements include low-latency access through optimized local memory, enlarged FIFOs within the Media Access Controller (MAC), and hardware-managed buffer pointer queues to streamline packet handling and reduce software overhead.

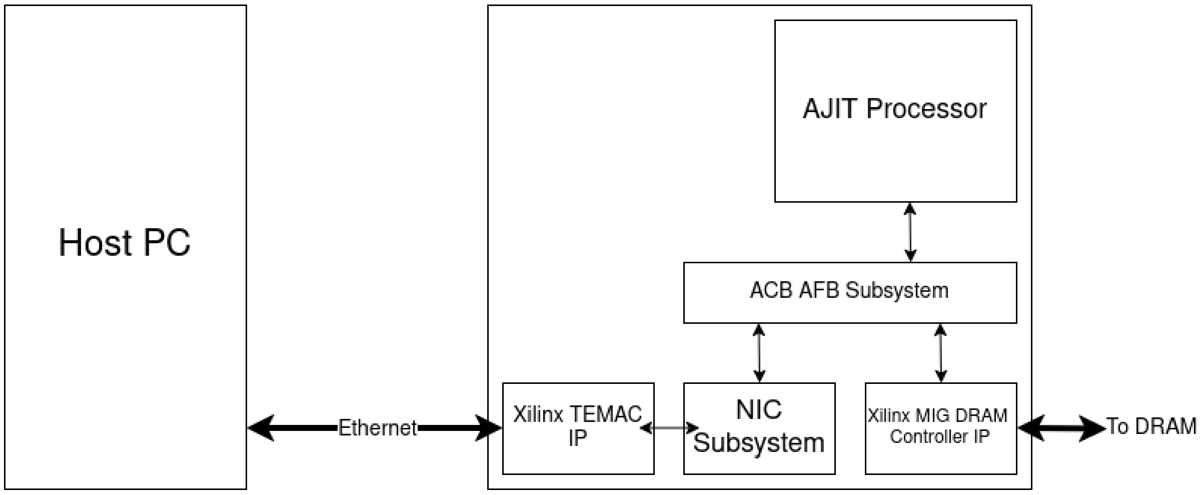

Figure 1.1: N-SoC platform with AJIT processor and custom NIC

1.4 Packet Buffer Management Scheme

Efficient packet buffer management is vital for sustaining high throughput and smooth data exchange between the Network Interface Card (NIC) and the processor. This thesis utilizes a hardware-managed circular buffer pointer system involving three primary queues and dedicated polling engines, structured as follows:

1. **Free Queue:** This queue holds packet pointers to available memory buffers. The NIC's *Receive Engine* continuously polls the Free Queue, popping a packet pointer whenever a new packet is expected. Upon receiving a packet, the Receive Engine writes the data into the corresponding memory buffer and pushes the packet pointer onto the Receive Queue.

2. **Receive Queue:** The processor continuously polls the Receive Queue for packet pointers indicating newly received packets. It pops a pointer from this queue,

reads and processes the packet data in memory, writes any updates back to the buffer, and pushes the packet pointer to the Transmit Queue.

3. **Transmit Queue:** The NIC's *Transmit Engine* continuously polls the Transmit Queue, popping packet pointers for packets ready to be sent. It reads the packet data from memory, transmits the packet over the network, and then returns the packet pointer back to the Free Queue for reuse.

In this scheme, the NIC operates two polling daemons: the Receive Engine polls the Free Queue for incoming packets, while the Transmit Engine polls the Transmit Queue for outgoing packets. Concurrently, the processor polls the Receive Queue to process incoming packet data. This circular buffer pointer management maximizes memory reuse and maintains continuous packet flow with minimal software overhead.

1.5 Organization of the Thesis

- **Chapter 2** introduces the baseline N-SoC platform, covering the AJIT processor, **NIC_1.1**, memory subsystems, FPGA implementation, and test applications including Ping and raw Ethernet transfers.
- **Chapter 3** describes relocating buffer pointer queues from DRAM to on-chip register FIFOs in **NIC_1.2**, detailing the queue management design and hardware components.
- **Chapter 4** discusses increasing MAC FIFO sizes to 16 KB and evaluates its impact on burst data handling and throughput.
- **Chapter 5** explores different packet scheduling schemes tested on **NIC_1.2**, culminating in a reliable acknowledgment-based protocol to improve throughput and data integrity.
- **Chapter 6** presents performance evaluation results, including round-trip times and throughput tests for raw packets, files, and images.
- **Chapter 7** summarizes the architectural improvements, compares optimizations, and outlines future work including wider datapath NICs and enhanced synchronization.
- **Appendices** contain detailed NIC implementation data, test utilities, and supplementary experimental results.

Chapter 2

Architecture and Optimization of the Baseline SoC Platform

2.1 Introduction

This chapter presents the design and optimization of the baseline System-on-Chip (SoC) platform, which serves as a foundational single-board computer (SBC) for subsequent architectural exploration and performance evaluation. The platform integrates a 32-bit AJIT processor, a custom Network Interface Controller (**NIC_1.1**), a DRAM-based memory subsystem, and essential peripheral interfaces.

The initial version of the NIC was developed by Harshad Ugale at IIT Bombay, providing the foundational network interface architecture. Siddhant later contributed significant memory subsystem optimizations and architectural refinements, resulting in **NIC_1.1**. Building on this, the author developed **NIC_1.2** with additional performance enhancements, discussed in later chapters.

This chapter focuses on the design evolution from the original NIC to **NIC_1.1**, highlighting the architectural contributions by Harshad and the memory-related optimizations introduced by Siddhant. The NIC's interaction with the AJIT processor and memory subsystem, and its role in handling network traffic, are discussed in detail.

The complete SoC, including **NIC_1.1**, was implemented and validated on an FPGA platform. Benchmarking was performed using `ping` for round-trip time (RTT) measurement and custom throughput tests to evaluate performance under varying traffic conditions.

Two key memory subsystem optimizations introduced by Siddhant were critical to improving packet processing efficiency:

1. **Shared Fast Local Packet Memory:** A high-speed memory shared between the processor and NIC reduces latency in packet handling.
2. **L2 Cache-Based Memory Access:** The addition of an L2 cache between the processor/NIC and main DRAM improves cache hit rates and alleviates memory bottlenecks.

These enhancements form the basis for performance improvements and serve as a reference point for further architectural advancements discussed in subsequent chapters.

2.2 Single-Board Computer (SBC) Architecture

The baseline SoC platform is realized as a Single-Board Computer (SBC) and is architecturally divided into two main sections: the **SBC Core** and the **SBC Periphery**, as illustrated in Figure 2.1. This modular organization facilitates streamlined data movement and simplifies integration with external interfaces and peripherals.

- **SBC Core:** Comprises the 32-bit AJIT processor, the custom **NIC_1.1**, and the internal interconnects—namely the AJIT Core Bus (ACB) and AJIT FIFO Bus (AFB).
- **SBC Periphery:** Includes peripheral and memory interface components such as the Xilinx Tri-Mode Ethernet MAC and the Xilinx MIG DRAM Controller IP cores.

2.2.1 SBC Core

The SBC Core includes the main computing and data movement components of the SoC:

1. **AJIT (1x1x32) Processor Subsystem**
2. **Custom Network Interface Controller (NIC_1.1) Subsystem**
3. **Bus Complex:** 64-bit AJIT Core Bus (ACB) and 32-bit AJIT FIFO Bus (AFB)

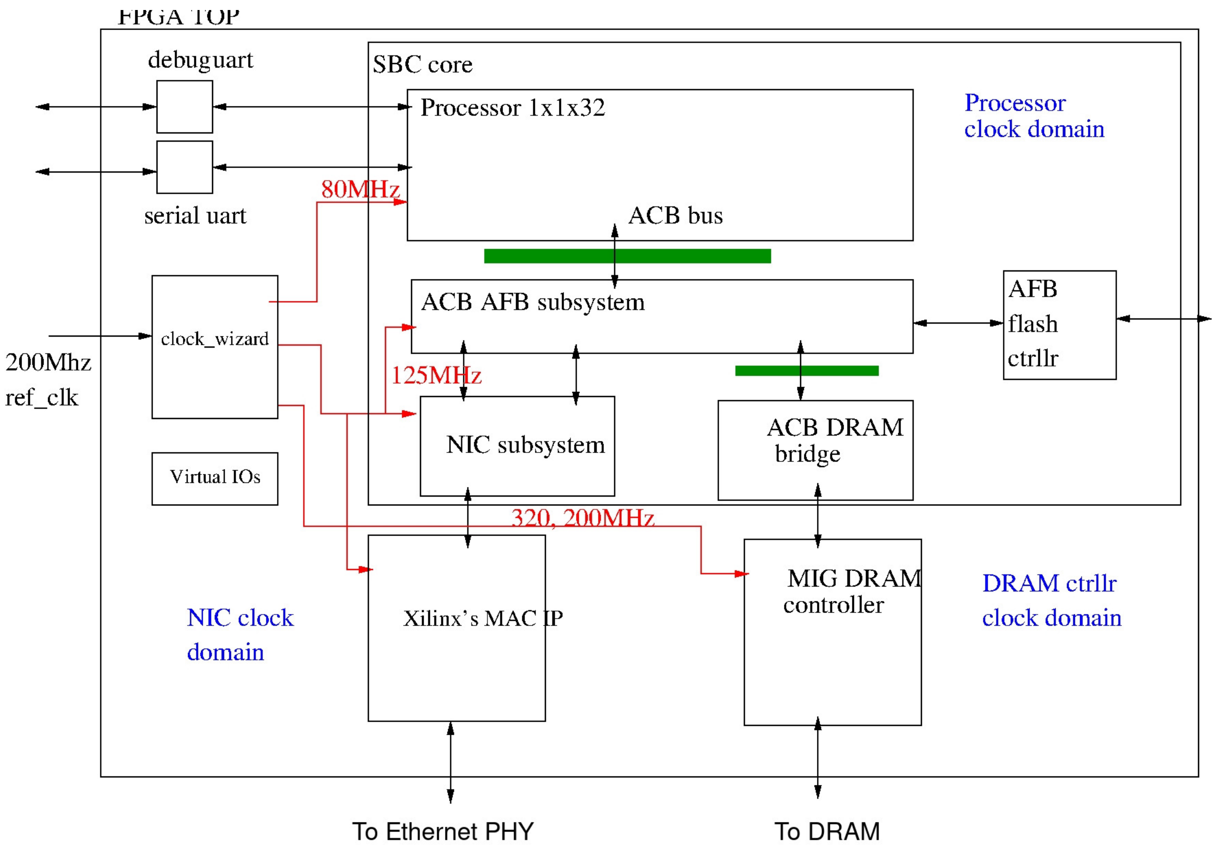

Figure 2.1: Architecture of single board computer with AJIT 1x1x32

2.2.1.1 AJIT (1x1x32) Processor Subsystem

The AJIT processor (Figure **??**) is a 32-bit, single-threaded CPU implementing the SPARC-V8 ISA [4] (IEEE Standard 1754-1996). Its key features include:

- **Instruction Cache (ICACHE):** 32 kB, direct-mapped, virtually indexed/tagged, 64-byte line size.
- **Data Cache (DCACHE):** 32 kB, direct-mapped, virtually indexed/tagged, write-through allocate policy, 64-byte line size.
- **Memory Management Unit (MMU):** Fully compliant with the SPARC-V8 architecture.
- **System Bus Interface:** A 64-bit ACB interface for accessing memory and peripherals.

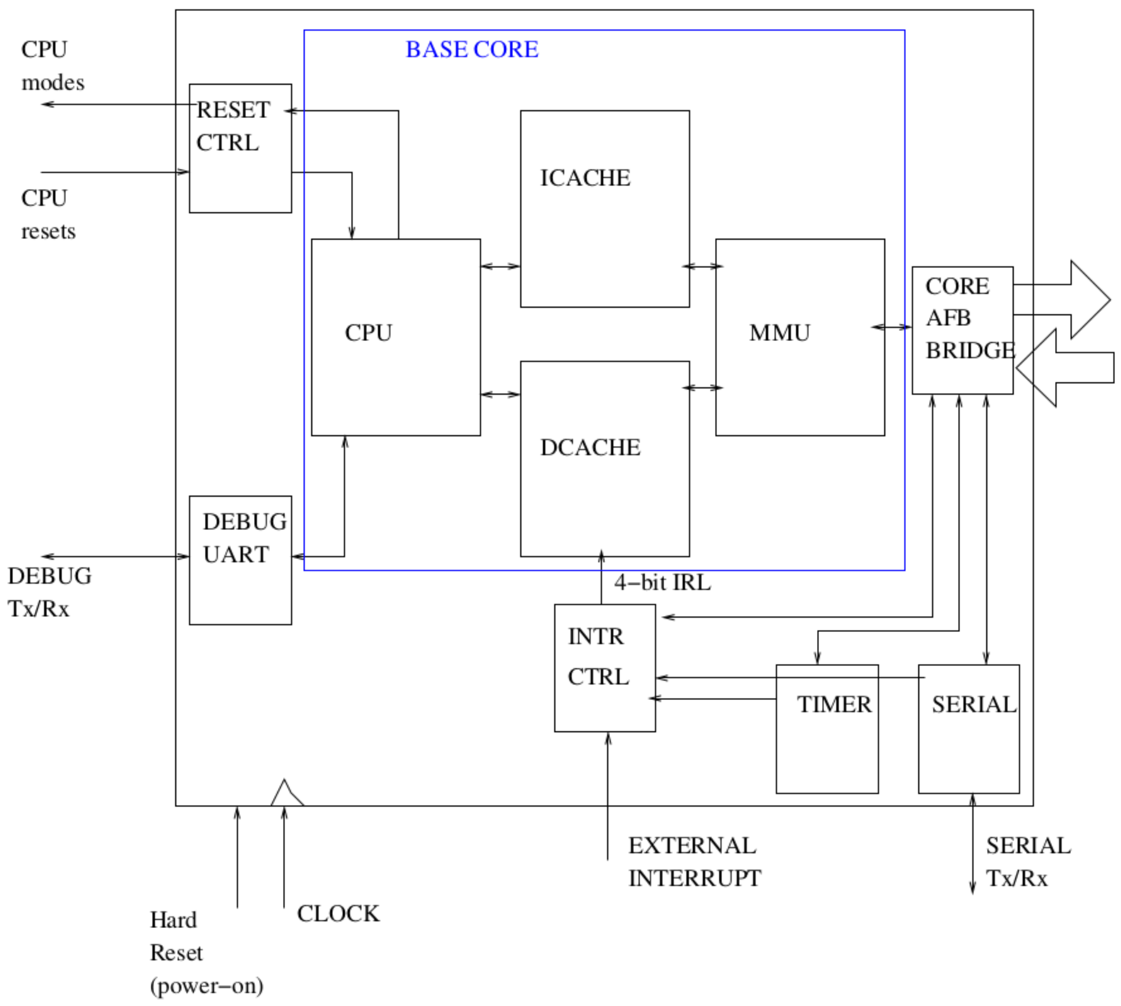

Figure 2.2: Generic 32-bit AJIT processor core

2.2.1.2 NIC_1.1 Subsystem

The custom **NIC_1.1** subsystem enables network packet transmission and reception, bridging the processor, DRAM, and Ethernet MAC. It includes the NIC core and a NIC-to-MAC bridge. Functional interfaces include:

- **Processor-to-NIC Slave Interface:** The processor configures **NIC_1.1** using memory-mapped control/status registers and queue pointer buffers.
- **NIC-to-Memory Master Interface:** Supports DMA transactions over ACB to DRAM using processor-supplied physical address queues.

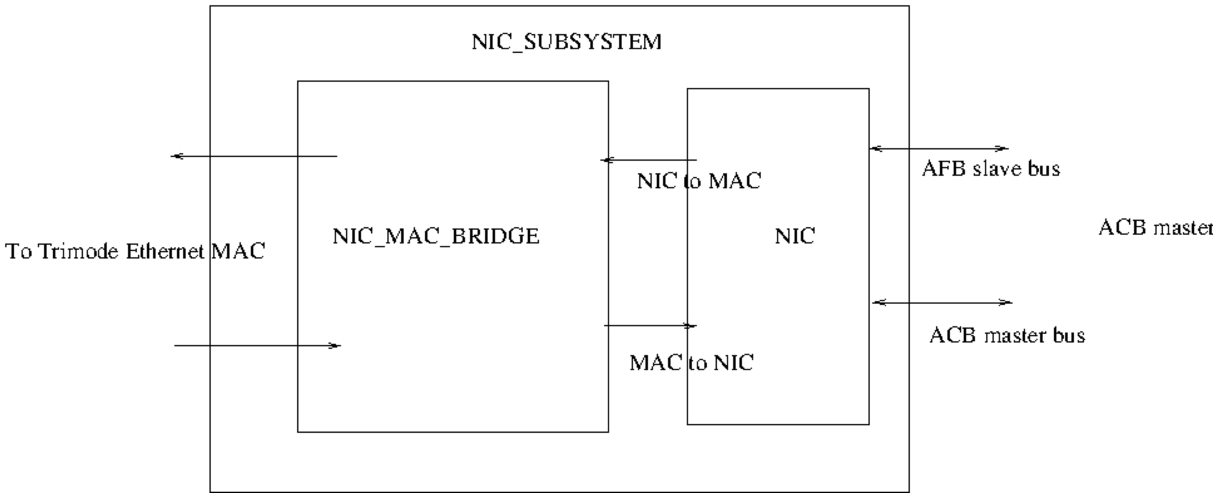

Figure 2.3: NIC Subsystem

- **NIC-to-MAC Interface:** Communicates with the Xilinx Tri-Mode Ethernet MAC IP to handle Ethernet frames.

2.2.1.3 Bus Complex: ACB and AFB

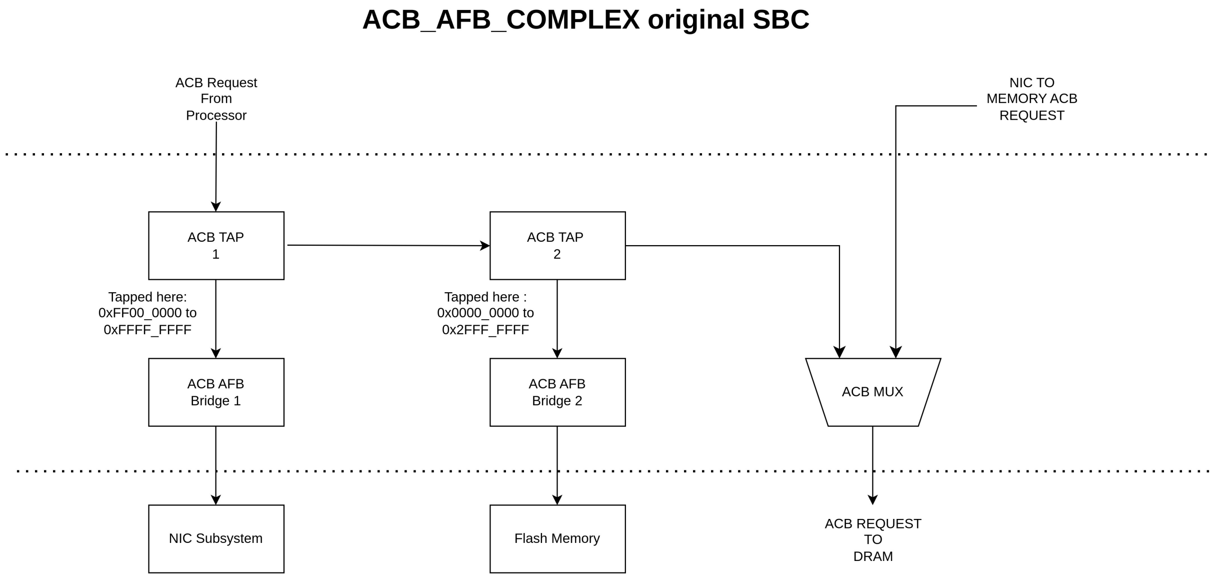
ACB_AFB_COMPLEX original SBC
ACB Request From Processor
NIC TO MEMORY ACB REQUEST
ACB TAP 1
ACB TAP 2
Tapped here: 0xFF00_0000 to 0xFFFF_FFFF
Tapped here : 0x0000_0000 to 0x2FFF_FFFF
ACB AFB Bridge 1
ACB AFB Bridge 2
ACB MUX
NIC Subsystem
Flash Memory
ACB REQUEST TO DRAM

Figure 2.4: ACB and AFB bus complex

The internal bus architecture consists of two primary buses:

- **AJIT Core Bus (ACB):** 64-bit data and 36-bit address width, used for memory-mapped transactions between the processor, NIC, and DRAM.

- **AJIT FIFO Bus (AFB):** 32-bit data and 36-bit address width, primarily used for communication with peripheral components such as the UART or debug interfaces.

Both buses implement a simple request-response protocol with a two-wire handshake mechanism to ensure reliable data transfers.

2.2.2 SBC Periphery

The SBC Periphery consists of IP cores interfacing with external memory and network hardware:

1. **Xilinx Tri-Mode Ethernet MAC (TEMAC) IP**
2. **Xilinx MIG DRAM Controller IP**

2.2.2.1 Xilinx Tri-Mode Ethernet MAC (TEMAC) IP

The TEMAC IP supports Ethernet transmission at 10/100/1000 Mbps and interfaces directly with a physical Ethernet transceiver (PHY). Its key components include:

- **TEMAC Core:** Implements Ethernet MAC layer logic.
- **AXI-Lite Control Interface:** Used for MAC and PHY configuration and management.
- **AXI-Streaming FIFOs:** Provide packet buffering between the MAC and the NIC.

2.2.2.2 Xilinx MIG DRAM Controller IP

The Xilinx Memory Interface Generator (MIG) IP provides a robust interface to external DDR2/DDR3 DRAM. It comprises:

- **Memory Controller:** Manages DRAM command sequencing and timing.
- **PHY Interface:** Ensures electrical-level interfacing with DDR memory.
- **User Logic Interface:** Allows the SoC components (e.g., processor, NIC) to initiate memory transactions.

2.2.3 Clock Domains

The SBC operates with two asynchronous clock domains:

- **Processor and DRAM:** 80 MHz
- **NIC_1.1:** 125 MHz

To maintain data integrity across clock boundaries, dual-clock asynchronous FIFOs are employed between ACB request and response channels. Clock signals are generated using Xilinx Clock Wizard IP cores.

2.3 System Organization: Processor and NIC_1.1 Packet Processing

This section describes the interaction between the processor and **NIC_1.1** for packet reception and transmission using a shared memory descriptor ring model. The design employs a single *free queue*, a *receive queue*, and a *transmit queue* to coordinate buffer ownership efficiently.

For detailed information on the NIC architecture, hardware interfaces, and register mappings, refer to Appendix A.

2.3.1 Packet Reception

Packet reception proceeds as follows:

1. **Buffer Allocation:** The processor initializes the system by populating the *free queue* with pointers to empty packet buffers in DRAM.
2. **NIC Receive Engine:** The receive engine in **NIC_1.1** continuously polls the *free queue*. Upon receiving a packet, it dequeues a buffer pointer, writes the incoming data directly into the corresponding memory buffer, and enqueues the pointer into the *receive queue*.
3. **Processor Handling:** The processor polls the *receive queue* for available buffer pointers. When one is dequeued, the processor processes the packet and then enqueues the pointer into the *transmit queue* for transmission.

2.3.2 Packet Transmission

Packet transmission involves the following steps:

1. **NIC Transmit Engine:** The transmit engine in **NIC_1.1** continuously polls the *transmit queue*. When a buffer pointer is available, it reads the packet from memory and transmits it over Ethernet.

2. **Buffer Recycling:** After transmission, the NIC returns the buffer pointer to the *free queue*, completing the buffer lifecycle.

2.3.3 Unified Buffer Queue Design

Using a single shared *free queue* for buffer management simplifies ownership transfer between the processor and **NIC_1.1**, reduces software complexity, and supports efficient cyclic reuse of packet buffers. With the NIC's receive and transmit engines and the processor each polling their respective queues, the system maintains continuous packet flow with minimal intervention.

2.4 Performance Evaluation of the Baseline SoC

The baseline SoC was implemented and validated on the Xilinx KC-705 FPGA platform. Performance was analyzed using two custom bare-metal applications that bypass the lwIP TCP/IP stack, enabling direct evaluation of the processor and custom **NIC_1.1** data path performance via raw Ethernet communication.

A Python-based test harness on a host PC generated and received raw Ethernet frames over a 1 Gbps link connected to the FPGA. Key metrics such as average round-trip time (RTT) and data throughput were measured under controlled conditions.

2.4.1 Average Round-Trip Time (RTT)

RTT was measured by sending custom raw Ethernet frames from the host to the FPGA SoC. Upon reception, **NIC_1.1** signals the processor, which loops the data back, creating an application-level echo. The processor ran at 80 MHz and the **NIC_1.1** at 125 MHz. RTT measurements were taken across various packet sizes to analyze latency effects.

2.4.2 Raw Ethernet Throughput

Throughput testing involved a unidirectional stream where the host continuously sent Ethernet frames to the FPGA, which immediately returned them. The host recorded the rate of successfully echoed frames to calculate throughput. The test flow included:

- Host PC generating raw Ethernet packets via Python scripts.
- **NIC_1.1** receiving packets and triggering processor interrupts.
- Processor promptly forwarding packets back through **NIC_1.1**.
- Host recording RTT and throughput data.

This setup captures the practical data handling performance including buffer management and coordination overhead.

2.5 Motivation for Optimization

While the baseline SoC handles raw Ethernet packets adequately, its performance is constrained by memory latency and synchronization overhead. Descriptor rings and packet buffers reside in DRAM, resulting in increased access times that limit latency and throughput.

To address these bottlenecks, two architectural optimizations were proposed to reduce memory latency and enhance communication efficiency between the processor and **NIC_1.1**:

- **Optimization A:** Introduction of a high-speed shared local memory for descriptor rings and packet buffers.
- **Optimization B:** Deployment of an inclusive write-back L2 cache shared by both **NIC_1.1** and processor on the path to DRAM.

The subsequent sections provide detailed descriptions and evaluations of these optimizations.

2.6 Optimization A: Fast Local Packet Memory Shared Between NIC_1.1 and Processor

In this optimization, the descriptor rings and packet buffers are relocated from off-chip DRAM to a fast on-chip memory block that is directly accessible by both the processor and **NIC_1.1**. To ensure consistency and avoid stale data issues, this memory is marked as non-cacheable.

Placing critical data structures in this low-latency shared memory significantly reduces access times for both entities. The result is improved responsiveness in buffer management and higher throughput, as the shared memory provides a direct, high-bandwidth communication path without DRAM-induced latency.

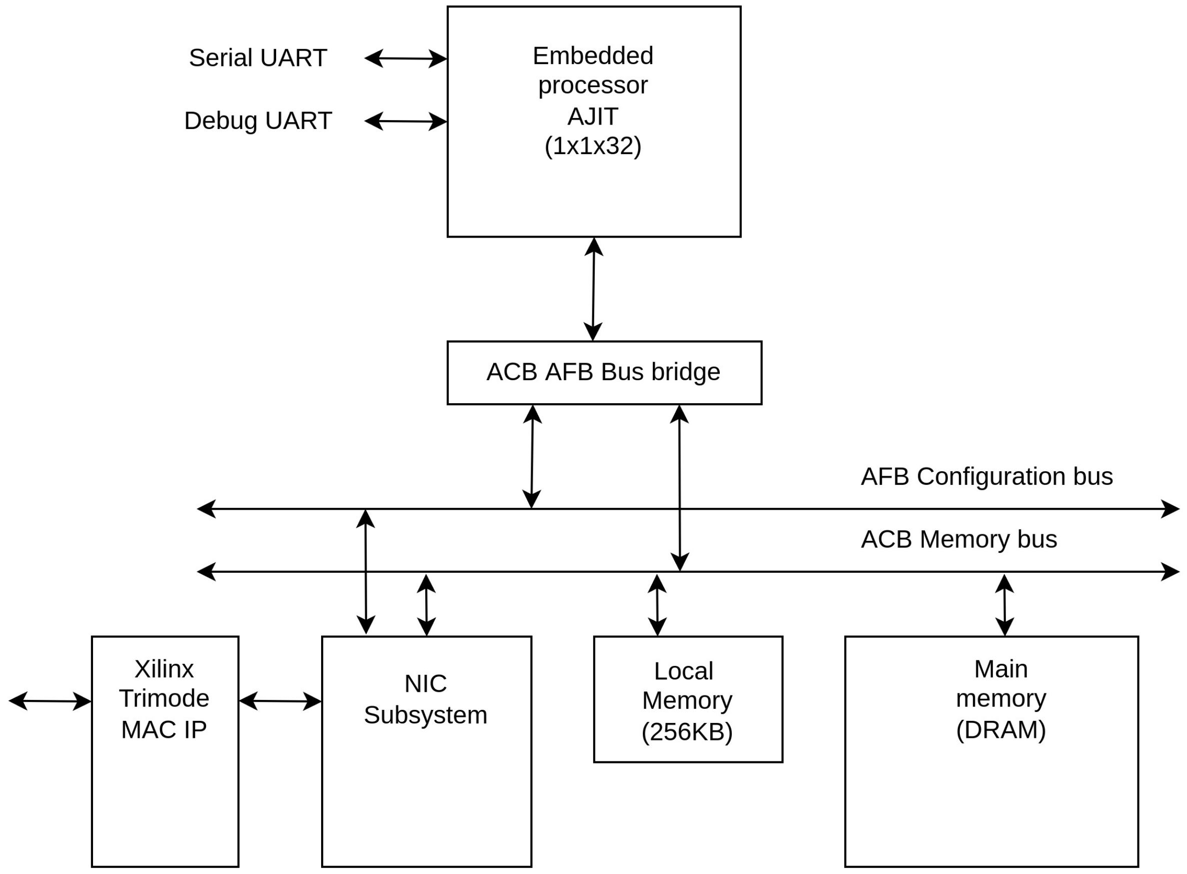

Figure 2.5: Architecture showing the NIC and processor sharing a 256KB fast local packet memory

2.7 Optimization B: Shared Inclusive Write-Back L2 Cache for NIC_1.1 and Processor

This optimization introduces a shared inclusive write-back L2 cache on the path to DRAM, accessible by both the processor and **NIC_1.1**. The L2 cache holds descriptor rings and packet buffers, allowing frequently accessed data to reside closer to the compute and I/O engines.

The inclusivity property ensures coherence by making data visible to both the processor and the NIC. By reducing redundant DRAM accesses and improving temporal locality, this cache architecture lowers memory access latency and increases packet processing throughput.

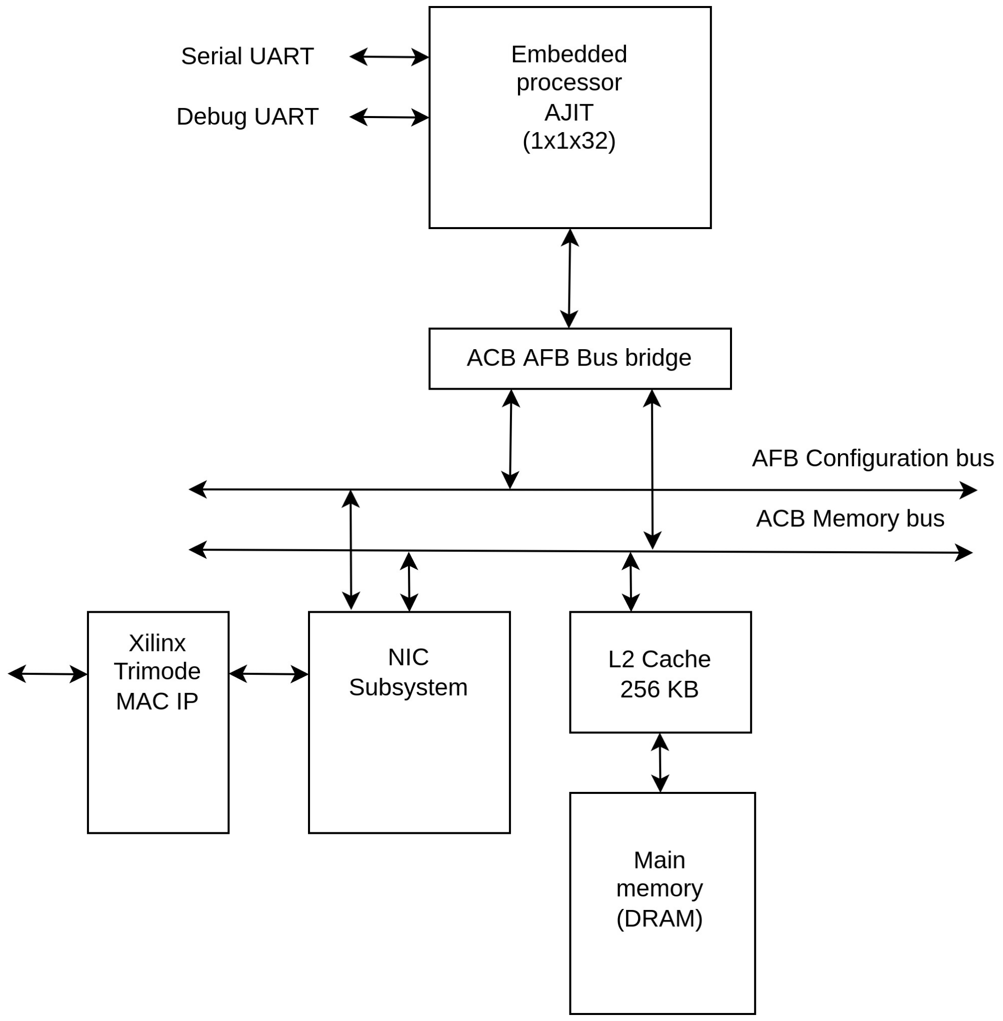

Figure 2.6: NIC and processor sharing access to main memory via an L2 cache

2.8 Summary and Comparison of Baseline and Optimized Architectures

This section presents a consolidated comparison of key performance metrics—including average round-trip time (RTT) and raw Ethernet throughput—across the baseline design and the two proposed optimizations. The tests were performed using the same custom Python tool and raw Ethernet packet methodology described earlier.

2.8.1 Round-Trip Time (RTT) Comparison

Table 2.1: Average Round-Trip Time (RTT) in microseconds for Various Payload Sizes

Payload Size (bytes)	Baseline	Opt A (Fast Local Memory)	Opt B (L2 Cache)
64	230	192	202
128	244	201	213
256	270	220	234
512	323	261	280
1024	409	333	354

2.8.2 Raw Ethernet Throughput Comparison

Note: The total Ethernet frame size equals 14 bytes of Ethernet header (6 bytes destination MAC + 6 bytes source MAC + 2 bytes Ethertype) plus the payload length.

From the results, **Optimization A**, which employs fast local shared memory, consistently achieves the lowest RTT and highest throughput across all payload sizes. **Optimization B** also shows significant performance gains over the baseline by mitigating DRAM latency through the use of a shared inclusive L2 cache. These findings highlight the importance of memory subsystem design in improving overall SoC network performance.

Table 2.2: Raw Ethernet Throughput (Mbps) for Various Payload Sizes

Payload Size (bytes)	Baseline	Opt A (Fast Local Memory)	Opt B (L2 Cache)
64	9.28	9.46	9.41
128	15.84	17.37	16.95
256	18.56	32.42	28.64
512	27.92	49.97	43.96
1024	36.24	57.33	51.58
1486	40.72	62.37	56.46

2.9 Preliminary Performance Verification of Processor and NIC

Although Optimizations A and B—incorporating fast local memory and a shared L2 cache—yielded reductions in round-trip time (RTT) and increases in throughput, the improvements were not sufficient to meet the target performance of approximately 600 Mbps. This suggested that memory latency was not the dominant limiting factor.

To isolate the performance bottleneck, a 32-bit cycle counter was integrated into the **NIC_1.1** to measure timing from within the AJIT processor. Loopback experiments involving simple MAC address swapping revealed that the processor contributed only 10 to 15 μs of latency, confirming that processor execution time was not a major constraint.

Further, the *Tri-Mode Ethernet MAC Example Design on KC705* was employed to evaluate the Ethernet MAC performance independently of both the NIC and processor. As presented in Appendix B, this test setup achieved low RTT and high throughput, verifying that the MAC subsystem itself was not a bottleneck.

With the processor and MAC subsystems effectively ruled out, the focus of further optimization shifted to the internal architecture of the NIC. Subsequent chapters explore enhancements to the NIC design aimed at closing the gap between observed performance and the expected throughput target.

Chapter 3

Optimization 1: Relocation of Queue Structures to NIC_1.2 Hardware

3.1 Introduction

This chapter introduces the first major architectural enhancement in the Network-Centric System-on-Chip (N-SoC): the transition from **NIC_1.1** to **NIC_1.2**. The key improvement involves relocating queue data structures from off-chip DRAM to on-chip hardware FIFOs, implemented using FPGA BRAM. This shift reduces latency, alleviates bus contention, and improves overall packet throughput.

The updated design is implemented and validated on the Xilinx KC705 FPGA platform. Further details on the NIC architecture and register mapping are available in Appendix A.

3.2 Queue Structure Relocation

3.2.1 Baseline Architecture Limitations

In **NIC_1.1**, the receive, transmit, and free queues are stored in off-chip DRAM. Since both the processor and NIC access these queues via the shared AJIT Core Bus (ACB), this results in increased latency and bus contention, slowing down packet handling operations.

3.2.2 NIC_1.2 Architecture

The **NIC_1.2** redesign moves all queue structures into on-chip hardware FIFOs, implemented using FPGA BRAM and accessed via memory-mapped registers. This relocation yields several benefits:

- **Lower Queue Access Latency:** Hardware FIFOs enable faster enqueue/dequeue operations.
- **Reduced DRAM Bandwidth Usage:** DRAM is freed for packet buffers rather than queue management.
- **Simplified Polling Logic:** The processor interacts with memory-mapped FIFO registers directly, reducing control overhead.

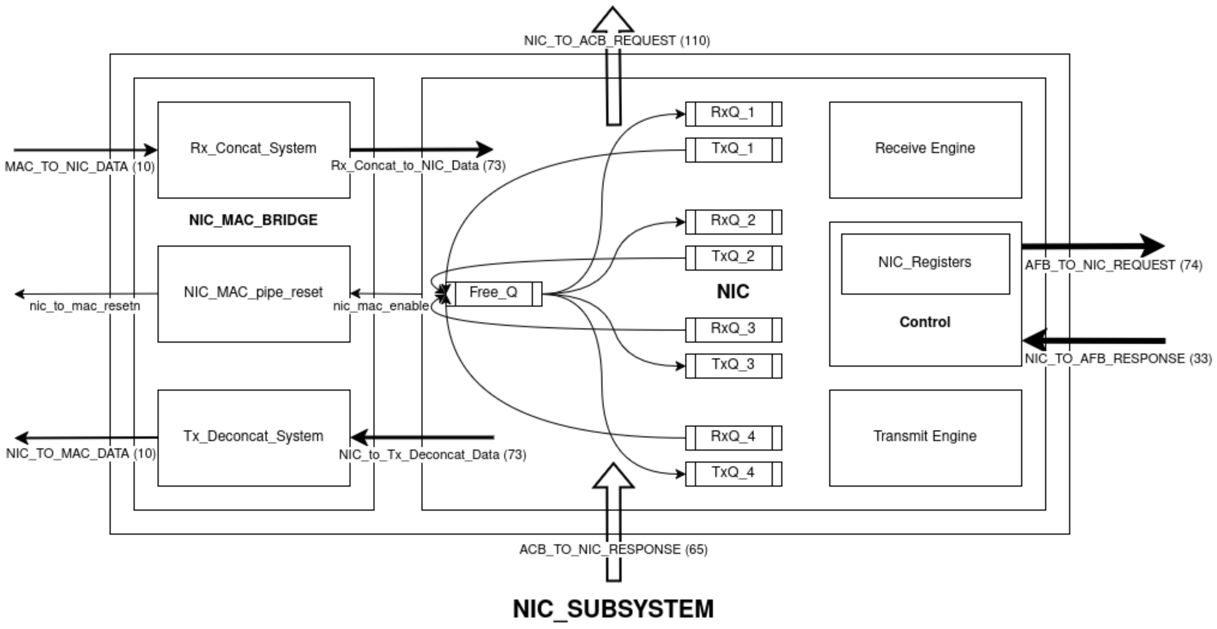

Figure 3.1: Queue structures implemented as hardware FIFOs inside **NIC_1.2**

3.2.3 On-Chip Queue Architecture and Management

NIC_1.2 implements all buffer management queues as on-chip hardware FIFOs using FPGA BRAM. Each FIFO has a depth of 64 entries and is accessible through a set of memory-mapped registers.

The queues are organized as follows:

- **Free Queue:** Shared between the processor and NIC to manage available buffer indices.
- **Receive Queues (Rx):** Four independent FIFOs, one per server, for incoming packet buffer pointers.
- **Transmit Queues (Tx):** Four FIFOs, one per server, for outgoing packet buffer pointers.

Queue configuration and activation are controlled dynamically at runtime by the processor through control registers.

Hardware Daemons Each queue category is serviced by a dedicated hardware daemon within **NIC_1.2**:

- **Free Queue Daemon:** Manages all free queue transactions, ensuring mutual exclusion via a lock mechanism.
- **Receive Queue Daemon:** Populates server-specific Rx queues using round-robin scheduling across enabled channels.
- **Transmit Queue Daemon:** Services Tx queues in a similarly balanced manner.

3.2.4 Concurrency and Synchronization

- **Free Queue:** Shared access between the processor and NIC necessitates a hardware lock to ensure atomic operations.
- **Rx/Tx Queues:** Accessed exclusively by either the processor or NIC, allowing lock-free operation.

Free Queue Lock Protocol A lock register provides atomic access to the free queue:

1. The processor or NIC reads `free_queue_lock`. A value of `1` indicates the lock is granted.
2. The queue operations (`push`, `pop`, `status`) are performed.
3. The lock is released by writing `1` back to the same register.
4. If the lock is busy (`0` returned), the agent retries until successful.

3.2.5 Queue Operations and Register Interface

All queue interactions are carried out via memory-mapped registers:

- **Push:** Writing a 32-bit buffer pointer to a queue register enqueues the pointer.
- **Pop:** Reading from a queue register dequeues a pointer.
- **Status:** Reading the status register returns the current queue occupancy.

Buffer pointers are **8-byte aligned**, so their three least significant bits are always zero. These bits are repurposed to encode operation status, enabling atomic `push` and `pop` operations without additional synchronization.

Operation Status Encoding

- **Push Return Values:**
  - `0x00000000` (bit 1 = 0): Push successful.
  - `0x00000002` (bit 1 = 1): Push failed (e.g., queue full).
- **Pop Return Values:**
  - Valid buffer pointer with bit 0 = 0: Pop successful.
  - `0x00000001` (bit 0 = 1): Pop failed (e.g., queue empty).

3.2.6 Register Map Overview

- **Free Queue:**
  - `free_queue_lock_reg`
  - `free_queue_reg`
  - `free_queue_status_reg`
- **Per Server (up to 4):**
  - `rx_queue_reg[n]`, `rx_queue_status_reg[n]`
  - `tx_queue_reg[n]`, `tx_queue_status_reg[n]`

This results in three registers for the free queue and sixteen registers for Rx and Tx queues combined.

Atomicity Assurance By embedding status flags within buffer pointer values, all push and pop operations remain atomic, enabling reliable concurrent access without complex handshaking or race conditions.

3.3 Simulation and Testing Infrastructure

To verify the correctness and performance of the **NIC_1.2** design, a dedicated simulation environment was developed. It includes hardware test modules within the NIC and a C-based testbench that interacts with the simulated system.

3.3.1 Hardware Test Modules

Two simulation-only daemons were integrated into **NIC_1.2** to validate internal functionality independently of external systems:

- **queueTestDaemon:** Validates queue operations such as `lock`, `unlock`, `push`, `pop`, and `status`.
- **memoryTestDaemon:** Tests NIC-initiated memory access using a simulated memory model.

These daemons support targeted testing of NIC components in isolation.

3.3.2 C-Based Simulation Testbench

A C-based testbench drives simulation by emulating external system components and interacting with NIC registers. Key elements include:

- **Dummy Processing Element:** Emulates a host processor to initiate tests.
- **Dummy Memory:** Simulates main memory and buffer regions.

Supported Test Macros

The testbench supports several macros for targeted validation:

- `CHECK_QUEUES`: Tests queue operations from the processor side.
- `DEBUG_QUEUE`: Tests queue functionality via `queueTestDaemon`.

- `CHECK_QUEUE_SEQUENCE`: Verifies correct sequencing of queue operations.
- `DEBUG_QUEUE_SEQUENCE`: Tests sequencing using `queueTestDaemon`.
- `CHECK_MEMORY_ACCESS`: Verifies processor-initiated memory access.
- `DEBUG_MEMORY_ACCESS`: Validates NIC-side memory access via `memoryTestDaemon`.
- `CHECK_NIC`: Conducts an end-to-end packet processing test.

Functional Testing of NIC_1.2

The `CHECK_NIC` macro performs full-system testing with:

- **Packet Generator/Transmitter:** Simulates incoming traffic.
- **Packet Receiver/Checker:** Validates transmitted packets.
- **Forward Daemon:** Emulates Rx-to-Tx forwarding.

All modules interact with dummy memory via the `memoryDaemon`, simulating realistic buffer usage.

3.3.3 Coverage and Validation Scope

The simulation framework verifies:

- Queue operations and atomic locking.
- NIC memory access behavior.
- End-to-end packet handling across the data path.

This infrastructure ensures functional correctness prior to hardware deployment.

3.4 NIC_1.2 Test Modes

To evaluate functionality and isolate performance overheads, **NIC_1.2** supports three test modes. Each mode can be selected using simulation or hardware macros:

1. **Normal Mode** (`CHECK_NIC` / `NORMAL_MODE`)
2. **NIC Loopback Mode** (`NIC_LOOPBACK`)
3. **MAC Loopback Mode** (`MAC_LOOPBACK`)

3.4.1 Normal Mode

Validates the complete data path, including interaction with the processor:

- NIC dequeues a buffer pointer from the free queue.
- Incoming packet is written to the buffer.
- Pointer is enqueued to the receive queue.
- Processor moves the pointer to the transmit queue.
- NIC transmits the packet and returns the pointer to the free queue.

3.4.2 NIC Loopback Mode

Evaluates internal NIC functionality, excluding the processor:

- NIC receives a packet and stores it in a buffer.
- The buffer is immediately looped to the transmit queue.
- After transmission, the pointer is recycled to the free queue.

This mode isolates processor overhead by comparison with Normal Mode.

3.4.3 MAC Loopback Mode

Tests the MAC IP core in isolation:

- Packets received by the MAC's RX FIFO are directly looped back to its TX FIFO.

Comparison with NIC Loopback isolates NIC-internal latency.

3.4.4 Timing Breakdown Summary

Round-trip time (RTT) comparisons across the three modes yield estimates for individual latency components:

- **Processor Overhead:** Normal Mode RTT − NIC Loopback RTT
- **NIC Hardware Latency:** NIC Loopback RTT − MAC Loopback RTT
- **MAC Latency:** Directly measured from MAC Loopback RTT

3.5 Simulation Timing Results

Simulation results confirm substantial performance improvements in **NIC_1.2** compared to **NIC_1.1**, particularly in register access and queue operations.

- **Register Access:** Each read or write completes in approximately **15 clock cycles** (150 ns at a 10 ns clock period), enabling efficient communication between the processor and NIC.
- **Queue Operations (push/pop):**
  - **NIC_1.1:** Around 286 clock cycles (2.86 μs) per operation.
  - **NIC_1.2:** Reduced to just 20 clock cycles (200 ns) per operation.

A full queue traversal sequence—transferring a buffer pointer through the free, receive, and transmit queues:

$$\texttt{pop}_{\text{free}} \rightarrow \texttt{push}_{\text{receive}} \rightarrow \texttt{pop}_{\text{receive}} \rightarrow \texttt{push}_{\text{transmit}} \rightarrow \texttt{pop}_{\text{transmit}} \rightarrow \texttt{push}_{\text{free}}$$

—takes only **218 clock cycles** (2.18 μs) in **NIC_1.2**, representing a major improvement over **NIC_1.1**.

These enhancements significantly reduce latency and improve packet throughput, resulting in a more responsive and efficient NIC.

3.6 FPGA-Based Testing with AJIT Processor

For hardware-level validation, **NIC_1.2** was integrated with the AJIT processor on the Xilinx KC705 FPGA. A C program running on AJIT exercises the NIC in all three test modes:

3.6.1 Normal Mode (`NORMAL_MODE`)

Validates the full datapath with processor involvement. The processor manages buffer pointer transitions across all queues, enabling end-to-end flow testing.

3.6.2 NIC Loopback Mode (`NIC_LOOPBACK`)

Tests autonomous NIC operation. The NIC independently completes the receive-forward-transmit cycle. RTT comparisons with Normal Mode reveal processor overhead.

3.6.3 MAC Loopback Mode (`MAC_LOOPBACK`)

Isolates the MAC IP core by looping packets internally between RX and TX FIFOs. RTT comparison with NIC Loopback reveals NIC hardware latency.

3.6.4 Summary

These modes provide comprehensive functional and timing validation of **NIC_1.2** on FPGA hardware, confirming correct integration with the AJIT processor, quantifying latency contributors, and verifying overall design robustness.

Chapter 4

Optimization 2: Increasing MAC FIFO Depths (NIC_1.2)

4.1 Introduction

This chapter describes the second architectural optimization in the Network-Centric System-on-Chip (N-SoC) design, focusing on enhancements from **NIC_1.1** to **NIC_1.2**. The main improvement involves increasing the FIFO buffer sizes in the Xilinx Tri-mode Ethernet MAC (TEMAC) IP core, expanding both receive (RX) and transmit (TX) FIFOs from 4kB to 16kB. This change significantly enhances throughput stability and reliability, particularly under bursty or high-traffic conditions.

The enhancement was implemented and validated on the KC-705 FPGA platform. Its impact on performance is discussed below.

4.2 Increasing MAC FIFO Depths in NIC_1.2

4.2.1 Baseline Configuration

In **NIC_1.1**, the TEMAC FIFOs were configured with 4kB buffers for both RX and TX paths. While adequate for steady, moderate traffic, these limited buffer depths led to packet loss and reduced reliability when faced with traffic bursts or sustained high throughput due to buffer overflow.

4.2.2 Design Modifications in NIC_1.2

To mitigate these issues, the FIFO depths were increased to 16kB by adjusting the FIFO address widths and BRAM configurations within the TEMAC wrapper. Furthermore, the FIFO size is now fully parameterized in the VHDL wrapper, allowing flexible adjustments in powers of two for future scalability.

Specifically:

- **RX FIFO:** Enlarged to 16kB, accommodating larger or bursty incoming traffic without overflow.
- **TX FIFO:** Expanded to 16kB, enabling buffering of more outgoing packets and reducing transmission backpressure.

These changes improved data stability and throughput efficiency under stress testing.

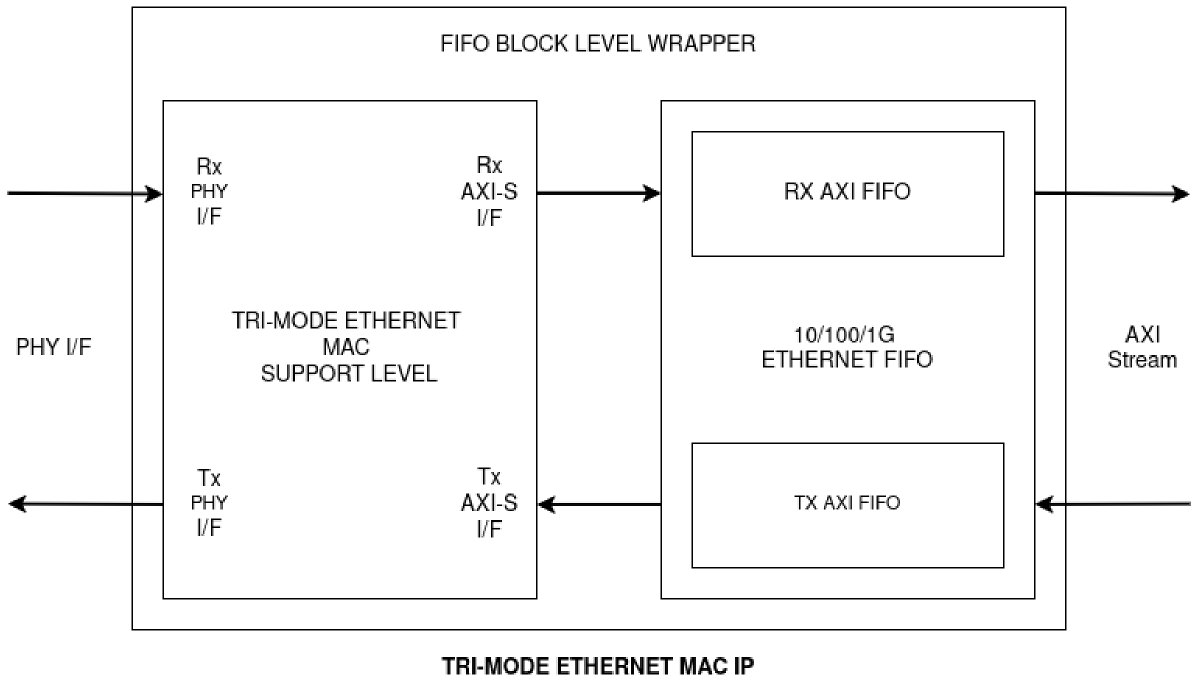

Figure 4.1: TEMAC IP with Expanded RX/TX FIFOs (16kB each) in NIC_1.2

4.2.3 FIFO Implementation Details

The FIFO buffers are dual-port BRAMs interfaced via AXI-Stream protocols, supporting asynchronous read/write clock domains between MAC and user logic.

RX FIFO

- Stores incoming Ethernet frames, detecting frame boundaries using a parity bit.
- Handles bad frames by overwriting data flagged via the `rx_axis_mac_tuser` signal.
- Supports graceful overflow recovery by resetting the write pointer.
- Supports minimum frame size of 8 bytes.
- Memory capacity upgraded from 4kB to 16kB by widening address bus.

TX FIFO

- Buffers outbound Ethernet frames for transmission.
- Supports frame drops when exceeding available space.
- Maintains retransmit window features for half-duplex operation.
- Supports minimum frame size of 14 bytes.
- Memory capacity expanded from 4kB to 16kB.

4.2.4 VHDL Wrapper and Parameterization

Previously, FIFO sizes were hardcoded in the TEMAC wrapper. The updated design introduces a parameterized VHDL wrapper that allows the FIFO depth to be set via generics at instantiation, enabling flexible adjustment based on system requirements.

> *"This FIFO block VHDL wrapper enhances the standard Tri-mode Ethernet MAC core by integrating AXI-Stream FIFOs with configurable sizes adjustable through generics."*

4.3 Summary

By increasing MAC FIFO depths to 16kB and parameterizing their size, **NIC_1.2** achieves higher resilience against packet loss under bursty and high-load scenarios, thereby improving overall system throughput and stability.

Chapter 5

Optimization 3: Packet Scheduling Strategies (NIC_1.2)

5.1 Introduction

This chapter presents the third architectural optimization of the Network-Centric System-on-Chip (N-SoC) design, highlighting the transition from **NIC_1.1** to **NIC_1.2**. The focus is on improving packet scheduling to eliminate head-of-line blocking and enhance bandwidth utilization in multi-flow, high-throughput Ethernet scenarios. These enhancements are essential for supporting sustained large data transfers reliably and efficiently.

The updated scheduling mechanism has been implemented and validated on the Xilinx KC705 FPGA platform. The following sections describe the design and analyze its performance impact.

5.2 Packet Scheduling Strategies in NIC_1.2

5.2.1 Motivation

In high-throughput environments with multiple active data flows, basic FIFO-based transmission can lead to head-of-line blocking and suboptimal bandwidth utilization. To address this, more effective scheduling is required to maximize throughput and maintain reliable data delivery, particularly when handling large payloads.

5.2.2 Implemented Scheduling Schemes

5.2.2.1 Scheme 1: FIFO Scheduling with Inter-Packet Delay

This baseline method transmits packets in strict FIFO order, inserting a fixed delay between packets to prevent receiver buffer overflow. While it ensures zero packet loss, throughput is limited due to conservative pacing.

When transmitting from the host system, this inter-packet delay critically affects the maximum throughput achievable without packet loss. Sending packets too rapidly can overflow MAC FIFOs or system buffers, causing drops; excessive delays, however, reduce throughput. Thus, careful tuning of this delay is essential. Appendix C provides detailed analysis and measurements on host-side packet pacing.

5.2.2.2 Scheme 2: Burst Transmission with Controlled Delay

Packets are sent in bursts followed by short pauses, with burst sizes optimized to match MAC FIFO capacities to avoid overflow:

- For 4 kB FIFOs (**NIC_1.1**), optimal bursts contain 4–5 packets with approximately 0.1 ms delay.
- For 16 kB FIFOs (**NIC_1.2**), burst sizes increase to 16–18 packets with the same delay.

This scheme substantially improves throughput while maintaining zero packet loss.

5.2.2.3 Scheme 3: Continuous Transmission with FIFO Monitoring

Transmission begins with a burst (e.g., 16–18 packets) and thereafter a new packet is sent each time one is dequeued, keeping the MAC FIFOs continuously full. This approach maximizes throughput under ideal conditions but lacks mechanisms for detecting or recovering from packet loss or corruption, reducing reliability.

5.2.2.4 Scheme 4: Reliable Packet Scheduling with Acknowledgment Protocol

To achieve both high throughput and reliability, **NIC_1.2** employs a custom lightweight HDL protocol featuring three packet types:

- **Data Packet (Type 0x01):** Consists of a 14-byte Ethernet header, a 1-byte packet type field, a 4-byte sequence number, and up to 1481 bytes of payload (to fit within a 1500-byte MTU).
- **Summary Packet (Type 0x02):** Sent after all data packets, it conveys the total number of packets and overall payload size.
- **Acknowledgment Packet (Type 0x03):** Sent by the receiver to indicate either complete receipt (”OK”) or missing packets (”MI”) along with their sequence numbers.

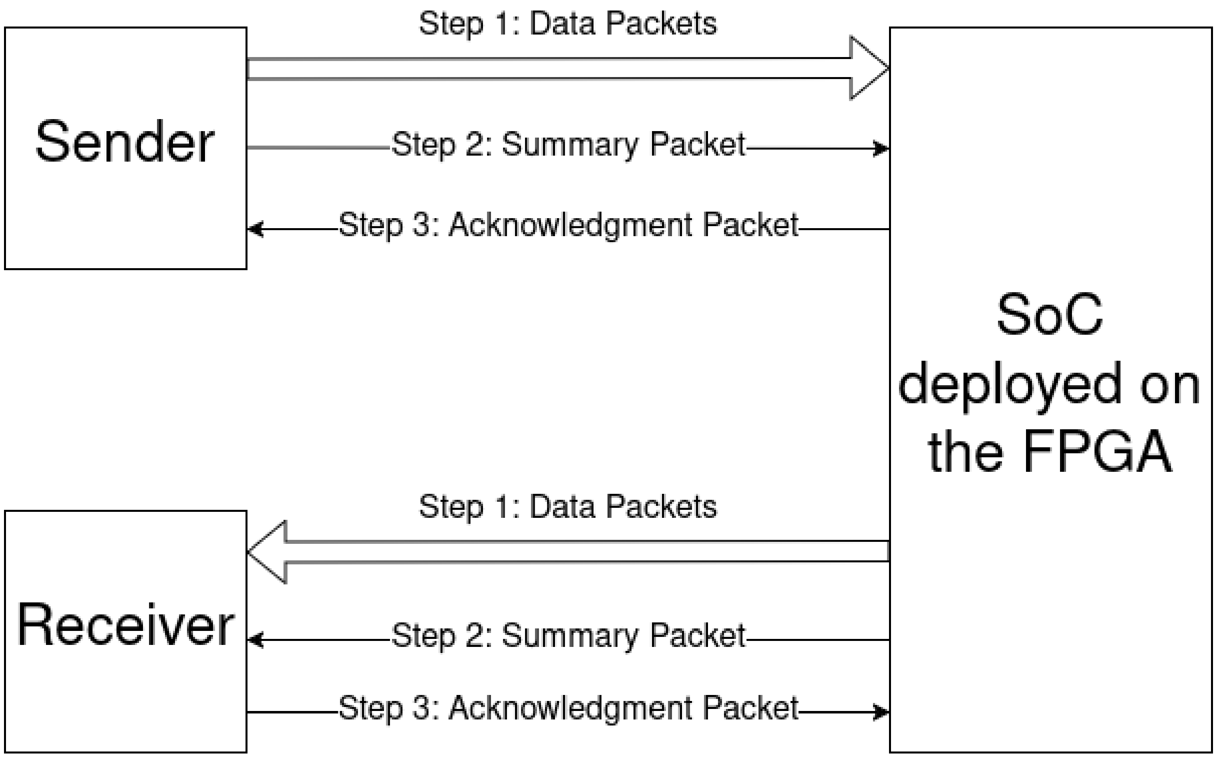

Figure 5.1: NIC_1.2 Implementation of Scheme 4: Acknowledgment-Based Reliable Packet Scheduling

Protocol operation:

1. The sender segments the data into data packets and transmits them sequentially.
2. Once transmission completes, a summary packet is sent specifying the total packets and payload size.

3. The receiver verifies completeness and responds with an acknowledgment indicating success or listing any missing packets.
4. The sender retransmits any missing packets and resends the summary packet.
5. This process repeats until the receiver confirms full receipt.

Timeout mechanisms are incorporated to prevent deadlocks: if no packets or acknowledgments are received within 10 seconds, both sender and receiver terminate gracefully, ensuring robustness in case of faults.

5.2.3 Advantages of the Reliable Protocol

Scheme 4 delivers multiple key benefits:

- Ensures data integrity without relying on a full IP stack.
- Reduces overhead by retransmitting only lost packets.
- Maintains high throughput by efficiently saturating MAC FIFOs.
- Offers explicit feedback and clean termination conditions.
- Supports robust, high-speed large data transfers suitable for accelerator pipelines.

This protocol removes the Ethernet interface as a throughput bottleneck by enabling reliable, high-speed streaming with efficient software-managed control. The design supports low latency and consistent performance.

5.3 Summary

This chapter presented packet scheduling improvements in **NIC_1.2**, emphasizing fair resource sharing, throughput optimization, and data reliability through a custom acknowledgment protocol.

The combined effect of these enhancements on round-trip latency, throughput, and overall system responsiveness is analyzed in Chapter 6.

Chapter 6

Evaluation and Testing of NIC_1.2

6.1 Overview

This chapter presents a comprehensive evaluation of the **NIC_1.2** design, with a particular focus on two critical performance metrics: **Round-Trip Time (RTT)** and **Throughput**. The evaluation includes both simulation-based analyses—conducted under various loopback configurations—and real hardware experiments implemented on an FPGA platform. Performance comparisons are made between the baseline and optimized versions of the design across different payload sizes and operating modes.

6.2 Simulation-Based Evaluation

6.2.1 RTT Measurements

Table 6.1 reports the RTT (in microseconds) for varying packet lengths across three simulation modes: **Normal Mode**, **NIC_loopback Mode**, and **MAC_loopback Mode**. RTT measures the total time for a packet to be sent, processed, and retransmitted.

Table 6.1: RTT Comparison Across Simulation Modes (µs)

Packet Length (bytes)	**Normal Mode**	**NIC_loopback**	**MAC_loopback**
32	3.574	3.164	0.834
64	4.694	4.164	1.474
128	6.864	6.424	2.754
256	12.104	11.544	5.314
512	22.184	21.784	10.434
1024	42.624	42.264	20.674
1500	61.804	61.314	30.194

6.2.2 Throughput Comparison

Table 6.2 presents throughput (Mbps) results for 1024-byte packets under the three simulation modes. The **MAC_loopback Mode** achieves the highest throughput, benefiting from internal loopback efficiencies.

Table 6.2: Simulation Throughput for 1024-byte Packet (Mbps)

Mode	**Send Throughput**	**Receive Throughput**
Normal Mode	1233.33	1234.07
NIC_loopback	1248.15	1248.90
MAC_loopback	1600.15	1600.15

6.3 FPGA-Based Evaluation

6.3.1 RTT Measurements on Hardware

Table 6.3 lists average RTTs measured on FPGA using MAC loopback mode. Additionally, Table 6.4 compares the RTT between baseline and Optimization A (using fast local memory) under normal mode.

Table 6.3: RTT in MAC Loopback Mode (NIC_1.2 on FPGA, µs)

Payload Length (bytes)	**Average RTT**
64	108
128	125
256	136
512	146
1024	166

Table 6.4: RTT Comparison: Baseline vs Opt A (Fast Local Memory) (Normal Mode, µs)

Payload Length (bytes)	**Baseline**	**Opt A (Fast Local Memory)**
64	143	140
128	146	145
256	151	149
512	165	156
1024	188	180

6.3.2 Throughput on FPGA

Throughput measurements performed on the KC-705 FPGA board are summarized in the following tables.

MAC Loopback Throughput Table 6.5 shows the send and receive throughput achieved in MAC loopback mode for various payload sizes.

Table 6.5: MAC Loopback Throughput (FPGA, Mbps)

Payload Length (bytes)	Send	Receive
64	297.07	297.07
128	407.06	407.06
256	689.93	689.93
512	867.73	867.73
1024	986.16	986.16
1486	1018.85	1018.85

Throughput After All Optimizations Table 6.6 shows the throughput results after applying all three optimizations described above, including moving queues into NIC hardware with fast local memory, Scheme 3 continuous scheduling, and increasing the MAC FIFO size to 16 kB.

Table 6.6: Throughput with 16-kB MAC FIFOs (FPGA, Mbps) — Final Optimized Design

Payload Length (bytes)	Baseline	Optimized (All Steps)
64	52.60	62.74
128	84.79	100.94
256	151.49	179.19
512	270.42	334.75
1024	477.09	602.45
1486	526.12	731.89

6.4 Image Transfer Performance

Real-world image transfer tests were conducted using two protocols: **Scheme 3** (continuous scheduling) and **Scheme 4** (acknowledgment-based). Both small and large image sizes were evaluated, with throughput metrics and images transferred per second reported for comprehensive assessment. *Images/sec values were calculated using the total transmission (Tx) and reception (Rx) time.*

Table 6.7: Image Transfer Performance for NIC_1.2 with Fast Local Memory — Small Image (Measured on FPGA)

Metric	**Send (Mbps)**	**Receive (Mbps)**	**Images/sec**
Small Image (347.5 kB; 680 × 680) — Continuous Scheduling (Scheme 3)			
Average Throughput	659.12	657.66	115.6
Maximum Throughput	791.18	781.49	–
Small Image (347.5 kB; 680 × 680) — ACK-Based Protocol (Scheme 4)			
Average Throughput	647.79	647.31	113.8
Maximum Throughput	761.81	750.63	–

Table 6.8: Image Transfer Performance for NIC_1.2 with Fast Local Memory — Large Image (Measured on FPGA)

Metric	**Send (Mbps)**	**Receive (Mbps)**	**Images/sec**
Large Image (15.9 MB; 4600 × 4600) — Continuous Scheduling (Scheme 3)			
Average Throughput	653.03	652.56	2.45
Maximum Throughput	706.40	705.65	–
Large Image (15.9 MB; 4600 × 4600) — ACK-Based Protocol (Scheme 4)			
Average Throughput	638.25	637.37	2.39
Maximum Throughput	702.61	701.89	–

6.5 Summary

The evaluation shows that **NIC_1.2** significantly improves performance by relocating queue structures from memory to on-chip hardware, increasing the MAC FIFO size from 4 kB to 16 kB, and applying advanced packet scheduling strategies. In particular, continuous transmission combined with larger FIFOs boosts throughput for large data transfers.

Real-world image transfer tests demonstrate stable, high performance, achieving over 115 images/sec for small images and around 2.45 images/sec for large images. FPGA-based experiments validate these architectural enhancements, confirming the effectiveness of hardware queue management and scheduling. The next chapter summarizes these key optimizations and their impact on the overall N-SoC architecture.

Chapter 7

Conclusion & Future Work

7.1 Conclusion

This thesis presented the design, implementation, and validation of a Network-Centric System-on-Chip (N-SoC) platform integrating a 32-bit AJIT processor, a custom high-performance Network Interface Controller (NIC), and high-capacity DRAM on a Xilinx KC705 FPGA. The goal was to enable high-throughput, low-latency packet processing over raw Ethernet.

Starting from the baseline **NIC_1.1**, the design was optimized to **NIC_1.2** with key enhancements:

- Moving queue buffer pointer management from DRAM to internal register-based FIFOs, reducing memory bandwidth and overhead.
- Expanding MAC FIFO size from 4 KB to 16 KB for more efficient continuous packet scheduling.
- Employing continuous, flow-based scheduling strategies to improve throughput and packet handling.
- Implementing reliable scheduling protocols to ensure fairness and robustness under heavy network load.

These improvements significantly boost packet processing efficiency, confirmed through simulation and FPGA testing, providing a solid foundation for advanced network-centric systems.

Performance Comparison: NIC_1.1 vs NIC_1.2

The following tables summarize the improvements observed in key performance metrics for various payload sizes.

- **Table 7.1** presents reductions in Round-Trip Time (RTT) across different payload sizes, demonstrating latency improvements in **NIC_1.2** compared to the baseline design.

Table 7.1: Average RTT (μs) Comparison: NIC_1.1 vs NIC_1.2

Payload (bytes)	**Baseline**		**Opt A (Fast Local Memory**	
	NIC_1.1	NIC_1.2	NIC_1.1	NIC_1.2
64	230	143	192	140
128	244	146	201	145
256	270	151	220	149
512	323	165	261	156
1024	409	188	333	180

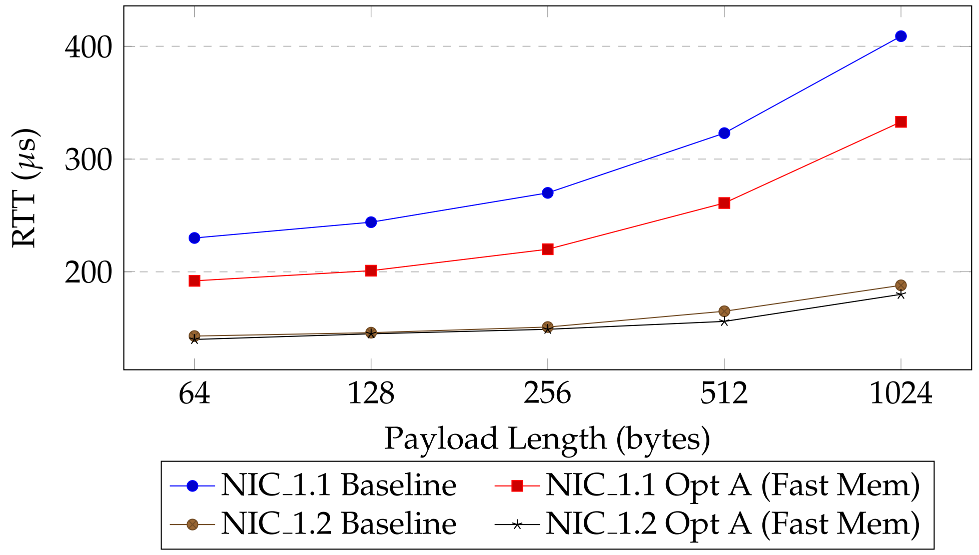

Figure 7.1: RTT Comparison between NIC_1.1 and NIC_1.2 (Baseline and Optimization A)

- **Table 7.2** shows throughput enhancements for various payload sizes, reflecting the increased data transfer rates and improved link utilization.

Table 7.2: Throughput (Mbps) Comparison: NIC_1.1 vs NIC_1.2

Payload (bytes)	**Baseline**		**Opt A (Fast Local Memory**	
	NIC_1.1	NIC_1.2	NIC_1.1	NIC_1.2
64	9.28	52.60	9.46	62.74
128	15.84	84.79	17.37	100.94
256	18.56	151.49	32.42	179.19
512	27.92	270.42	49.97	334.75
1024	36.24	477.09	57.33	602.45
1486	40.72	526.12	62.37	731.89

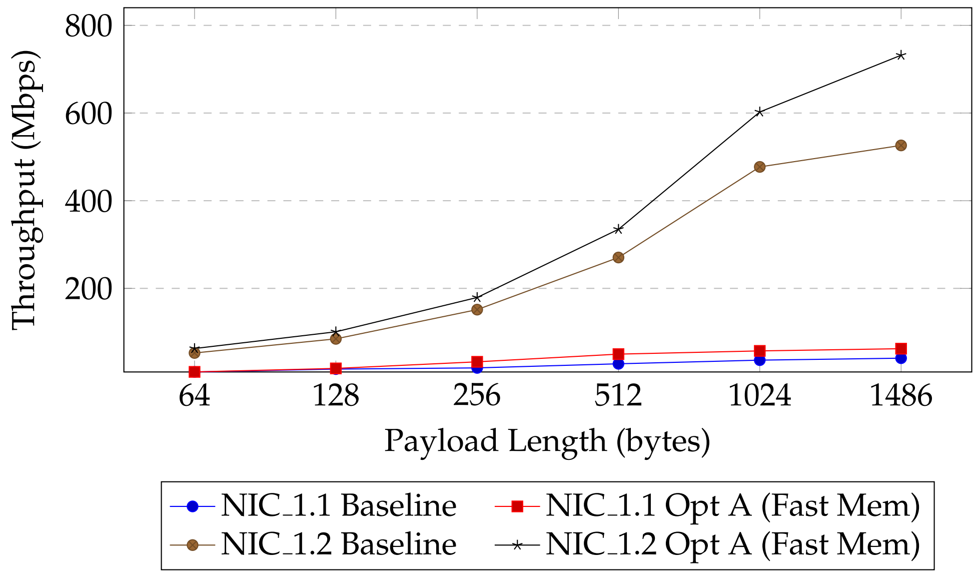

Figure 7.2: Throughput Comparison between NIC_1.1 and NIC_1.2 (Baseline and Optimization A, with 16-kB MAC FIFO)

Furthermore, image transmission tests using Scheme 3 showed NIC_1.2 achieved average throughputs of:

- **659.12 Mbps** for sending,
- **657.66 Mbps** for receiving,

demonstrating near-saturation of a 1 Gbps Ethernet link and confirming the efficiency of the architectural enhancements.

Summary of Performance Improvements

- **RTT Reduction: NIC_1.2** achieves up to **2×** lower latency compared to **NIC_1.1**.
- **Throughput Increase:** Throughput improves by up to **12×** for small payloads and **11×** for larger payloads.
- **Near-Saturation Image Transfer:** Sustains throughput close to **659 Mbps** on a 1 Gbps Ethernet link during image transmission tests, confirming the effectiveness of the architectural enhancements.

7.2 Summary of Findings

RTT measurements indicate that hardware-level Ethernet communication is highly efficient. However, host OS involvement contributes roughly 60 µs of latency, approximately equally divided between transmission and reception paths. This highlights the critical need to minimize OS overhead in latency-sensitive networking applications.

Throughput performance is influenced by the physical network capacity, Ethernet interface speed (e.g., 1 Gbps vs. 10 Gbps), and OS buffering strategies. Understanding and optimizing these factors is essential for maximizing the efficiency of FPGA-based network interfaces.

7.3 Future Work

Building on the architectural improvements, the following future work directions are proposed:

1. **64-bit NIC Upgrade:** Transition NIC_1.2 to 64-bit data width to support higher bandwidth and more efficient DRAM utilization.

2. **Custom Protocol Design:** Develop application-specific network protocols to optimize communication on the platform.

3. **Multi-core SoC Architecture:** Expand the design to multiple cores with separate control and data planes to improve scalability and performance.

4. **Achieve Beyond 1 Gbps Throughput:** Further architectural enhancements to surpass 1 Gbps for industrial and time-sensitive networking applications.

This N-SoC platform lays a solid foundation for scalable, high-performance, and extensible network-centric systems, addressing evolving demands in specialized networking environments.

Appendices

Appendix A

Standard NIC_1.2 (Network Interface Controller) Design

This appendix details the design of **NIC_1.2**, the Network Interface Controller developed and optimized as part of this thesis. **NIC_1.2** is an enhanced version of the baseline **NIC_1.1** (introduced in Chapter 2), incorporating several improvements to significantly boost throughput and resource utilization. It plays a critical role in providing high-performance network connectivity and communication within the AJIT processor-based System-on-Chip (SoC).

The design process involved considerations such as network protocol support, efficient packet handling, optimized memory management, and robust interfacing with the AJIT processor. This appendix provides a comprehensive overview of the **NIC_1.2** architecture, highlighting its key components, their interconnections, and the design choices made to overcome performance challenges.

A.1 Design Decisions

Before delving into the **NIC_1.2** architecture, it is important to review the key design decisions. The **NIC_1.2** receives packet data from the MAC, which is stored in memory. The processor is responsible for allocating this memory and informing **NIC_1.2** accordingly. Overall, the design requires three main interfaces to **NIC_1.2**. Figure A.1 illustrates these interfaces, which are discussed in detail in the following sections.

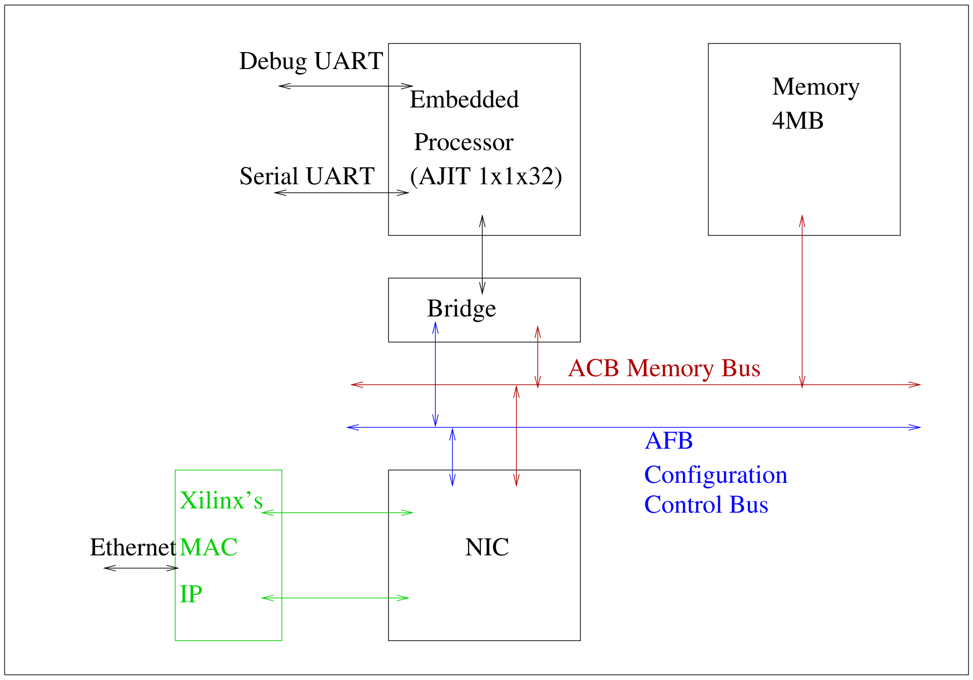

Figure A.1: Top level with Interfaces for NIC_1.2

A.1.1 NIC_1.2–MAC Interface

The **NIC_1.2**–MAC interface is used by **NIC_1.2** to receive and transmit packets to and from the MAC. Memory accesses on this interface provide 8 bytes (64 bits) per request. Accordingly, the interface width is set to 73 bits, comprising 64 bits of data and 9 bits of control signals. The detailed bit mapping is presented in Table A.1.

Signal Name	Location	Signal Description
tlast	[72:72]	The *tlast* signal becomes '1' if the 64-bit chunk is the last chunk of the packet.
tdata	[71:8]	The *tdata* contains the actual packet data chunk.
tkeep	[7:0]	The *tkeep* is an 8-bit field where each bit corresponds to one byte of data. If a bit is '1', the corresponding byte in the data is valid; otherwise, it is not.

Table A.1: NIC_1.2–MAC interface description

This single interface supports both packet reception and transmission operations.

A.1.2 NIC_1.2–Memory Interface

The **NIC_1.2**–Memory interface facilitates storing and loading packets to and from memory. This interface utilizes the already developed ACB (AJIT Core Bus) protocol.

The ACB protocol comprises two distinct interfaces:

1. **ACB Request:** Used by **NIC_1.2** to send requests for storing or loading packet data. The bit mapping for this interface is detailed in Table A.2.
2. **ACB Response:** Used by memory to send responses back to **NIC_1.2** following a request. The bit mapping is provided in Table A.3.

Signal Name	**Location**	**Signal Description**
lock	[109:109]	Lock bit; if set to '1' by a master, other masters are denied access to memory.
read/write_bar	[108:108]	'1' for read request, '0' for write request.
byte_mask	[107:100]	The *byte_mask* is an 8-bit field, each bit corresponding to one byte of data. If a bit is '1', the corresponding byte is valid.
address	[99:64]	Byte address where the read or write operation should be performed.
write_data	[63:0]	Data to be written.

Table A.2: NIC_1.2 - Memory interface description

Signal Name	**Location**	**Signal Description**
err	[64:64]	Value '1' indicates an errored response.
data	[63:0]	Contains read data if the request was a read operation.

Table A.3: Memory–NIC_1.2 interface description

A.1.3 NIC_1.2–Processor Interface

This control interface allows the processor to allocate memory for packet buffering and configure internal registers of **NIC_1.2**. These registers can be accessed using the AFB (AJIT FIFO Bus) protocol, which uses narrower address fields compared to ACB.

The AFB protocol includes two unidirectional channels:

1. **AFB Request:** Carries register read/write commands from the processor to NIC_1.2.
2. **AFB Response:** Returns corresponding data or acknowledgments from NIC_1.2.

For register details, refer to Section A.3.1.

Signal Name	Location	Signal Description
lock	[73:73]	Lock signal; when set to '1' by a master, other masters are blocked from accessing memory.
read/write_bar	[72:72]	Read/write control: '1' for read request, '0' for write request.
byte_mask	[71:68]	4-bit field indicating valid bytes in the data word. Each bit maps to 1 byte of the 32-bit data. A '1' marks the byte as valid.
address	[67:32]	Byte address where the memory-mapped register operation is to occur.
write_data	[31:0]	Data to be written to the specified register.

Table A.4: Processor to NIC_1.2 interface (AFB request) signal mapping

Signal Name	Location	Signal Description
err	[32:32]	Error signal; set to '1' if the request could not be serviced.
data	[31:0]	Data returned by NIC_1.2 in response to a read request.

Table A.5: NIC_1.2 to Processor interface (AFB response) signal mapping

A.2 Interface Data Structures

NIC_1.2 employs three hardware FIFOs—`free_queue`, `rx_queue`, and `tx_queue`—to manage buffer coordination and streamline communication between the processor and NIC hardware.

- **Free Queue (`free_queue`):** Stores physical addresses of available packet buffers and is shared between the receive and transmit paths to enable unified buffer reuse.
  - *Receive Path:* NIC_1.2 pops a buffer address to store incoming packets and pushes it to `rx_queue`. After processing, the processor returns the buffer to `free_queue`.
  - *Transmit Path:* The processor pops a buffer address, fills it with data, and pushes it to `tx_queue`. After transmission, NIC_1.2 returns the buffer to `free_queue`.
- **Receive Queue (`rx_queue`):** Indicates to the processor that a new packet is available. NIC_1.2 pushes buffer pointers after reception; the processor pops them for processing.
- **Transmit Queue (`tx_queue`):** Indicates to NIC_1.2 that a packet is ready to transmit. The processor pushes buffer addresses; NIC_1.2 pops and sends the corresponding packets.

Queue Synchronization

- **Free Queue:** Shared access by both NIC_1.2 and the processor requires atomic `lock/unlock` mechanisms for synchronization.
- **Rx/Tx Queues:** Accessed exclusively—`rx_queue` by the processor and `tx_queue` by NIC_1.2—thus no locking is required.

Initialization Sequence

At startup, the processor:

1. Allocates memory for packet buffers.

2. Populates `free_queue` with buffer addresses.
3. Configures NIC_1.2 via its control and queue registers.

These shared FIFOs and their associated protocols enable efficient, coordinated packet handling between the processor and NIC_1.2 hardware.

A.3 Network Interface Controller (NIC_1.2)

A.3.1 Register File Map

NIC_1.2 includes a memory-mapped register file of 256 registers that support configuration, status reporting, and queue management. Only a subset of these registers are used; others are reserved for future expansion.

Registers are categorized into:

- **Queue Registers:** Control and manage the hardware FIFOs (`rx_queue`, `tx_queue`, and `free_queue`), including push/pop operations, status checks, and locking.
- **Control and Status Registers:** Provide mechanisms for enabling servers, setting control flags, monitoring buffer availability, reading MAC addresses, accessing counters, and supporting diagnostics.

Tables A.6 and A.7 list the detailed mapping of these registers.

A.3.2 Receive Engine

The Receive Engine accepts packets from the MAC, allocates free buffers from the shared `free_queue`, writes incoming packet data into these buffers, and pushes the buffer pointers into the `rx_queue`. This mechanism ensures continuous availability of buffers for incoming traffic.

A.3.3 Transmit Engine

The Transmit Engine monitors the `tx_queue` for packets ready to send. Upon transmission completion, it recycles the corresponding buffers back to the `free_queue`, enabling efficient reuse and maintaining a steady packet flow.

Reg. ID	Address Offset	Description
0	0x00	Control Register
1	0x04	Servers Enabled
2	0x08	Debug Register 1
3	0x0C	Debug Register 2
4	0x10	Buffers in Each Queue
208	0x33C	MAC Address [31:0]
209	0x340	MAC Address [47:32]
210	0x344	Packets Transmitted Counter
211	0x348	Packets Received Counter
212	0x34C	Status Word
255	0x3FC	Free-Running Counter

Table A.6: NIC_1.2 Control and Status Registers Map

A.3.4 Register Access Daemon

This daemon enables processor access to NIC_1.2 registers over the AFB protocol. It supports:

- Configuration of queue operations and server controls.
- Reading debug and status registers.
- Managing queue operations protected by hardware locks.

The processor accesses registers via the IDs detailed in Tables A.6 and A.7. Registers not explicitly assigned are reserved for future use.

Reg. ID	Address Offset	Description
8	0x20	RxQ Server 0 Register
9	0x24	RxQ Server 0 Status Register
10	0x28	RxQ Server 1 Register
11	0x2C	RxQ Server 1 Status Register
12	0x30	RxQ Server 2 Register
13	0x34	RxQ Server 2 Status Register
14	0x38	RxQ Server 3 Register
15	0x3C	RxQ Server 3 Status Register
16	0x40	TxQ Server 0 Register
17	0x44	TxQ Server 0 Status Register
18	0x48	TxQ Server 1 Register
19	0x4C	TxQ Server 1 Status Register
20	0x50	TxQ Server 2 Register
21	0x54	TxQ Server 2 Status Register
22	0x58	TxQ Server 3 Register
23	0x5C	TxQ Server 3 Status Register
24	0x60	Free Queue Register
25	0x64	Free Queue Status Register
26	0x68	Free Queue Lock Register

Table A.7: NIC_1.2 Queue Registers Map

Signal Name	**Location**	**Signal Description**
read/write_bar	[42:42]	Indicates the request type: '1' for read, '0' for write.
byte_mask	[41:38]	A 4-bit field where each bit corresponds to one byte of data. If a bit is set to '1', the respective byte in `write_data` is valid.
reg_index	[37:32]	Specifies the index of the target register for the read or write operation.
write_data	[31:0]	Contains the data to be written to the selected register.

Table A.8: NIC_1.2 to Register File interface description

Signal Name	**Location**	**Signal Description**
err	[32:32]	Set to '1' if the response indicates an error condition.
data	[31:0]	Contains the read data returned from the register if the request was a read operation.

Table A.9: Register File to NIC_1.2 interface description

A.4 Helper Modules

The **NIC_1.2** design incorporates several internal modules that facilitate synchronized memory access and queue management, enabling efficient packet processing. These modules primarily serve the NIC hardware and guarantee atomic execution where necessary.

1. **accessMemoryDword:** Handles all NIC-initiated memory accesses, performing 64-bit (double word) reads and writes.

2. **pushIntoQueue / popFromQueue:** Utilized by NIC hardware to push or pop buffer pointers into/from queues. These modules delegate to queue-specific command executors to ensure atomic and synchronized operations.

3. **exec_free_queue_command:** Executes commands on the shared `free_queue` such as push, pop, status queries, and lock/unlock. It is the sole executor implementing locking to provide mutual exclusion for concurrent access by processor and NIC.

4. **exec_receive_queue_command** and **exec_transmit_queue_command:** Perform push, pop, and status operations on `rx_queue` and `tx_queue`, respectively. No locking is necessary as these queues have exclusive access (NIC for Rx, processor for Tx).

5. **Processor Queue Access:** The processor does not directly call queue operation modules; instead, it interacts through the `accessRegister` interface, which issues queue commands via NIC_1.2's register file.

Appendix B

Tri-mode Ethernet MAC Example Design on KC705

To evaluate raw Ethernet performance without processor involvement, the Xilinx Tri-mode Ethernet MAC (TEMAC) IP core was used. Xilinx provides a dedicated example design for the KC705 evaluation board, enabling direct communication over the onboard Gigabit Ethernet interface [1].

This design operates entirely within the programmable logic, utilizing built-in packet generators and checkers. It serves as a processor-independent benchmark for measuring round-trip time (RTT) and throughput, offering a baseline for link-level performance.

Key Modules

- **TEMAC:** Implements the Ethernet MAC layer.
- **AXI-lite Control FSM:** Configures the MAC and PHY through AXI-lite for enabling data transfer.
- **Pattern Generator/Checker with Address Swapping:** Creates Ethernet frames, checks returned packets, and performs MAC address swapping for internal loopback.
- **AXI-Streaming FIFOs:** Buffer data between the MAC and traffic generator/checker modules.

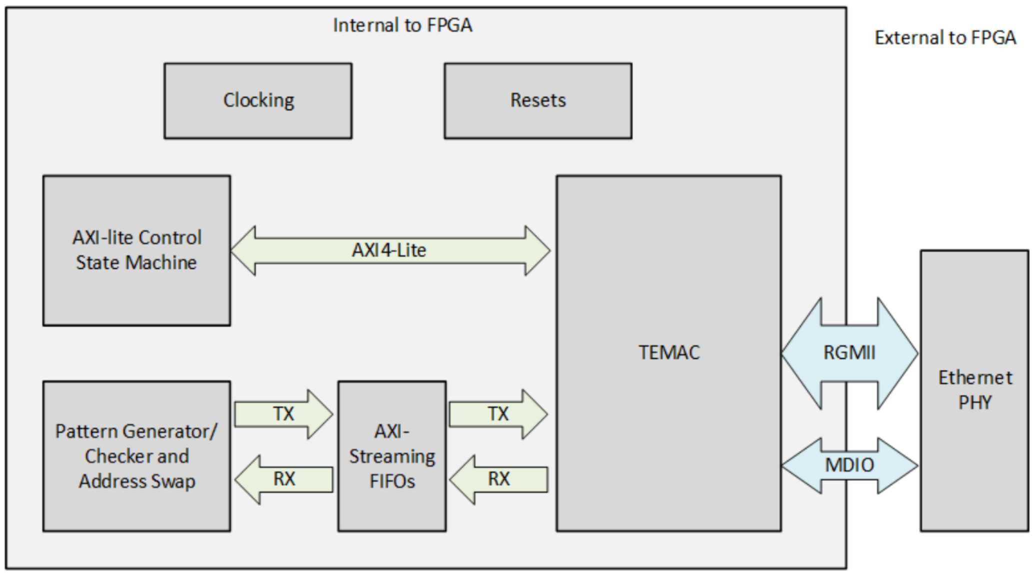

Figure B.1: Block diagram of the Tri-mode MAC IP example design [1].

This design enables hardware-only performance testing, minimizing software and processor overhead.

Round-Trip Time (RTT) Measurement

RTT is measured by looping back internally generated packets using MAC address swapping. The packet checker records the time taken for a packet to return after transmission.

Table B.1: Average RTT for Various Payload Lengths (TEMAC Example Design)

Payload Length (bytes)	**Average RTT (μs)**
64	99
128	109
256	120
512	137
1024	156

These values reflect the minimum latency achievable over the host-PC to FPGA Ethernet link without involving a software stack.

Raw Ethernet Throughput Measurement

The internal packet generator transmits a continuous stream of frames with a fixed 1 μs inter-frame delay. Throughput is computed using internal send/receive counters.

Table B.2: Raw Ethernet Throughput (TEMAC Example Design)

Payload Length (bytes)	**Send Throughput (Mbps)**	**Receive Throughput (Mbps)**
64	144.09	144.09
128	170.36	170.36
256	487.85	487.85
512	867.51	867.51
1024	852.00	852.00
1486	888.19	888.19

These results demonstrate high efficiency and near-saturation of a 1 Gbps Ethernet link, confirming the capability of a hardware-only design for achieving consistent, low-jitter data rates.

Appendix C

Transmission Delay Measurement on Host-PC to FPGA Link

This chapter presents an empirical analysis of the minimum delay between consecutive packet transmissions from a host PC to an FPGA. The measurements are critical for understanding timing constraints and traffic modeling during validation of the NIC integrated with the AJIT processor.

C.1 Experiment Overview

To estimate the lower bound on inter-packet transmission delay, a Python script was used to test the resolution and accuracy of Python's `time.sleep()` function for very short intervals. The objective was to measure how closely the actual delay matches the requested delay and identify the smallest achievable gap between two packet sends in a software-based test environment.

```
import time

def measure_sleep(delay):
    start_time = time.perf_counter()
    time.sleep(delay)
    end_time = time.perf_counter()
    return end_time - start_time

delays = [1e-324, 1e-323, 1e-9, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e
    -2, 1e-1]
```

```python

print("Requested vs Actual Delay:")
for delay in delays:
    actual = measure_sleep(delay)
    print(f"Requested: {delay:.1e} s, Actual: {actual:.5e} s")
```

Listing C.1: Python Script for Measuring Sleep Delay

C.2 Results

The table below lists the requested delay values alongside the actual delays measured during execution on a typical desktop PC:

Requested Delay (s)	Actual Measured Delay (s)
0.0	4.25e-06
1e-323	5.98e-05
1e-09	5.68e-05
1e-08	5.61e-05
1e-07	5.59e-05
1e-06	5.59e-05
1e-05	6.62e-05
1e-04	1.55e-04
1e-03	1.05e-03
1e-02	1.02e-02
1e-01	1.00e-01

Table C.1: Requested vs Actual Delays Using `time.sleep()`

C.3 Key Observations

- A minimum delay ranging from about 5 microseconds (for zero delay requests) up to around 60 microseconds (for very small delays) occurs.
- This baseline delay is caused by OS scheduling and interpreter overhead, limiting packet rates in Python-based test scripts.
- Delay accuracy improves for requests above 1 millisecond, where actual sleep times closely match the requested duration.

Implication: Software-based testbenches (e.g., Python scripts) must consider this inherent delay when measuring performance or throughput, as it limits precise emulation of high-frequency traffic without specialized low-level or real-time mechanisms.

References

[1] J. Johnson, "Driving ethernet ports without a processor." `https://www.fpgadeveloper.com/driving-ethernet-ports-without-a-processor/`, 2016.

[2] S. S. Tomar, "Towards an soc architecture for software-defined networking (sdn)," 2024.

[3] M. P. Desai, *The AJIT processor*. IIT Bombay.

[4] *The SPARC Architecture Manual*.