Evaluating Memory Sharing Architectures Between NIC and Processor to Enhance Packet Processing

Submitted in partial fulfillment of

Master's Thesis Stage - I (EE-797)
by

Siddhant Singh Tomar
Roll No. 213079010

under the supervision of
Prof. Madhav P. Desai

Department of Electrical Engineering

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY
Powai, Mumbai - 400076
October 2023

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/-source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

.....
Mr. Siddhant Singh Tomar
Roll No. 213079010

Date : 2024-06-21

Acknowledgment

I want to express my heartfelt gratitude towards **Prof. Madhav P. Desai** for allowing me to work on this project and providing her valuable guidance throughout. His suggestions have helped me in gaining a better understanding of this research topic. Lastly, I am perpetually thankful to my family and friends for their unwavering encouragement and support.

Abstract

This thesis examines the implications of sharing memory between the processor and the Network Interface Card (NIC) at a level closer to the CPU than the main memory (DRAM) to enhance the packet processing rate. Two architectures are explored: one where fast local packet memory, is shared by the processor and NIC and another where NIC directly writes to the main memory via L2 cache. The performance of these architectures is evaluated using a standard ping application and a Network-attached storage (NAS) caching application, running on a TCP/IP stack.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The interaction between processors, memory, and NIC adapters involves various data structures and system mechanisms. Guided by software, the processor sets up a main memory buffer for transmission or reception and provides the NIC with a descriptor, containing a pointer to the buffer. This enables the NIC to read or write data to/from the main memory. Descriptors are typically organized in a circular ring structure. The NIC, equipped with a descriptor, can independently complete block transfers and update the processor with a status in the main memory.

Modern systems employ various optimizations to reduce the overhead of memory access latency, although these are implementation-specific. This thesis focuses on main memory-based interactions involving packet descriptors, circular rings, and payload data structures, noting that memory latency alone can limit the processor's ability to achieve full network I/O rates. To address this limitation, the thesis investigates a novel approach to enhancing packet processing rates by sharing memory between the processor and the NIC at a level closer to the CPU than traditional main memory (DRAM), thereby reducing memory access latency and improving the overall packet processing rate.

Chapter 2

Baseline Architecture: Direct NIC to Main Memory

2.1 Introduction

This chapter introduces the integration of the NIC with the rest of the SoC. The architecture presented here will serve as the baseline for comparison with other shared memory architectures. This chapter includes the work done in M.Tech Project Phase-I. In the first phase, we have taken a first step towards realizing our goal of creating a System on Chip (SoC) that integrates computing, networking, and storage. This system will serve as the template for future system/architecture exploration.

2.2 SBC architecture

2.3 Features of SBC

The main subsystems of SBC are as follows:

1. AJIT (1x1x32) processor subsystem.
2. Tri-mode Ethernet MAC IP and NIC subsystem.
3. AFB(Ajit Fifo Bus) and ACB(Ajit Core Bus) complex.

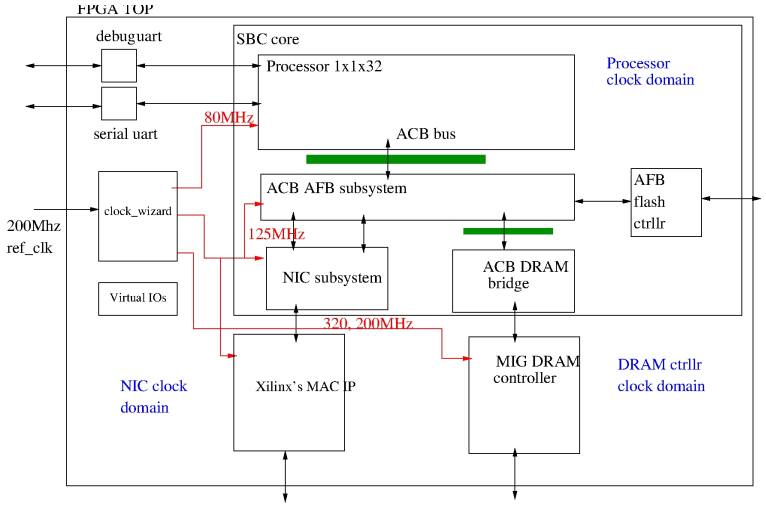

Figure 2.1: Architecture of single board computer with AJIT 1x1x32

4. ACB DRAM bridge and MIG series DRAM controller.

5. AFB FLASH controller and Flash memory.

2.3.1 Processor 1x1x32

1. AJIT processor central processing unit (CPU): implements the SPARC-V8 ISA (Draft IEEE standard 1754-1996).

2. Instruction Cache (ICACHE): A 32-kB (64-byte line size), direct mapped, virtually indexed, and virtually tagged instruction cache.

3. Data Cache (DCACHE): A 32-kB (64-byte line size), direct mapped, vir- actually indexed and virtually tagged data cache with write-through allocate policy.

4. Memory management unit (MMU): implements all required aspects of the SPARC reference MMU. A 64-bit AJIT Core Fifo Bus interface is provided.

2.3.2 NIC subsystem

The NIC is linked to a tri-mode Ethernet MAC IP in order to capture Ethernet frames via the NIC-MAC bridge.
Processor to NIC slave interface: The processor will allocate the memory space for packet storage and provide that info to NIC using this interface. NIC will have registers inside which will written by the processor using this interface. AFB(AJIT FIFO Bus)

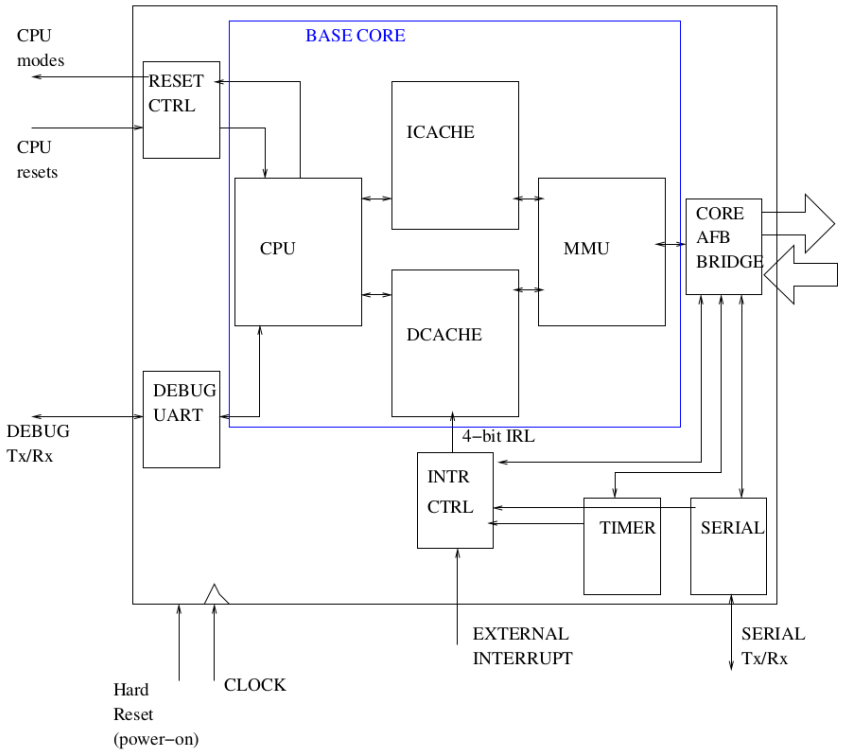

Figure 2.2: The Generic 32-bit AJIT processor core

protocol will be used for this.

NIC to Memory master interface: The NIC-Memory interface is required for storing and loading packets to and from memory. Already developed ACB(AJIT Core Bus) protocol will be used for this.

2.3.3 ACB AFB Bus Complex

The primary interfaces between the AJIT processor and the external world are:

1. AJIT CORE BUS interface (ACB): A 64-bit data, 36-bit address system memory interface.

2. AJIT FIFO BUS interface (AFB): A 32-bit data, 36-bit peripheral/memory interface.

Both these bus interfaces involve a request and a response interface. The bus master sends a request word to the slave, which responds to the request with a response word.

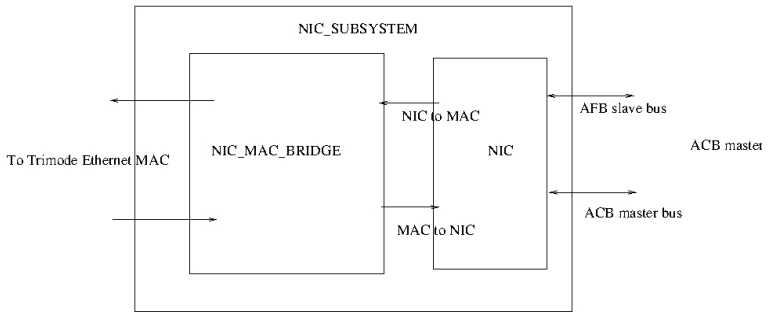

Figure 2.3: NIC Subsytem

The exchange of data between the master and slave is regulated using a simple two-wire protocol.

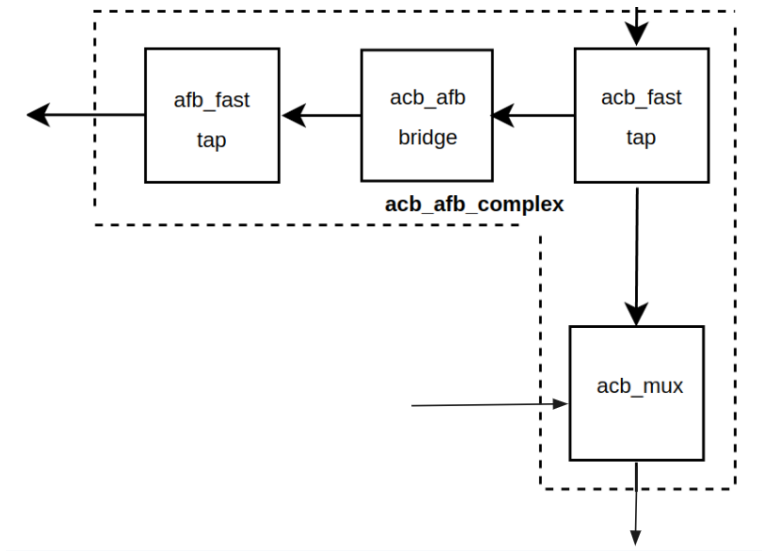

Figure 2.4: ACB AFB bus complex.

2.3.4 Flash Memory contoller

This controller provides a simple AFB interface to a serial SPI Flash memory such as the ones on the Kintex 705 FPGA cards from Xilinx (See Xilinx doc- umentation note XAPP586 from www.xilinx.com).

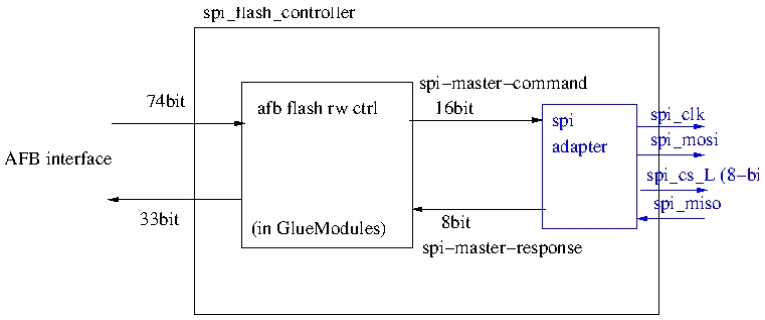

Figure 2.5: SPI Flash controller

2.4 Clock Domains in SBC

The single-board computer (SBC) is designed with three distinct clock domains. The Processor subsystem operates at 80MHz, the Dram controller functions at 200MHz, and the NIC subsystem operates at 125MHz. To facilitate the connection between these subsystems, Dual Clock asynchronous FIFOs are employed, one for ACB Request and another for ACB Response. The system is organized in a manner that minimizes the quantity of FIFOs needed. Clock signals are distributed throughout the system via clock wizard IPs.

Chapter 3

Architecture A: NIC and Processor Shared Fast Local Memory

3.1 Introduction

This chapter explores how using main memory for packet descriptors, circular rings, and payload data can slow down network performance due to latency. To address this, we introduce a new approach where the NIC accesses main memory through a 256 KB L2 cache, bringing memory closer to the CPU and improving network I/O rates.

3.2 Packet Reception

Processor (driver code) talks to network interface card (NIC) to exchange packets. Processor(driver code) maintains rings of packet descriptors in main memory(DRAM). Driver populates the ring with empty packet buffer pointers. NIC pops an empty buffer pointer from ”free queue”, and writes data directly into the buffer (residing in main memory), and pushes the free buffer to ”receive queue”. Processor continuously polls the ”receive queue”, for new packets. After reading the filled buffer pointer, and processing the packet, the buffer pointer is pushed back to ”free queue”.

3.3 Packet Transmission

3.4 Packet Transmission

3.5 Proposed Architecture

A tentative diagram of the architecture is shown below.

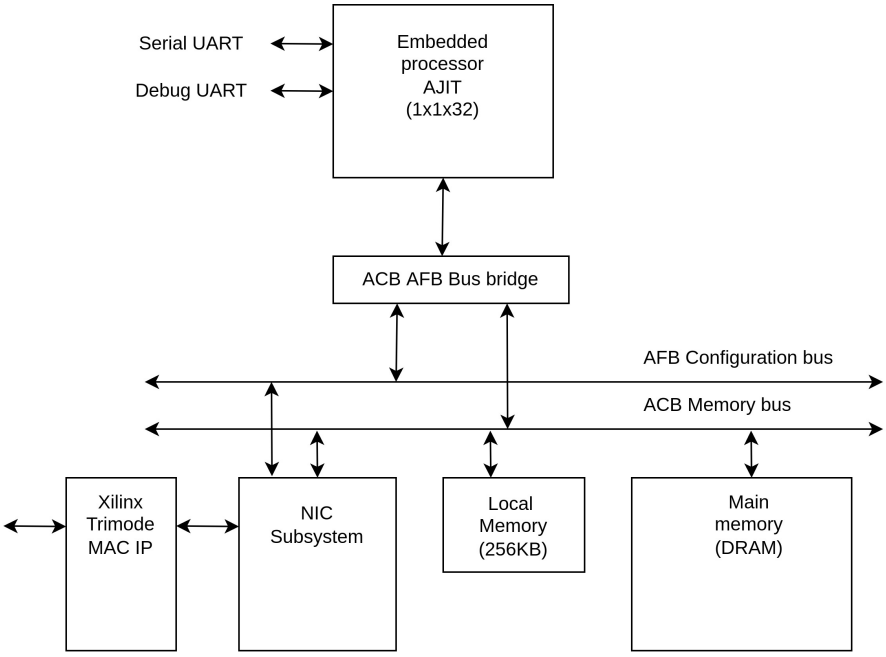

Figure 3.1: Top level with Interfaces

3.6 Performance on Ping application

3.6.1 RTT

3.6.2 various packet sizes

3.7 Performance with NAS caching application

3.7.1 upload speed

3.7.2 download speed

Chapter 4

Architecture B: NIC Accessing DRAM via L2 Cache

4.1 Introduction

This chapter examines main memory-based interactions involving packet descriptors, circular rings, and payload data structures, highlighting how memory latency can impede the processor's ability to reach optimal network I/O rates. To mitigate this limitation, a novel approach is introduced, where memory is shared between the processor and the NIC at a level closer to the CPU than conventional main memory (DRAM). Specifically, rather than utilizing the main memory, the NIC and processor share a fast local packet memory of 256KB, which contains packet descriptor rings and packet buffers.

4.2 Packet Recption

Processor (driver code) talks to network interface card (NIC) to exchange packets. Processor(driver code) maintains rings of packet descriptors in main memory(DRAM). Driver populates the ring with empty packet buffer pointers. NIC pops an empty buffer pointer from ”free queue”, and writes data into the buffer, and pushes the free buffer to ”receive queue”. Processor continuously polls the ”receive queue”, for new packets. After reading the filled buffer pointer, and processing the packet, the buffer pointer is pushed back to ”free queue”.

4.3 Proposed Architecture

A tentative diagram of the architecture is shown below.

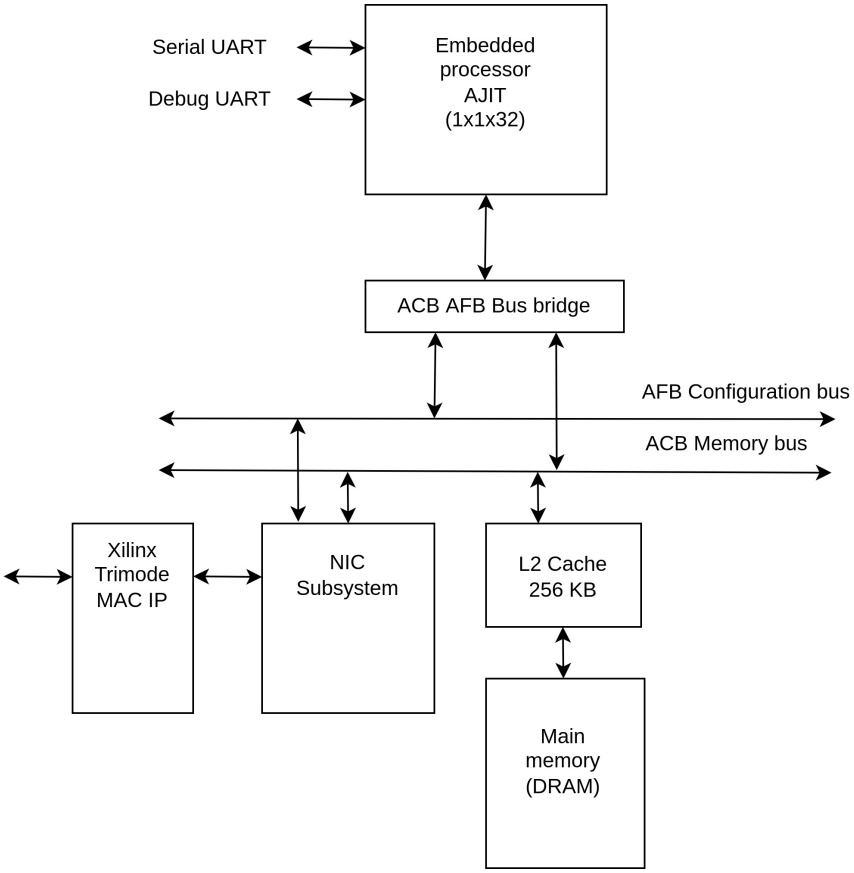

Figure 4.1: Top level with Interfaces

4.4 Performance on Ping application

4.4.1 RTT

4.4.2 various packet sizes

4.5 Performance with NAS caching application

4.5.1 upload speed

4.5.2 download speed

Chapter 5

Impact of Zero Copy Rx on Proposed Architectures

Chapter 6

benchmarking application overview using lwIP

Chapter 7

Conclusion & Future Work

7.1 Conclusion

Network appliances capable of delivering computing capacity, networking capabilities, and storage solutions are ideally suited for being infrastructure endpoints. It can enable the Disaggregation of network infrastructure services and applications to provide improved performance and security.

7.2 Future Work

1. Adding IPv4 Forwarding information base (FIB) and ARP tables to NIC.
2. Design of "NIC Switch" to enable NIC-to-NIC communication.
3. Identifying the parts to offload to hardware, to free up the processor.
4. Driver code to populate the NIC forwarding caches.

.

Appendix A

Standard NIC (Network Interface Controller) Design - By MPD

This chapter focuses on the design and validation of the Network Interface Controller (NIC) for the AJIT processor-based System-on-Chip (SoC). The NIC plays a critical role in enabling network connectivity and communication capabilities within the SoC.

The design of the NIC involves several components and considerations, including network protocol support, data packet handling, memory management, and interface with the AJIT processor. To ensure a robust and reliable design, the NIC undergoes a rigorous validation process to verify its functionality, performance, and compatibility with the AJIT SoC architecture.

The primary goal of this chapter is to provide a detailed overview of the design process, highlighting the key components and their interconnections. It explores the challenges encountered during the design phase and discusses the solutions and design choices made to overcome them. Additionally, the chapter delves into the validation methodologies employed to ensure the correctness and efficiency of the NIC design.

By describing the design and validation of the NIC, this chapter aims to provide insights into implementing a network interface solution for the AJIT processor-based SoC. It serves as a foundation for subsequent chapters, which will explore specific aspects of the NIC design, such as the data packet handling mechanisms, and memory management schemes.

A.1 Design decisions

Before directly jumping on NIC design let's take a look at the necessary design decisions made. The NIC will receive packet data from MAC which will be stored in memory. The processor will need to allocate this memory and provide that information to NIC. This overall needs to 3 main interfaces to NIC. Figure A.1 shows all the interfaces.We examine each interface in detail.

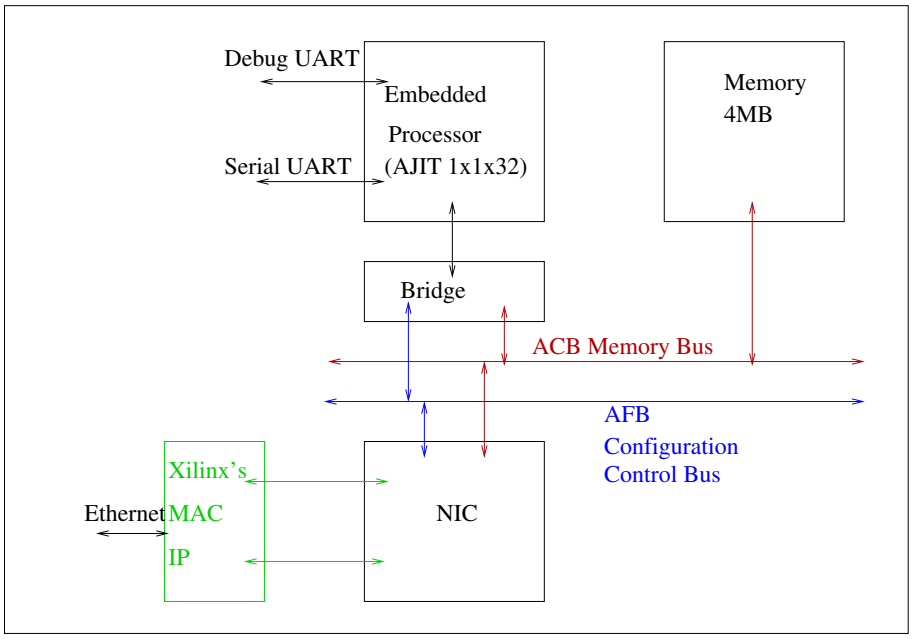

Figure A.1: Top level with Interfaces

A.1.1 NIC-MAC interface

NIC to MAC interface will used by NIC for receiving and transmitting packets from MAC. The memory which will be used is provided 8 bytes per request. So this interface is kept 73(64 bit data + 9 bit control) bits. The bit mapping is shown in table A.1. The same interface will be used for both reception and transmission of packets.

A.1.2 NIC-Memory interface

The NIC-Memory interface is required for storeing and loading packets to and from memory. Already developed ACB(AJIT Core Bus) protocol will be used for this. The protocol consists of two interfaces,

Signal Name	Location	Signal Description
tlast	[72:72]	The *tlast* becomes '1' if the 64 bit chunk is last chunk of packet.
tdata	[71: 8]	The *tdata* is actual packet data chunk.
tkeep	[7: 0]	The *tkeep* is 8 bit field, each bit is mapped to 8 bytes of data. If any bit is '1' then corresponding byte in data is valid else not.

Table A.1: NIC-MAC interface description

1. ACB Requeuest : Requests from NIC to store and load the packet will be sent through this interface. see table A.2 for bit mapping.
2. ACB Response : Response generated by memory to the request will be sent back to NIC on this interface. see table A.3 for bit mapping.

A.1.3 NIC-Processor interface

This will be more of a control interface. Processor will allocate the memory space for packet storage and provide thaat info to NIC using this interface. NIC will have registers inside which will written by the processor using this interface. For further information see section A.3.1. An already developed AFB(AJIT FIFO Bus) protocol will be used for this. This AFB protocol also has two interfaces like ACB protocol only the address width is half.

1. AFB Requeuest : Requests from Processor to write or read the NIC reg will be sent through this interface. see table A.4 for bit mapping.
2. AFB Response : Response generated by NIC to the request will be sent back to Processor on this interface. see table A.5 for bit mapping.

Signal Name	Location	Signal Description
lock	[109:109]	Lock bit, if set to '1' by a master then other master's don't get access to Memory.
read/write_bar	[108:108]	If '1', the request is read request, if '0', the request is write request.
byte_mask	[107: 100]	The *byte_mask* is 8 bit field, each bit is mapped to 8 bytes of data. If any bit is '1' then corresponding byte in data is valid else not.
address	[99:64]	The addres(byte) where read/write should be performed.
write_data	[63: 0]	Data to be written.

Table A.2: NIC - Memory interface description

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

Table A.3: Memory - NIC interface description

Signal Name	Location	cSignal Description
lock	[73:73]	Lock bit, if set to '1' by a master then other master's don't get access to Memory.
read/write_bar	[72:72]	If '1', the request is read request, if '0', the request is write request.
byte_mask	[71: 68]	The *byte_mask* is 4 bit field, each bit is mapped to 4 bytes of data. If any bit is '1' then corresponding byte in data is valid else not.
address	[67:32]	The addres(byte) where read/write should be performed.
write_data	[31: 0]	Data to be written.

Table A.4: Processor - NIC interface description

Signal Name	Location	Signal Description
err	[32:32]	Value '1' indicates errored response.
data	[31: 0]	Contains read data if the req. was read req.

Table A.5: NIC - Procesor interface description

A.2 Interface data structures

The interface data structures used in the NIC design consist of three queues: the free_queue, rx_queue, and tx_queue.

- **free_queue**: This queue holds the addresses of free buffers that are available for storing packets. The processor initially assigns a set of buffers and pushes their addresses into the free_queue. These buffers do not have any active packets and are ready to be utilized for storing incoming packets. Both the processor and the NIC can push and pop from the free_queue.

- **rx_queue**: The rx_queue is pushed by the NIC and popped by the processor. It holds the addresses of buffers that currently contain active packets. When the NIC receives a packet, it stores the packet in a buffer and pushes the address of that buffer into the rx_queue. This allows the processor to identify the buffers with active packets that are ready for processing.

- **tx_queue**: The tx_queue is pushed by the processor and popped by the NIC. Once the processor has finished processing a packet, it pushes the address of the processed packet buffer into the tx_queue. The NIC monitors the tx_queue and retrieves the buffer addresses from it to send the packets out over the network.

These queues enable efficient coordination and communication between the processor and the NIC, ensuring the proper handling and processing of packets.The queue header format is shown in Algorithm **??**,

To ensure synchronization and prevent conflicts during access to these queues, a locking mechanism is implemented. The locking mechanism utilizes atomic operations, which guarantee thread-safe access and modifications to the queues. This ensures that only one entity can perform push and pop operations on the queues at a given time, preventing simultaneous modifications and preserving the integrity of the queue data.

At startup, the processor initializes the queues by allocating memory for them and configuring their initial state. The addresses of these queues are then communicated to the NIC by writing to specific NIC registers. This process allows the NIC to access and manipulate the queues effectively during runtime.

A.3 Network Interface Controller

A.3.1 Register File Map

The NIC (Network Interface Controller) registers are specific memory locations within the NIC that are used for configuration, control, and status monitoring purposes. These registers allow communication between the processor and the NIC, enabling the processor to control and monitor NIC. The NIC registers provide a standardized interface

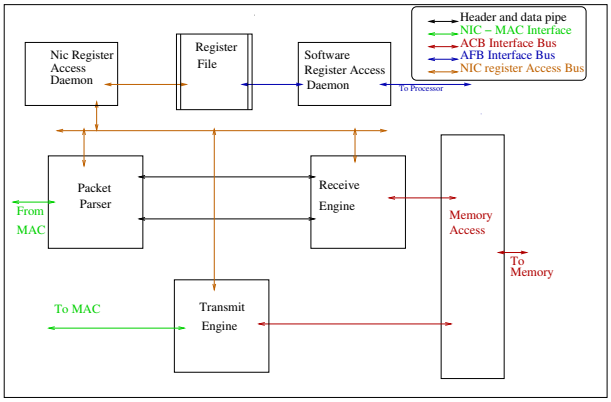

Figure A.2: Architecture of Network Interface Controller

for the processor to interact with the NIC and perform tasks such as enabling or disabling the NIC, setting queue address and monitoring the status of data transmission and reception. See table A.6 for description of NIC registers.

Reg. ID	Address offset	Description
0	0x00	Control reg
1	0x04	Number of Servers
2	0x08	Address of rx_queue
10	0x28	Address of tx_queue
18	0x48	Address of free queue

Table A.6: NIC registers map

A.3.2 Parser

The parser daemon receives data from the MAC (Media Access Control) through a pipe and extracts relevant information from Ethernet packets. It utilizes a state machine to process chunks of data and identifies essential details such as source and destination MAC addresses, type/length field, and packet data. The parsed information is then forwarded to receive engine daemon. The algorithm **??** shows psuedo code of parser.

A.3.3 Receive Enigne

The receive engine daemon is responsible for the storage of packets coming from the parser. It receives the parsed packet information from the parser daemon and interacts with the processor to ensure the proper handling of received packets. The receive engine daemon uses the free_queue, to get an empty buffer address to store the active packets in buffers. Then uses rx_queue to provide their(buffer's) addresses to the processor for processing. The algorithm **??** showspsuedo code of receive engine,

A.3.4 Transmit Engine

The transmit engine daemon focuses on transmitting processed packets from the processor to the external network. It receives the addresses of processed packet buffers from the processor via the tx_queue and sends the corresponding packets out through the Ethernet interface. The transmit engine daemon monitors the tx_queue and retrieves the buffer addresses to facilitate efficient packet transmission. The daemon also pushes free_queue with the address of buffers which is sent out. This allows the reuse of buffers. The algorithm **??** shows psuedo code of transmit engine.

A.3.5 Software register Access

The software register access daemon enables the the processor to access and modify the NIC's software registers. These registers contain various control and configuration parameters, allowing the processor to configure and manage the behavior of the NIC. The software register access daemon handles the communication between the processor and the NIC registers, ensuring reliable and secure access. The processor uses AFB protocol(see section A.1.3) for register access.

A.3.6 Nic register Access

The NIC register access daemon provides the necessary interface for the NIC to read from and write to its internal registers. These registers store critical information for the proper functioning of the NIC, including configuration settings, status flags, and other control parameters. The NIC register access daemon ensures that the processor can interact with these registers and modify them as needed to configure and manage

the NIC's behavior.

Signal Name	Location	cSignal Description
read/write_bar	[42:42]	if '1', the request is read request, if '0', the request is write request.
byte_mask	[41: 38]	*byte_mask* is 4 bit field, each bit is mapped to 4 bytes of data. if any bit is '1' then corresponding byte in data is valid else not.
reg_index	[37:32]	index of register to which read/write should be performed.
write_data	[31: 0]	Data to be written.

Table A.7: NIC to Register file interface description

Signal Name	Location	Signal Description
err	[32:32]	Value '1' indicates errored response.
data	[31: 0]	Contains read data if the req. was read req.

Table A.8: Register file to NIC interface description

A.4 Helper Modules

1. **memoryAccess**: This module can be called from other modules and used to read and write from and to memory.
2. **pusiIntoQueue**: This module is used to push buffer pointer to provided queue.
3. **popFromQueue**: This module is used to pop buffer pointer form provided queue.

4. **acquireLock**: This module is used to lock the queue for push and pop operation. It first reads the queue lock by setting memory lock to '1' and then acquires queue lock if its available and then releases memory lock.

5. **releaseLock**: This module is used to release the acquired queue lock.

A.5 Validation and Performance

The data rate found using 1x1x32 (single core single-threaded) AJIT processor are as shown in table A.9. The peak performance achived in this case is 114.81Mbps.

Table A.9: Data rate achieved for differnet number of packets & packet sizes

	Packet Size(in Bytes)		
No. of Packets	**48**	**136**	**236**
	Data Rate(in Mbps)		
1	11.3316	31.7085	51.2869
10	18.8431	57.2255	102.4069
100	22.4589	64.1140	110.9845
1000	23.0982	64.0665	114.4293
10000	23.0892	64.3163	114.7907
50000	23.0983	64.4662	114.8123

To avoid the processor from locking queues the setup was designed where processor will first generate 1750 packets of size 146 bytes, and push them to tx_queue. NIC will read the tx_queue and send the packets out during this time. The peak performance achieved is 433.2847 Mbps in this case.

This chapter focused on NIC architecture and interfaces which was then followed by NIC's performance. Now we describe the design and validation of SoC using this NIC and an accelerator in the next chapter.