Towards an SoC Architecture for Software-Defined Networking (SDN)

Dissertation submitted in the fulfillment of

Master of Technology (M.Tech)
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

Dissertation Approval

This dissertation entitled

Towards an SoC Architecture for Software-Defined Networking (SDN)

by

Mr. Siddhant Singh Tomar
Roll No. 213079010

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: 2024-07-18
Place: IIT Bombay

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 : 2024-07-18

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

In this thesis, we describe the design and implementation of a System on Chip (SoC) platform for Software-Defined Networking (SDN). The SoC integrates a 32-bit AJIT processor with a custom network interface controller (NIC) and high-capacity main memory (DRAM). The entire SoC was implemented and validated on a Xilinx KC705 FPGA. To characterize the performance of the SoC, we used two applications: Ping and Network Content Cache, utilizing the open-source TCP/IP stack called LwIP.

Our findings indicate that the performance of these applications is influenced by the architecture of the memory subsystem. To enhance the SoC's performance, we explored two memory subsystem architectures between the NIC and the processor. The first architecture employs a fast local packet memory shared by both the processor and the NIC. The second architecture uses L2 cache in the processor and NIC's access path to the main memory. Both architectures resulted in performance improvements, with the first architecture performing better for the applications considered. Thus, the first architecture can serve as a foundational design for SoCs aimed at high-performance networking, with potential for further optimizations.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

The advent of Software-Defined Networking (SDN) has revolutionized network design, management, and optimization. By decoupling the control plane from the data plane, SDN allows for more flexible and dynamic network management. As SDN continues to evolve, the demand for efficient and high-performance System-on-Chip (SoC) platforms tailored for SDN applications has surged.

In this thesis, we have focussed on the design and implementation of such an SoC platform, which can provide computing, networking, and storage capabilities for SDN applications. The SoC platform includes a 32-bit AJIT processor integrated with a custom network interface controller (NIC) and a large-capacity main memory (DRAM). This entire SoC setup was implemented and validated on a Xilinx KC-705 FPGA. To assess the performance of the SoC, we employed two applications—Standard Ping and Network Content Caching. We verified the two applications in a bare-metal environment, utilizing the lwIP (lightweight IP) open-source TCP/IP stack, which was ported for the AJIT processor and custom NIC. lwIP (lightweight IP) is a widely used open-source TCP/IP stack designed for embedded systems.

We have studied the impact of the memory subsystem architecture on the performance of these two applications. To enhance the overall data plane performance, we investigated two modifications to the memory subsystem architecture. The first architecture employs a fast local packet memory shared by both the processor and the NIC. The second architecture uses L2 cache in the processor and NIC's access path to the main memory.
Both architectures demonstrated performance improvements. Specifically, the first architecture achieved a 34.5% speedup, while the second architecture realized a 18.3% speedup in the considered applications compared to our baseline architecture. Therefore, the first architecture is recommended for designing the memory subsystem to ensure sufficient fast buffering for packets in the data plane.
Further optimization opportunities can be explored by enhancing the architecture to offload additional computational tasks to the NIC. This approach could streamline processing within the SoC, leveraging the NIC's capabilities to handle specific tasks more efficiently.

1.2 Objectives

This research aimed to develop an SoC architecture capable of integrating computing, networking, and storage functionalities for SDN applications. This SoC includes a single-core 32-bit AJIT processor, along with a custom-designed NIC and DRAM subsystem for storage. The architecture of this SoC serves as a foundational framework for constructing more advanced SoCs featuring multiple processor cores and NICs for high-performance networking and further research.

1.3 Organization of the thesis

In Chapter 2 of this thesis, the detailed organization and performance characterization of the baseline SoC architecture, specifically the single board computer (SBC), are presented. In Chapter 3, we discuss the optimizations made to the baseline architecture, with a focus on the memory subsystem, and provide their performance characterization. In Chapter 4, a comparative analysis of the three architectures is conducted, along with a discussion on future work for further optimization. The Appendix contains the detailed design of the NIC and the network caching application.

Chapter 2

Baseline Architecture of the SoC

2.1 Introduction

This chapter includes a detailed description of the baseline architecture of the SoC, which is essentially a single-board computer (SBC), incorporating a 32-bit AJIT processor integrated with a custom-designed network interface controller (NIC) and a memory subsystem (DRAM), together with some peripherals. This chapter also gives a brief description of the custom NIC, designed by Harshad Ugale at IITB, which was modified and optimized as part of this project. The integration and working of the NIC with the rest of the SoC is thoroughly discussed. The SoC described in this chapter, will serve as the template for further system/architecture exploration. The architecture presented here will serve as the baseline for comparison with other memory subsystem architectures.

2.2 The Single board computer (SBC)

The architecture of a single-board computer (SBC) is organized into a ”SBC core” and a ”SBC periphery” as shown in Figure 2.1. The ”core” comprises a single-threaded 32-bit AJIT processor, a custom-designed NIC, an ACB, and an AFB bus complex. The ”periphery” includes the Xilinx tri-mode Ethernet MAC IP, which interfaces with both the NIC subsystem in the core and the PHY on the FPGA, and the Xilinx MIG DRAM controller IP, which connects to the ACB DRAM bridge in the core and manages the DRAM memory on the FPGA..

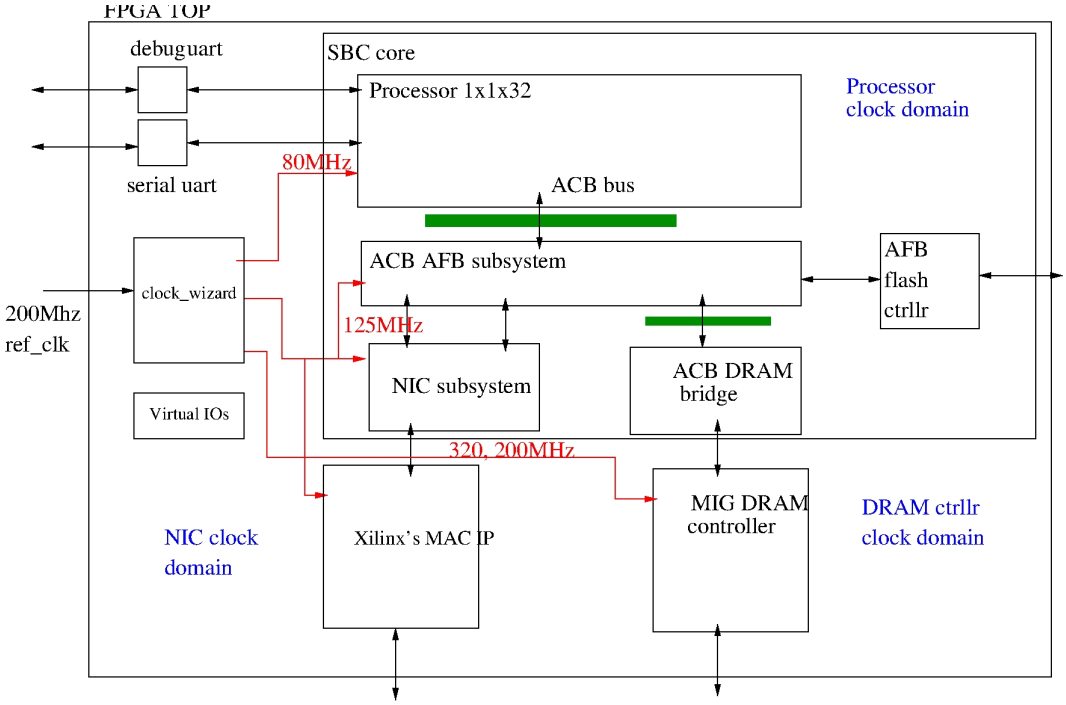

Figure 2.1: Architecture of single board computer with AJIT 1x1x32

2.3 SBC Core

The main subsystems of the "core" in SBC are as follows:

1. AJIT (1x1x32) processor subsystem.
2. Custom Network Interface Controller (NIC) subsystem.
3. Bus complex consisting of 32-bit AFB and 64-bit ACB buses.

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.

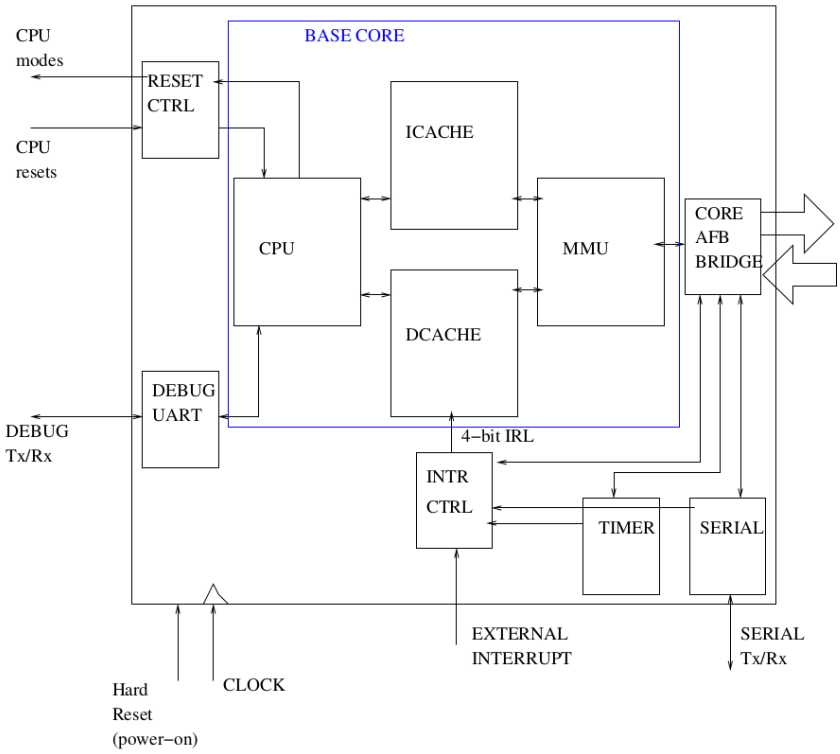

Figure 2.2: The Generic 32-bit AJIT processor core

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 subsystem comprises the custom-designed NIC and NIC-TO-MAC-BRIDGE.

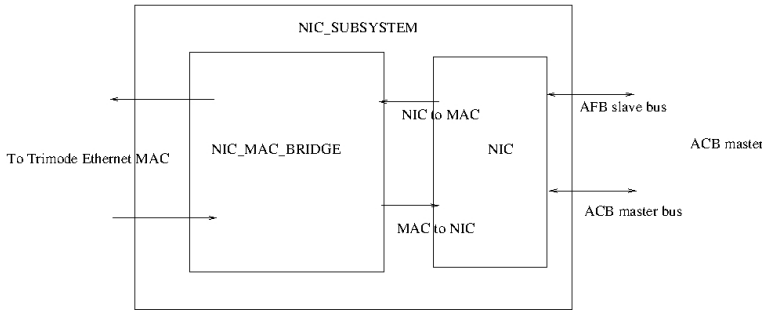

Figure 2.3: NIC Subsytem

The various interfaces that NIC exposes to rest of the SoC are discussed below:

1. **Processor to NIC slave interface**: The processor will allocate memory regions from the address space for packet buffers and queue data structures that hold pointers to these packet buffers. It also configures the NIC's registers with the physical addresses of the queue data
2. **NIC to Memory master interface**: The queues are populated with the physical addresses of the packet buffers, allowing the NIC to write directly to the main memory using the ACB (AJIT CORE Bus 64-bit) protocol.
3. **NIC subsystem To MAC interface**: The NIC subsystem is linked to a tri-mode Ethernet MAC IP to capture Ethernet frames via the NIC-MAC bridge. Tri-mode Ethernet MAC IP connects to ethernet PHY on the FPGA.

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

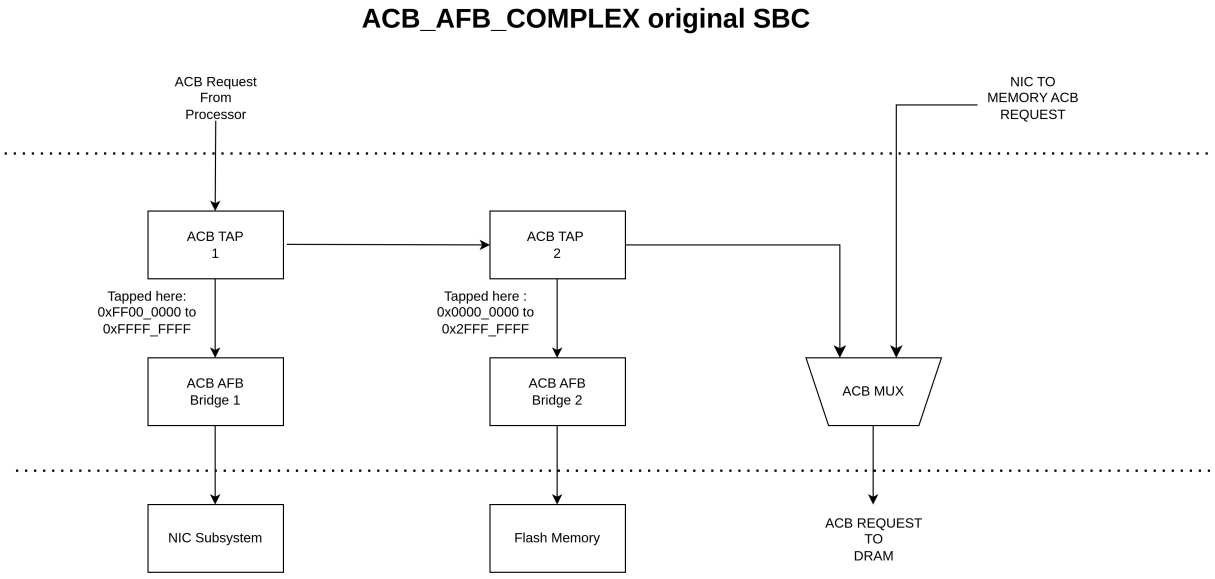

Figure 2.4: ACB AFB bus complex.

2.4 SBC periphery

The peripherals contain the Xilinx IPs used in the SBC:

1. Xilinx Tri-mode ethernet MAC IP
2. Xilinx MIG DRAM controller IP

2.4.1 Xilinx tri-mode ethernet MAC IP

The block diagram above shows the various elements of the example design and where they physically reside. Here is a basic description of the significant blocks and their purpose in the design. The Tri-Mode Ethernet Media Access Controller (TEMAC) solution comprises the 10/100/1000 Mb/s and 10/100 Mb/s Intellectual Property (IP) cores along with the optional Ethernet AVB Endpoint which are fully verified designs. A MAC is responsible for the Ethernet framing protocols and error detection of these frames. The MAC is independent of, and can be connected to, any type of physical layer.

1. **TEMAC**: The Tri-Mode Ethernet MAC implements the MAC (media access control) layer of the IP stack, a sublayer of the data link layer.

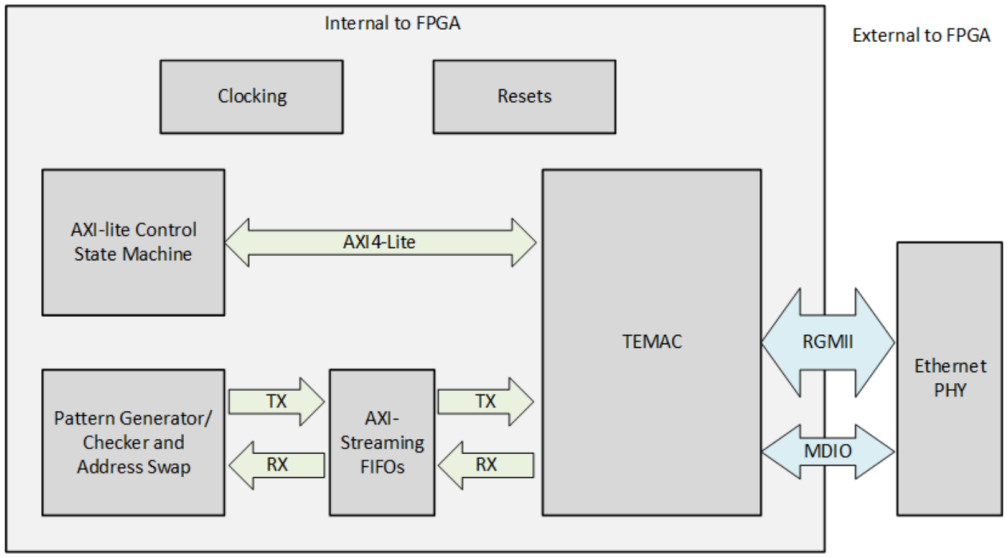

Figure 2.5: Block diagram of the Trimode MAC IP.

2. **AXI-lite Control State Machine**: The state machine performs basic transactions over the AXI-lite interface of the TEMAC, and brings up the MAC and the external Ethernet PHY to allow basic frame transfer.

3. **AXI-Streaming FIFOs**: The FIFOs add buffering between the TEMAC and the packet generator/checker.

2.4.2 Xilinx MIG DRAM controller IP

The MIG 7 Series DDR3/DDR2 LogiCORE IP is provided as a full memory interface design with a physical layer (PHY), a highly efficient memory controller, and user interface blocks. All blocks are provided as HDL source code. Xilinx supports using the PHY-only portion of the MIG 7 Series IP to interface with the custom controller.

2.5 Clock Domains in SBC

The single-board computer (SBC) is designed with two distinct clock domains. The Processor subsystem and DRAM memory subsystem operate at 80MHz, and the NIC subsystem operates at 125MHz. To facilitate the connection between these subsystems, Dual Clock asynchronous FIFOs are employed (depicted as green boxes in Figure 2.1), 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.

2.6 System organization: Working of Processor and NIC in packet processing

This section provides an overview of the mechanisms in packet reception and transmission, involving a Network Interface Card (NIC) and the processor.

2.6.1 Packet Reception

In packet reception, the processor, through the driver code, interacts with the Network Interface Card (NIC) to manage the flow of incoming network packets. The processor and NIC utilize a system of packet descriptors organized in rings within the main memory (DRAM).

1. Packet Descriptor Rings: The processor, specifically the driver code, maintains rings of packet descriptors in DRAM. These descriptors are essentially metadata structures that describe where packet data should be stored.

2. Initialization with Empty Buffers: The driver populates these descriptor rings with pointers to empty packet buffers. These buffers are areas in memory where incoming packet data will be stored.

3. NIC Data Writing Process: When a packet arrives, the NIC takes an empty buffer pointer from a structure known as the ”receive free queue”. The NIC then writes the incoming packet data directly into the buffer in main memory. After filling the buffer, the NIC moves the buffer pointer to the ”receive queue”.

4. Processor Polling: The processor continuously polls the receive queue for new packets. Once it finds a filled buffer pointer, it reads the data from the buffer, processes the packet, and then returns the buffer pointer to the free queue, readying it for the next incoming packet.

This process ensures an efficient and continuous flow of data from the network into the system's main memory, allowing the processor to handle packets as they arrive.

2.6.2 Packet Transmission

For packet transmission, the roles of the processor and NIC are somewhat reversed, but the underlying principles of using descriptor rings and queues remain the same.

1. Packet Descriptor Rings: As with reception, the processor's driver code maintains rings of packet descriptors in DRAM for outgoing packets.
2. Initialization with Empty Buffers: The driver also initializes these rings with pointers to empty packet buffers.
3. Processor Data Writing Process: When the processor has data to send, it takes an empty buffer pointer from the "transmit free queue", writes the data into the buffer in main memory, and then places the buffer pointer into the "transmit queue".
4. NIC Polling: The NIC continuously polls the transmit queue for buffers containing data ready to be sent. When it finds one, it transmits the packet over the network and then returns the buffer pointer to the free queue.

This mechanism ensures that data is efficiently transferred from the system's memory to the network, leveraging the NIC's capabilities to handle the actual transmission.

2.7 Performance Characterization of Baseline SoC

The SoC was implemented and validated on KC-705 FPGA. The performance of the SoC was characterized using two applications, namely the standard Ping application and the Network content cache application. Both the applications were executed in a bare-metal environment using an open-source TCP/IP stack, lwIP(lightweight IP), which was ported for the AJIT processor and custom-designed NIC.

2.7.1 Average round trip time (RTT)

To measure the average RTT, ICMP request packets were generated using the standard ping command from a Linux terminal on a host machine and sent to the SoC on the FPGA. The processor core was running at 80MHz, the NIC subsystem operated at 125MHz with a 1Gbps Ethernet link, and the lwIP TCP/IP stack generated ICMP response packets.

Table 2.1: Avg RTT time and Data rate achieved for different packet sizes

Packet size(bytes	**Average RTT (us)**
64	290
128	354
256	388
512	429
1024	449

2.7.2 Network Content Caching application

In this application, the SoC setup is deployed on the FPGA, where both the lwIP server and client are running. Two hosts, A and B, are operating on the host side (PC or laptop). Host A pre-populates the cache (DRAM) with frequently needed data via a TCP connection. When Host B requests data from Host A, Host A sends a ”forward” command to the lwIP server on the FPGA, including Host B’s IP address and port number. The lwIP server then forwards the data to Host B using a UDP connection. The data transmission speeds for populating the ”cache” on the FPGA by Host A and forwarding data from the ”cache” on the FPGA to Host B are reported below.

1. Host A(on PC) to Controller (On FPGA): 23.7 Mbps
2. Controller (On FPGA) to Host B (on PC) : 13.43 Mbps

(**Application is described in more detail in appendix B**)

2.8 Epilogue

To improve the performance of the aforementioned applications, we have explored enhancements to the memory subsystem architecture.
Two optimizations will be detailed in the following chapter

Chapter 3

Optimization of the Baseline Architecture

3.1 Introduction

In our efforts to optimize the baseline architecture, we have focused on enhancing the memory subsystem architecture. This component lies along the data plane path of packet processing and represents a crucial pathway where significant performance improvements can be achieved. We investigated two modifications to the memory subsystem architecture first architecture employs a fast local packet memory shared by both the processor and the NIC. The second architecture uses an L2 cache in the processor and NIC's access path to the main memory. Both the optimization and their impact are detailed in this chapter

3.2 Optimization A: Sharing a fast local packet memory between the NIC and the processor

The proposed architecture suggests a departure from the conventional approach of storing descriptor ring structures and packet buffers in the slower main memory (DRAM). Instead, these critical components are housed within a fast local memory that is shared between the processor and the Network Interface Card (NIC). Importantly, this shared memory region is designated as non-cacheable to ensure consistent states.

By leveraging a fast local memory, the architecture aims to significantly reduce ac-

cess latency for both the processor and the NIC. This memory is optimized for speed, making it ideal for storing time-sensitive data structures like descriptor rings and packet buffers. Placing the descriptor rings and packet buffers in a shared memory space facilitates efficient communication and data transfer between the processor and the NIC. This shared resource allows both entities to access and modify data quickly without the overhead typically associated with accessing DRAM. By reducing memory access latency and optimizing data-sharing mechanisms, the proposed architecture aims to enhance the system's overall performance and throughput. The architecture is shown below.

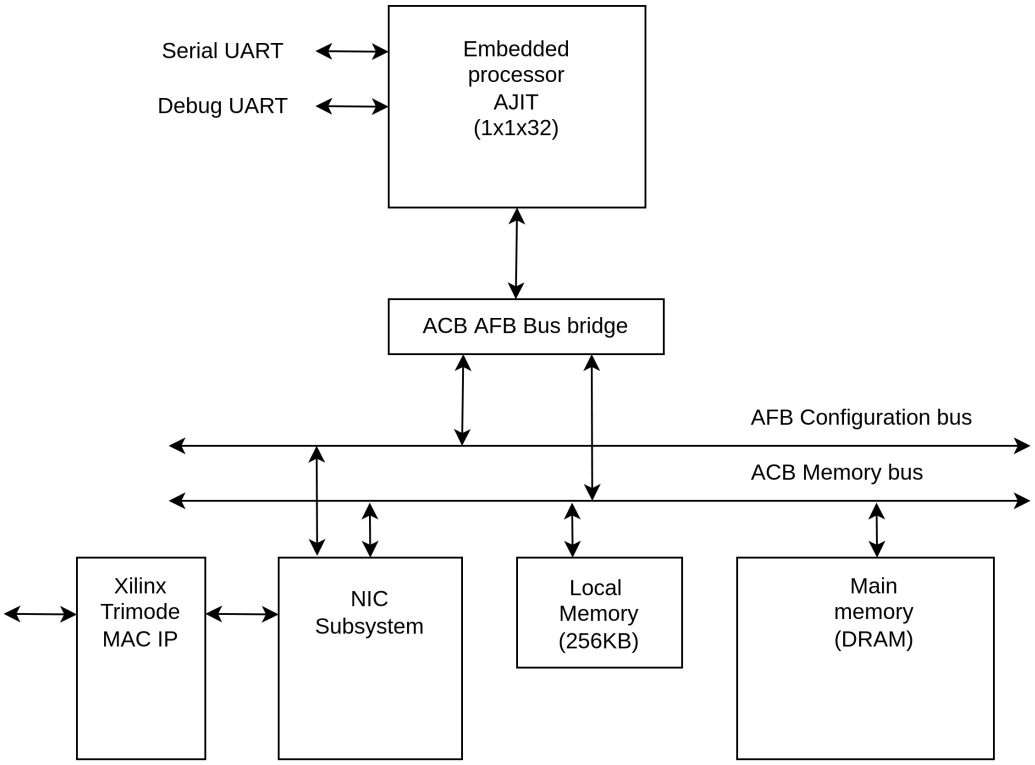

Figure 3.1: NIC and processor sharing a fast local packet memory of 256KB

3.3 Performance Characterization of Optimization A

The SoC was implemented and validated on KC-705 FPGA. The performance of the SoC was characterized using two applications, namely the standard Ping application and the Network content cache application. Both the applications were executed in a bare-metal environment using an open-source TCP/IP stack, lwIP(lightweight IP),

which was ported for the AJIT processor and custom-designed NIC.

3.3.1 Average round trip time (RTT)

To measure the average RTT, ICMP request packets were generated using the standard ping command from a Linux terminal on a host machine and sent to the SoC on the FPGA. The processor core was running at 80MHz, the NIC subsystem was operating at 125MHz with a 1Gbps Ethernet link, and the lwIP TCP/IP stack generated ICMP response packets.

Table 3.1: Avg RTT time and Data rate achieved for different packet sizes

Packet size(bytes)	**Average RTT (us)**
64	216
128	261
256	288
512	313
1024	337

3.3.2 Network Content Caching application

In this application, the SoC setup is deployed on the FPGA, where both the lwIP server and client are running. Two hosts, A and B, are operating on the host side (PC or laptop). Host A pre-populates the cache (DRAM) with frequently needed data via a TCP connection. When Host B requests data from Host A, Host A sends a ”forward” command to the lwIP server on the FPGA, including Host B’s IP address and port number. The lwIP server then forwards the data to Host B using a UDP connection. The data transmission speeds for populating the ”cache” on the FPGA by Host A and forwarding data from the ”cache” on the FPGA to Host B are reported below.

1. Host A(on PC) to Controller (On FPGA): 28.77 Mbps
2. Controller (On FPGA) to Host B (on PC) :16.22 Mbps

(**Application is described in more detail in appendix B**)

3.4 Optimization B: Using an L2 cache in NIC and the processor access path to main memory (DRAM)

The proposed architecture introduces a novel approach to accessing descriptor ring structures and packet buffers, moving away from direct access in slower main memory (DRAM). Instead, these critical components are accessed through an inclusive write-back L2 cache shared between the Network Interface Card (NIC) and the processor. This architectural choice aims to optimize data access speeds and enhance overall system performance.

By utilizing an inclusive write-back L2 cache, the architecture leverages the benefits of caching mechanisms to improve access latency for descriptor rings and packet buffers. This shared cache design ensures that frequently accessed data remains readily available to both the NIC and the processor, reducing the need for repeated accesses to the slower DRAM.

Accessing descriptor ring structures and packet buffers through the L2 cache streamlines data retrieval operations. The cache’s inclusive nature ensures that data accessed by one entity (either the NIC or the processor) is also available in the cache for subsequent accesses by the other entity. This approach minimizes latency and enhances the responsiveness of the system to incoming and outgoing network traffic. The architecture is shown below.

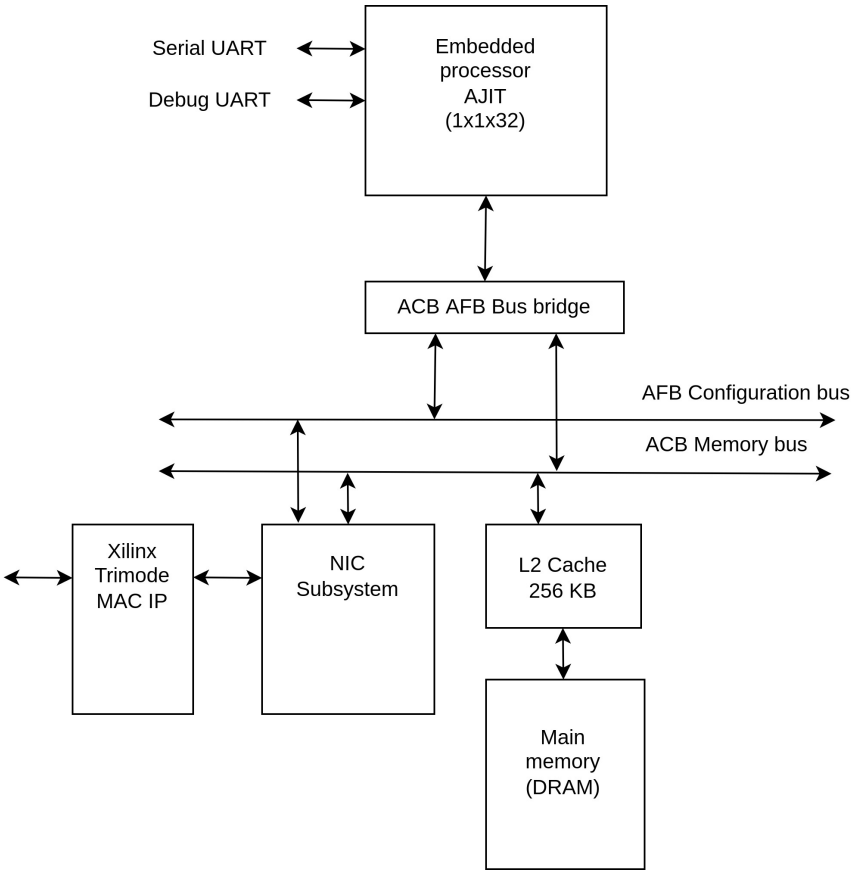

Figure 3.2: NIC and processor access main memory via L2 cache

3.5 Performance Characterization of Optimization B

The SoC was implemented and validated on KC-705 FPGA. The performance of the SoC was characterized using two applications, namely the standard Ping application and the Network content cache application. Both the applications were executed in a bare-metal environment using an open-source TCP/IP stack, lwIP(lightweight IP), which was ported for the AJIT processor and custom-designed NIC.

3.5.1 Average round trip time (RTT)

To measure the average RTT, ICMP request packets were generated using the standard ping command from a Linux terminal on a host machine and sent to the SoC on the FPGA. The processor core was running at 80MHz, the NIC subsystem was operating at 125MHz with a 1Gbps Ethernet link, and the lwIP TCP/IP stack generated ICMP response packets.

Table 3.2: Avg RTT time and Data rate achieved for different packet sizes

Packet size(bytes)	**Average RTT (us)**
64	245
128	288
256	322
512	350
1024	371

3.5.2 Network Content Caching application

In this application, the SoC setup is deployed on the FPGA, where both the lwIP server and client are running. Two hosts, A and B, are operating on the host side (PC or laptop). Host A pre-populates the cache (DRAM) with frequently needed data via a TCP connection. When Host B requests data from Host A, Host A sends a ”forward” command to the lwIP server on the FPGA, including Host B’s IP address and port number. The lwIP server then forwards the data to Host B using a UDP connection. The data transmission speeds for populating the ”cache” on the FPGA by Host A and forwarding data from the ”cache” on the FPGA to Host B are reported below.

1. Host A(on PC) to Controller (On FPGA): 27 Mbps
2. Controller (On FPGA) to Host B (on PC) : 14.22 Mbps

(**Application is described in more detail in appendix B**)

Chapter 4

Conclusion & Future Work

4.1 Conclusion

A comparative analysis of three architectures—the Baseline architecture, Optimization A where the Processor and NIC share a fast local packet memory, and Optimization B where the Processor and NIC access main memory via L2 Cache—reveals that Optimization A yields a performance gain of 34.5%, while Optimization B results in an 18.3% improvement. Therefore, the Optimization A architecture is recommended for designing the memory subsystem to ensure sufficient fast buffering for packets in the data plane.

From Table 4.1 and Figure 4.1, we can observe that to transmit an additional 512 bytes or 4096 bits, on average, 21.67 us is required. Therefore we can achieve a throughput of **378 Mbps**, with the current setup. One can also observe from the Y-intercept the software overhead of about 152 us.

Further optimization opportunities can be explored by enhancing the architecture to offload additional computational tasks to the NIC. This approach could streamline processing within the SoC, leveraging the NIC's capabilities to handle specific tasks more efficiently.

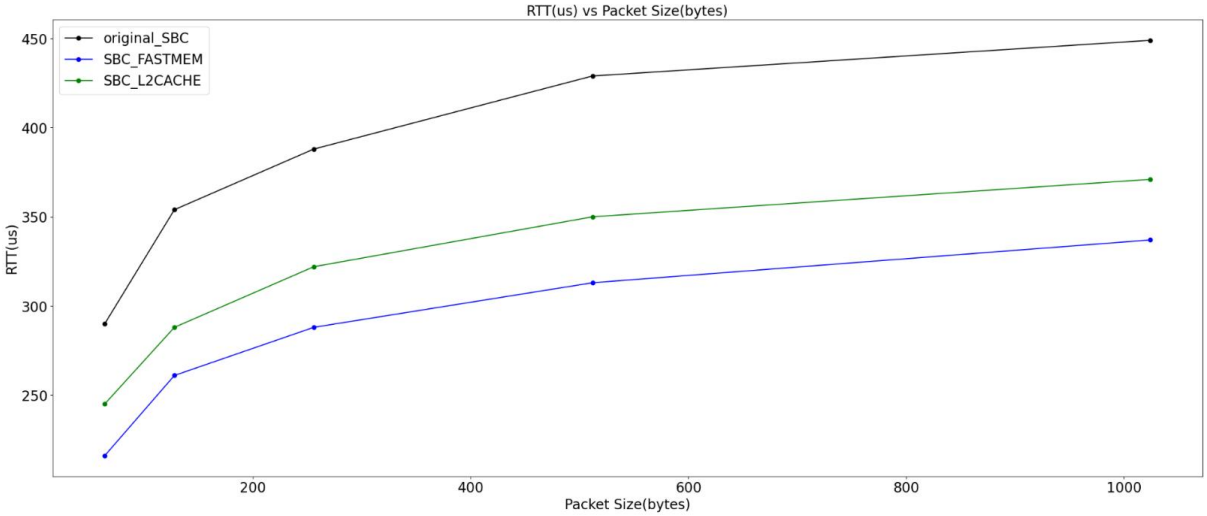

Figure 4.1: Round Trip Time vs Packet Size

Table 4.1: Avg RTT time and for different packet sizes and different architectures

Packet (bytes)	**RTT (us) Baseline**	**RTT (us) Packet memory**	**RTT (us) L2 Cache**
64	290	216	245
128	354	261	288
256	388	288	322
512	429	313	350
1024	449	337	371

Table 4.2: Uplink and Downlink speed (Mbps) for network caching application

	Data rate Baseline	**Data rate Packet memory**	**Data rate L2 Cache**
Host A To FPGA	23.7	28.77	27.12
FPGA To Host B	13.43	16.22	14.22

4.2 Future Work

1. Various computational tasks can be further optimized by offloading them to the NIC, such as TCP offloads and checksum generation.
2. In Optimization A, enhancing packet handling by directly transferring packets to main memory using DMA instead of the current ”memcpy()” function.
3. Scaling the SoC to incorporate multiple cores, allowing for separation of control plane management in one core and data plane operations in others.

.

Appendix A

Standard NIC (Network Interface Controller) Design - By MPD

This chapter focuses on the design of the Network Interface Controller (NIC) for the AJIT processor-based System-on-Chip (SoC). The NIC is crucial for providing network connectivity and communication capabilities within the SoC. The design process involves various components and considerations, such as network protocol support, data packet handling, memory management, and interfacing with the AJIT processor. This chapter aims to offer a comprehensive overview of the NIC design, detailing the key components and their interconnections. It also examines the challenges faced during the design phase and discusses the solutions and design choices implemented to address them.

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 B.1 shows all the interfaces.We examine each interface in detail.

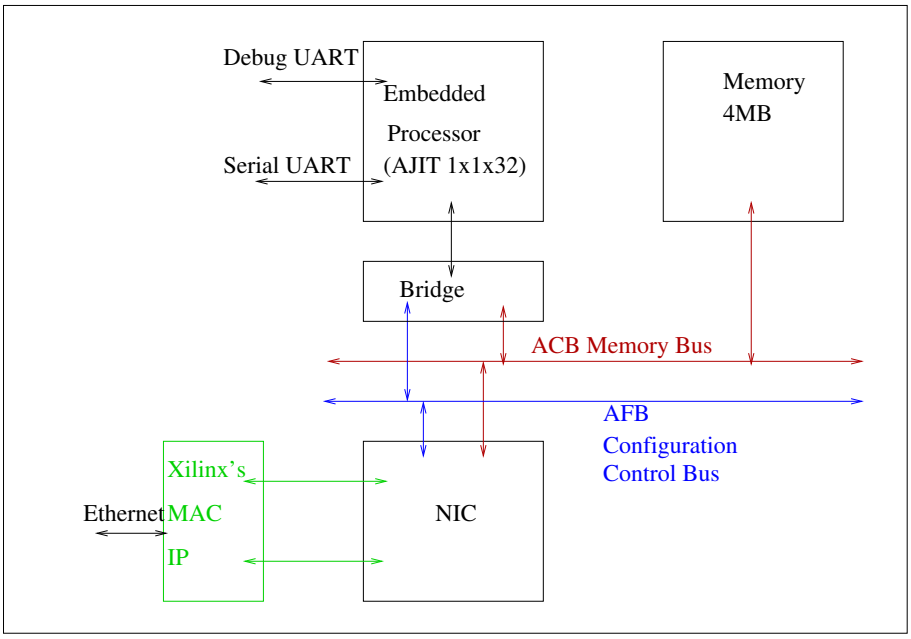

Figure A.1: Top level with Interfaces

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

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

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,

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.

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

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

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

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	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.

- **receive_free_queue**: This queue holds the addresses of free buffers that are available for storing/receiving packets. The processor initially assigns a set of buffers and pushes their physical addresses into this free queue. NIC (HW side) pops a free buffer pointer from this free queue, and writes data into the buffer, and pushes the free buffer to the receive queue. The processor (SW side) pops the buffer from the receive queue, processes the packet, and pushes the buffer pointer back to the free queue for further reception of packets.

- **transmit_free_queue**: This queue holds the addresses of free buffers that are available for transmitting the packets. The processor initially assigns a set of buffers and pushes their physical addresses into this free queue. Processor(SW side) pops a free buffer pointer from this free queue, and writes data into the buffer, and pushes the free buffer to the transmit queue. NIC (HW side) pops the buffer from the transmit queue, transmits it, and pushes the buffer pointer back to the free queue, for further transmission of packets.

- **rx_queue**: The rx_queue contents are 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 contents are 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.

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 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.

A.3.2 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.3.3 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.3.4 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.

A.4 Helper Modules

1. **doMemAccess**: All the memory accesses from NIC are guided via this module.
2. **pushIntoQueue**: This module is used to push the buffer pointer to the provided queue.
3. **popFromQueue**: This module is used to pop buffer pointer from the 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 the memory lock to '1' and then acquires the queue lock if it is available and then releases the memory lock.
5. **releaseLock**: This module releases the acquired queue lock.

Reg. ID	Address offset	Description
8	0xC0	RxQ server 0 physical address[35:32]
9	0x20	RxQ server 0 physical address[31:0]
10	0x24	RxQ server 0 lock physical address[35:32]
11	0x2C	RxQ server 0 lock physical address[31:0]
12	0x30	RxQ server 0 buffer physical address[35:32]
13	0x34	RxQ server 0 buffer physical address[31:0]
128	0x400	TxQ server 0 physical address[35:32]
129	0x404	TxQ server 0 physical address[31:0]
130	0x408	TxQ server 0 lock physical address[35:32]
131	0x40C	TxQ server 0 lock physical address[31:0]
132	0x410	TxQ server 0 buffer physical address[35:32]
133	0x414	TxQ server 0 buffer physical address[31:0]
200	0x320	Base address of free Queue Tx [35:4]
201	0x324	Base address of free Queue Tx [31:0]
202	0x328	Lock address for free Queue Tx [35:4]
203	0x32C	Lock address for free Queue Tx [31:0]
204	0x330	Buffer address for free Queue Tx [35:4]
205	0x334	Buffer address for free Queue Tx [31:0]
213	0x350	Base address of free Queue Tx [35:4]
214	0x354	Base address of free Queue Tx [31:0]
215	0x358	Lock address for free Queue Tx [35:4]
216	0x35C	Lock address for free Queue Tx [31:0]
217	0x360	Buffer address for free Queue Tx [35:4]
218	0x364	Buffer address for free Queue Tx [31:0]

Table A.6: NIC registers map

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

Appendix B

Network Content Cache Application

This section provides a detailed discussion of the network content caching application used to characterize the three architectures presented in this thesis.

B.1 Application Overview

In this application, the SoC setup is deployed on the FPGA, where both the lwIP server and client are running. Two hosts, A and B, are operating on the host side (PC or laptop). Host A pre-populates the cache (DRAM) with frequently needed data via a TCP connection (Step 1). When Host B requests data from Host A (Step 2), Host A sends a ”forward” command to the lwIP server on the FPGA (Step 3), including Host B’s IP address and port number. The lwIP server then forwards the data to Host B using a UDP connection (Step 4). The following diagram illustrates the application in more detail.

B.2 Application Details

Host A populates the cache (DRAM) on the FPGA, where lwIP server is running, via a TCP connection. Upon receiving the ”forward” command from Host A, the lwIP server forwards the data to Host B using 128-byte UDP packets. One can go for larger size UDP packets but while doing so one can observe on wireshark ”BAD UDP LENGTH >IP PAYLOAD LENGTH”, because of which python script running on host fails to receive packets.

NOTE: In this implementation, Step 2 is not explicitly integrated. It is assumed that Host B will eventually make a request, so Host A directly proceeds to Step 3, sending the command to forward the data to Host B.

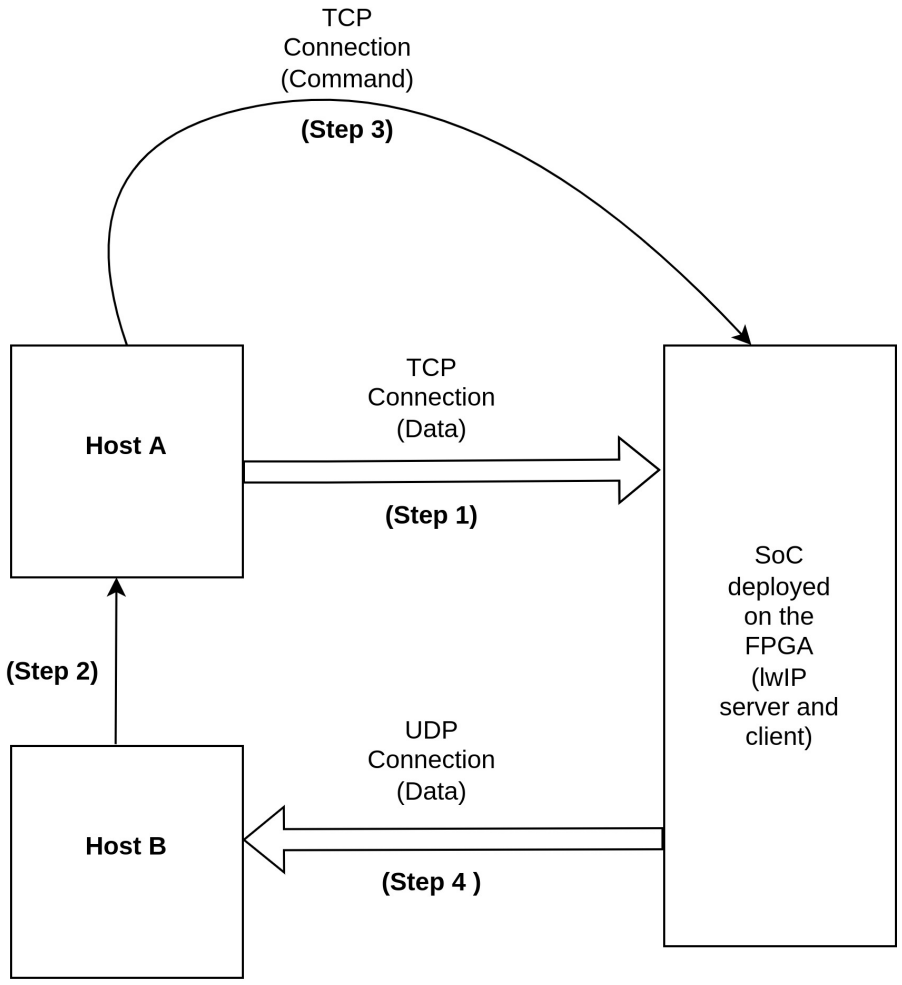

Figure B.1: Network caching application