ZeusUDP: A Near-Memory 100Gbps FPGA UDP Engine for Ultra-Low-Latency Networking

M.Subhi Abo Rdan
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, United States of America
msubhi_a@mit.edu

Mena Filfil
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, United States of America
menaf@mit.edu

***Abstract*—Network applications are increasingly critical in distributed systems, cloud services, space systems, high-frequency trading, and emerging edge-AI preprocessing workloads. Offloading the network stack from the CPU using accelerators and SmartNIC-like architectures—together with techniques such as RDMA—has become an effective strategy for achieving low latency and high throughput. In this work, we design and implement a low-latency 100 Gb/s Ethernet UDP Engine IP on an AMD UltraScale+ SoC. The engine leverages the QSFP28 interface and the hardened UltraScale+ 100 Gb/s Ethernet subsystem on the RFSoC, providing packet construction for transmission, packet filtering for received data, and a connection-management unit for bind/unbind bookkeeping. The engine is controlled by the Application Processing Unit (APU) of the Zynq UltraScale+ MPSoC through an AXI interface, and transfers packet payloads via DMA through both the non-coherent High-Performance (HP) ports and the low-latency Accelerator Coherency Port (ACP), which provides coherent access to the APU's snooping cache subsystem. This architecture enables one of the closest and lowest-latency Ethernet offload paths available to the processor.**

***Index Terms*—FPGA, Ultrascale+ SoC, AXI4, DMA, Ethernet, Low-Latency Networking, UDP, 100Gbps Ethernet**

I. Introduction

Network stack development has received significant attention in recent years. In a traditional Linux networking stack, an incoming packet must traverse multiple layers of the Linux kernel before reaching a user application. While this layered architecture provides reliability, isolation, and security, it introduces substantial overhead. Interrupt-driven packet delivery, context switching, and deep protocol-stack traversal increase latency to levels that are unacceptable for modern low-latency applications [4].

To overcome these limitations, a range of *kernel-bypass* networking techniques has emerged. Frameworks such as the Data Plane Development Kit (DPDK) allow user applications to directly access NIC queues by providing user-space, polling-mode drivers that bypass the kernel network stack. This eliminates interrupts, reduces context switching, and enables high-throughput packet processing [5]. Others, such as `PF_RING` ZC (Zero Copy), minimize memory movement between kernel and user space through zero-copy packet buffers that allow applications to access NIC memory regions directly [6]. Remote Direct Memory Access (RDMA) enables one machine's NIC to read or write the memory of another machine without involving the remote operating system or CPU in the data-transfer path [7]. These techniques form the foundation of modern high-performance Remote Procedure Call (RPC) systems—including FaRM, eRPC, and other microsecond-scale transports—yet recent studies show that RDMA-based RPCs are now increasingly CPU-bound, with performance gaps widening by up to 4$\times$ due to software overheads [8]. This highlights the importance of reducing CPU involvement to fully exploit next-generation networking hardware.

As heterogeneous and accelerated computing architectures have matured, *SmartNICs* have emerged as a powerful mechanism for offloading packet processing, storage functions, and compute kernels from the CPU. Platforms such as AMD's Alveo and NVIDIA's BlueField integrate programmable logic, embedded CPUs, high-speed transceivers, and specialized accelerators to support flexible, high-throughput, low-latency networking pipelines. A comprehensive survey of SmartNICs shows the breadth of applications they support—including storage disaggregation, virtualization, security, and distributed computing—and highlights emerging research directions toward deeper host-system integration and near-memory processing [9].

However, most SmartNICs interface with the host processor over PCIe, which remains a latency bottleneck for ultra–low-latency workloads such as high-frequency trading, spaceborne communication systems, and fine-grained serverless environments. Recent work such as Dagger shows that offloading the entire RPC stack onto a reconfigurable, FPGA-based NIC can significantly outperform PCIe-attached SmartNICs by taking advantage of Intel's CCI-P (Coherent Cache Interface – Protocol), a coherent accelerator interface that allows the NIC to directly access the Cache CPU's cache hierarchy [10]. By eliminating PCIe round-trip delays and enabling near-memory execution of RPC logic, Dagger demonstrates the effectiveness of tightly coupled, coherent NIC architectures for sub-microsecond networking.

System-on-Chip (SoC) FPGA platforms, such as AMD/Xilinx UltraScale+ MPSoCs and RFSoCs, provide a similar design opportunity. By integrating ARM CPUs, programmable logic, high-speed memory buses, and hardened networking blocks on a single die, these systems avoid PCIe entirely. They expose tightly coupled, cache-coherent interfaces—including AXI HPC and the Accelerator Coherency Port (ACP)—which enable programmable logic to access CPU memory with significantly lower latency. This architectural model parallels the near-memory NIC philosophy introduced by Dagger, but in a single-chip ARM+FPGA SoC.

In this work, we implement a full 100Gbps Ethernet UDP Engine on a Zynq UltraScale+ RFSoC platform. The design leverages a QSFP28 optical interface and the hardened UltraScale+ 100Gbps Ethernet subsystem to establish line-rate connectivity. A 512-bit AXI4-Stream datapath operating at 322.22 MHz enables high-throughput packet processing. We provide a memory-mapped control interface to the Application Processing Unit (APU) for configuring connections and managing state. The engine performs packet construction for outgoing traffic, packet filtering for incoming flows, and uses a custom connection-manager IP block based on a set-associative hash table to manage allowed flows efficiently. The UDP engine can communicates with the APU through DMA over either AXI HP or ACP ports, enabling low-latency, high-bandwidth data transfer between programmable logic and the processor.

The remainder of this paper is organized as follows: Section II-A introduces the UltraScale+ architecture with emphasis on the PS–PL AXI interfaces used for high-speed data movement. Section II-B reviews the networking stack targeted by our design. Section III presents the architecture of the UDP engine, including the Connection Manager, transmit (TX) and receive (RX) engines, control FSM, and DMA integration. Section IV discusses the implementation, experimental setup, measured latency, and resource utilization. Section V outlines future extensions, and Section VI concludes the paper.

II. Background

A. Zynq UltraScale+ PS–PL AXI Interfaces

The AMD/Xilinx UltraScale+ MPSoC and RFSoC platforms combine a quad-core Arm Cortex-A53 processing system (PS) with a highly reconfigurable programmable logic (PL) fabric, enabling tight integration between software-driven control and hardware acceleration. Communication between the PS and PL is achieved through a set of high-bandwidth AXI interfaces, each tailored for different memory-consistency and throughput requirements. Figure 1 illustrates the available datapaths.

The **High-Performance (HP)** ports provide non-coherent, high-throughput access from the PL to external DDR memory through the system memory management unit (SMMU), making them well suited for bulk data transfer and continuous streaming pipelines.

The **High-Performance Coherent (HPC)** ports optionally support I/O coherence through the Arm Cache Coherent Interconnect (CCI). When coherence is enabled, reads and writes issued through the PL can snoop the APU's L1 and L2 caches, ensuring visibility of cached data without manual flushing.

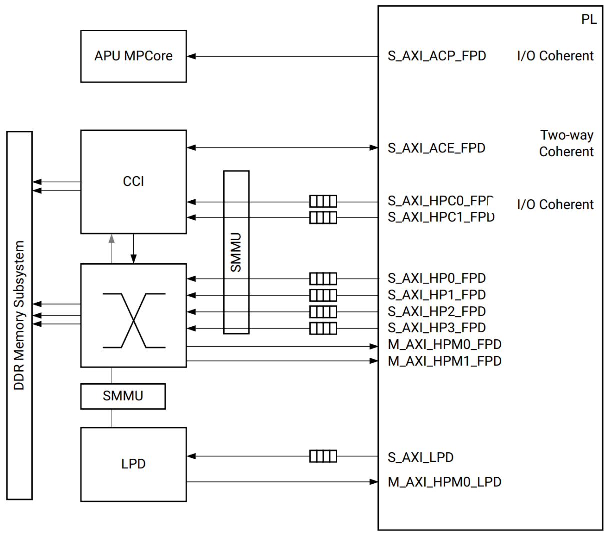

Fig. 1. PS–PL AXI interface datapaths. Diagram from [1].

The **Accelerator Coherency Port (ACP)** is a 128-bit AXI slave interface that connects the PL directly to the snoop control unit (SCU). The ACP provides a low-latency, two-way coherent path between the PL and the APU's caches. This port is most effective for medium-grain acceleration tasks that require fine-grained communication or shared-memory synchronization between the CPU and hardware accelerator [11].

In contrast, the HP ports deliver higher sustained bandwidth but lack hardware cache coherence. Software must explicitly manage cache maintenance operations (flushes and invalidations) when sharing buffers between the APU and the PL. Thus, designers often choose ACP for latency-sensitive, CPU-coupled tasks, and HP for throughput-oriented streaming workloads.

B. Network Stack

This project focuses on accelerating the network stack for high-frequency networking, specifically for packets that follow the Ethernet–IP–UDP protocol hierarchy. In classical layered networking, each protocol adds its own header information and encapsulates the payload of the layer above it. The layers relevant to this work are:

- **Link Layer (Ethernet)**: Responsible for local frame delivery within a broadcast domain. Each device is assigned a globally unique **MAC (Media Access Control)** address used by Ethernet switches for forwarding decisions.
- **Internet Layer (IPv4)**: Provides end-to-end addressing and routing across heterogeneous networks. This layer introduces logical addressing through 32-bit IPv4 addresses. Routers forward packets based on the destination IP.
- **Transport Layer (UDP)**: Provides a simple, stateless message-delivery abstraction. A UDP **socket** is identified

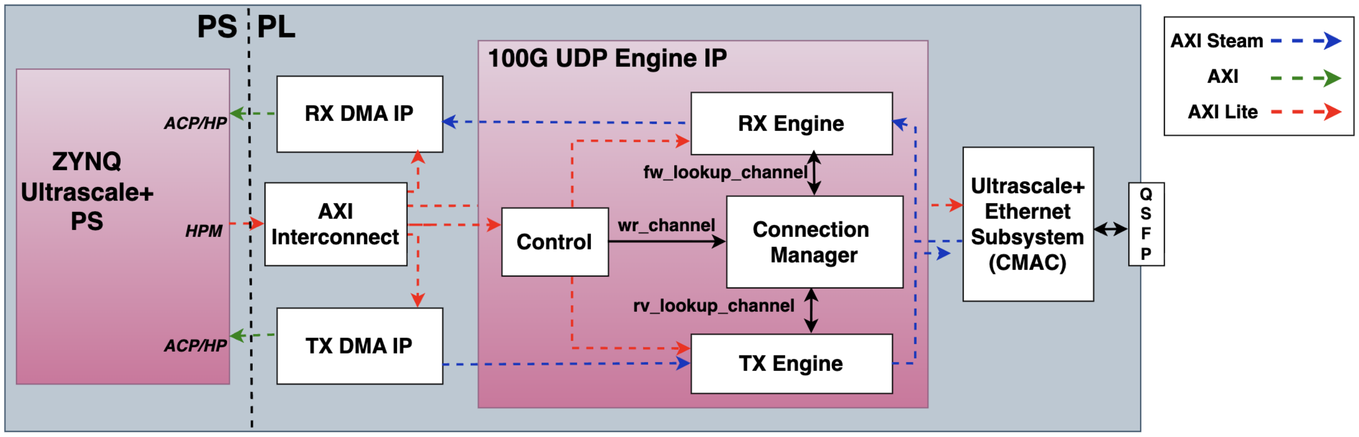

Fig. 2. UDP Engine High-Level Design

by an IP address and a port number. In a Linux application, the `bind()` system call associates a socket with a specific local endpoint. Unlike TCP, UDP performs no retransmission, congestion control, or byte-stream ordering, making it attractive for latency-sensitive applications and hardware offload engines.

Table II-B summarizes the structure of Ethernet, IPv4, and UDP frames, including key header fields and their byte lengths.

TABLE I
BREAKDOWN OF ETHERNET, IPV4, AND UDP FRAME STRUCTURE

Layer / Field	Description	Bytes
Ethernet II (14 bytes)		
Destination MAC	Target hardware address	6
Source MAC	Sender hardware address	6
EtherType (0x0800)	Indicates IP	2
IPv4 Header (20 bytes min)		
Version + IHL	IP Version + IP header length	1
DSCP + ECN	Traffic class	1
Total Length	IPv4 packet size	2
Identification	Fragment ID	2
Flags + Frag Offset	Fragmentation info	2
TTL	Hop count	1
Protocol (=17 UDP)	Transport protocol	1
Header Checksum	Header integrity	2
Source IP	Sender IPv4 address	4
Destination IP	Receiver IPv4 address	4
Options (optional)	Alignment or padding	0–40
UDP Header (8 bytes)		
Source Port	Sender port	2
Destination Port	Receiver port	2
Length	UDP header + data	2
Checksum	Optional in IPv4	2
UDP Payload	Application data	Variable

In high-performance environments, packets are typically identified by their:

$$(\text{Source IP}, \text{Destination IP}, \text{Source Port}, \text{Destination Port}),$$

which uniquely describes a flow for UDP-based communication. Many FPGA-based networking engines, including ours, use this tuple for classification, filtering, and forwarding decisions. In our design, the UDP Engine IP incorporates a dedicated **Connection Manager** unit that hashes flow tuples into compact *connection identifiers* (connectionIds). These connectionIds are used internally to accelerate lookups, enforce access control, and efficiently steer packets through the transmit and receive datapaths. On the transmit side, the connId enables fast connection lookups for packet header construction, while on the receive side it enables connection confirmation and filtering of allowed flows.

All network protocols standardize on **network byte order** (big-endian). Multibyte header fields must therefore be byte-swapped when interacting with little-endian processors or AXI-based FPGA pipelines. Additionally, IPv4 and UDP include lightweight checksums over their headers and payloads. Hardware network engines must compute, verify, or update these checksums while maintaining a fully pipelined datapath capable of sustaining line rate.

III. 100GBPS UDP ENGINE IP DESIGN

The UDP Engine IP is designed to interface with the hardened Integrated 100G Ethernet Subsystem (CMAC) on the UltraScale+ RFSoC. The CMAC is configured in CAUI-4 mode, operating over four 25.78125 Gbps/s GTY lanes on the QSFP28 interface. On the user side, the UDP Engine exposes a 512-bit AXI4-Stream datapath clocked at 322.22 MHz, enabling full 100 Gbps/s packet processing. [2]

As illustrated in Fig. 2, the UDP Engine consists of four major components: a transmit (TX) engine, a receive (RX) engine, a Connection Manager, and a control unit interfacing with the Processing System (PS) via AXI-Lite. The UDP Engine sits logically between the 100GbpsE CMAC and the PS memory system: it exchanges Ethernet frames with the CMAC on one side, and communicates with DMA engines on the other side to transfer application payloads to and from the PS over the HP and/or ACP ports.

On the receive path, incoming Ethernet frames from the CMAC are parsed, validated, and filtered. During header parsing, the RX engine extracts the source IP address and UDP source port and forwards them to the Connection Manager, which performs a forward lookup and returns a hit bit together with the corresponding connection identifier (connectionId) if the flow is registered. When both the packet format and the connection lookup are valid, the RX engine removes the Ethernet, IPv4, and UDP headers, annotates the payload with the returned connectionId, and forwards it to the RX DMA for delivery to the PS.

On the transmit path, application payloads and their associated connectionIds arrive from the TX DMA. The TX engine issues a reverse lookup to the Connection Manager using the provided connectionId, retrieving the stored destination IP address, UDP port. Using this information, the TX engine constructs the appropriate Ethernet, IPv4, and UDP headers, appends the payload, and produces a fully assembled packet, which is then delivered to the CMAC for 100 Gbps/s transmission.

The Connection Manager maintains a compact table of active connections using a set-associative hash structure. It supports both forward lookups, used by the RX engine to validate incoming flows, and reverse lookups, used by the TX engine to reconstruct packet headers from connectionIds. The control unit provides software access to configuration and management registers over the AXI-Lite interface, enabling the PS to set the FPGA's MAC/IP parameters and to bind or unbind entries in the Connection Manager.

Detailed descriptions of the Connection Manager, TX engine, RX engine, control logic, and DMA integration are presented in the following subsections.

A. Connection Manager

The Connection Manager implements a set-associative hash table used for bookkeeping of bound UDP connections. It exposes three logical interfaces: a forward-lookup channel, a reverse-lookup channel, and a serialized write channel used for bind and unbind operations. These channels operate independently and may run in different clock domains, allowing the control domain (PS) to run at a lower frequency (e.g., 100 MHz) while the lookup datapath channels operate at the 322.22 MHz CMAC rate. See diagram 3:

Forward lookup: The RX engine provides the source IP address and UDP source port to the forward-lookup channel. These fields are hashed into a 16-bit index using an XOR-fold hash function derived from the 32-bit IP address and 16-bit UDP port. The hash index selects a set in the table, which contains four ways in our implementation. Each way stores a {tag, valid} pair, where the tag corresponds to the concatenated {IP address, UDP port}. All ways are checked in parallel. If a matching valid entry is found, the Connection Manager returns a *hit* signal and a *connectionId*, which is uniquely defined by {way, index}. The forward-lookup logic is fully pipelined and can accept a new request every cycle.

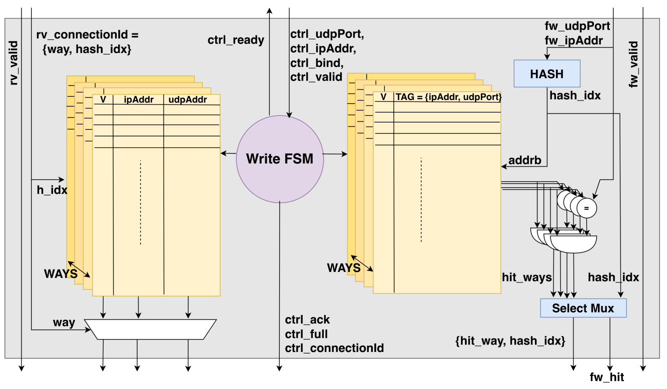

Fig. 3. Connection Manager Block Diagram

Reverse lookup: The TX engine provides a connectionId to the reverse-lookup channel. Because the connectionId encodes both the hash index and way, the lookup is direct: the corresponding table entry is read and returned without hashing or tag comparison. Like the forward-lookup path, this channel is deeply pipelined and supports one request per cycle.

Write channel (bind/unbind): Connection updates are performed through a serialized finite-state machine (FSM), ensuring that only one control command is in flight at a time. Bind operations proceed as follows:

1) Compute the hash index from {IP, UDP port}.
2) Read all ways at that index.
3) If a matching tag already exists, return the existing connectionId (`ack=1, full=0`).
4) Else, if an invalid way is available, allocate it and return a new connectionId.
5) Else, if all ways are valid, report the set as full (`ack=1, full=1`).

Unbind operations clear any entry within the set whose tag matches the provided {IP, UDP port}.

The lookup channels (forward and reverse) do not stall during bind or unbind operations; however, during an update, they may temporarily observe stale entries. New entries become visible once the write sequence completes. This behavior preserves full datapath throughput without imposing backpressure on packet processing.

Memory organization: Because each BRAM supports only two read/write ports, and the Connection Manager must sustain one request per cycle on both lookup channels while allowing occasional writes, the table is replicated across multiple BRAM banks. This avoids structural hazards and ensures concurrent forward lookup, reverse lookup, and control writes. A more area-optimized implementation could multiplex these accesses into a single BRAM, but doing so would restrict lookups during updates and degrade datapath availability.

B. TX Engine

The TX Engine is responsible for assembling outgoing UDP/IP packets and delivering them to the CMAC in the proper 100 GbE frame format. It operates at 322.22 MHz and

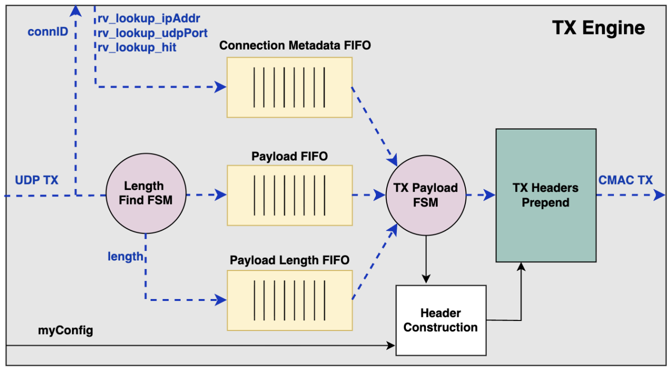

Fig. 4. TX Engine Block Diagram

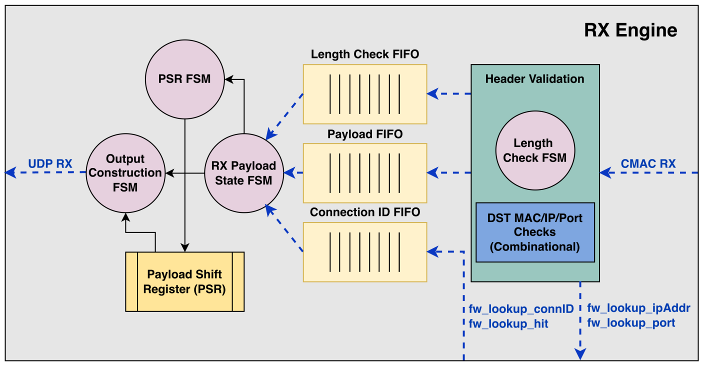

Fig. 5. RX Engine Block Diagram

receives application payloads over a 512-bit AXI4-Stream interface, with each packet annotated by a connectionId that identifies the destination flow. See diagram 4.

When a packet begins arriving, the TX Engine immediately initiates three parallel processes. First, the payload beats are streamed into a payload FIFO. As they arrive, the engine computes the payload length on-the-fly by counting the valid bytes in each beat until the `tlast` signal is observed. When `tlast` is received, the final payload length is pushed into a length-metadata FIFO.

Second, the connectionId from the first beat is forwarded to the Connection Manager, which performs a reverse lookup to retrieve the destination IP address and UDP port. This information is stored in a connection-metadata FIFO.

Together, the payload FIFO, length-metadata FIFO, and connection-metadata FIFO contain all information required to construct the Ethernet, IPv4, and UDP headers. A packet-construction FSM on the output side of the FIFOs waits until all metadata for a given packet is available, ensuring that (i) the complete payload is buffered, (ii) its length is known, and (iii) the required connection data has been resolved.

Once all metadata is ready, the FSM retrieves the destination connection information—specifically the destination IP address and destination UDP port—from the connection-metadata FIFO, and combines it with the locally configured network parameters, including the FPGA's source MAC address, source IP address, source UDP port, and destination MAC address (set by the Control FSM described later in Section III-D). Using these fields, the TX Engine constructs the complete Ethernet, IPv4, and UDP headers. A header-prepend FSM then emits the generated headers followed by the payload beats from the payload FIFO, producing a fully assembled packet that is forwarded to the CMAC for transmission.

The use of FIFOs is essential, as the engine must buffer a variable-length payload while concurrently resolving flow metadata and computing the payload length. This design allows the TX Engine to operate entirely on-the-fly, sustaining a throughput of one beat per cycle while maintaining proper alignment between header construction and payload emission. FIFO depths are provisioned to support payloads up to 4 KB.

C. RX Engine

Our RX engine decodes all incoming data beats from the CMAC RX port, buffers packet payloads, and performs full header and length validation. Packets whose payloads are corrupted during transmission are automatically discarded. The first stage of the RX pipeline validates the header by parsing the source and destination MAC addresses, IP address, and UDP ports, and comparing them against the NIC's configured MAC address, IP address, and UDP port (set by the **Control FSM** discussed later).

During the first AXI-S beat of each packet, a counter is initialized to track the packet length as subsequent beats arrive. The expected packet length from the header is captured at the same time. The length-check FSM writes an entry to the **Length Check FIFO** for every packet with a valid header; the FIFO data indicates whether the observed payload length matches the expected length. In parallel, the raw payload data is written to the **Payload FIFO**, and a forward lookup is issued to the **Connection Manager** to obtain a connection ID and determine whether the source IP address and UDP port are authorized. The Connection Manager response, which arrives after approximately six cycles, is written into the **Connection ID FIFO**. These FIFOs decouple the various pipeline stages, which resolve at different times. FIFO depths are provisioned to support packets up to 4 KB.

Downstream of the FIFOs, three FSMs manage the packet-processing flow:

- **RX Payload State FSM**: Maintains one of three states:
  - `IDLE` — waiting for the start of a packet,
  - `TRANSFER` — processing a validated payload until the final beat,
  - `SKIP` — entered when the source IP or UDP port is invalid, or when payload beats are dropped during transmission.
- **Payload Shift Register FSM**: Clears and stores partial payload transactions, based on the RX Payload State and FIFO availability.
- **Output Construction FSM**: Consumes the payload shift register state, the RX Payload State, and FIFO output

status to enable back-to-back handling of single-beat UDP packets with zero added latency at the FIFO outputs.

D. Control FSM

The UDP Engine exposes a set of AXI-Lite configuration registers used to control the FPGA's local network parameters, destination addressing, and connection-manager bind/unbind operations. These registers are written by the PS through an AXI-Lite slave interface implemented in the PL by a dedicated control FSM operating at 100 MHz. Each AXI-Lite write updates an internal memory-mapped register, and these registers drive configuration signals throughout the engine, including the TX and RX pipelines and the Connection Manager.

Table II summarizes the AXI-Lite register map implemented by the control FSM.

TABLE II
AXI-LITE CONFIGURATION REGISTER MAP

Offset	Name	Description
0x00	`CTRL`	Control bits (enable, bypass, loopback)
0x04	`MY_MAC_HIGH`	Local MAC address [47:32]
0x08	`MY_MAC_LOW`	Local MAC address [31:0]
0x0C	`MY_IP`	Local IPv4 address
0x10	`MY_PORT`	Local UDP source port
0x14	`DST_MAC_HIGH`	Destination MAC address [47:32]
0x18	`DST_MAC_LOW`	Destination MAC address [31:0]
0x1C	`CONN_IP`	Connection Manager: IP for bind/unbind
0x20	`CONN_PORT`	Connection Manager: UDP port for bind/unbind
0x24	`CONN_BIND`	Bind flag (1 = bind, 0 = unbind)
0x28	`CONN_TRIGGER`	Trigger to start bind/unbind FSM
0x2C	`CONN_STATUS`	Status bits: `ack[0]`, `full[1]`
0x30	`CONN_ID`	Returned connectionId after bind/unbind

The `CTRL` register contains several control bits used to manage datapath behavior: bit 0 enables the TX engine, bit 1 enables the RX engine, bit 2 enables the RX bypass mode, and bit 3 enables internal loopback. These control bits, along with the FPGA's network-configuration registers (MAC, IP, and port), cross from the 100 MHz AXI-Lite domain into the 322.22 MHz TX and RX clock domains through CDC synchronizers. Multi-bit configuration updates assume that the engines are idle during register modifications.

The Connection Manager's write channel also operates in the 100 MHz clock domain, so no CDC is required. A bind or unbind operation proceeds by programming `CONN_IP`, `CONN_PORT`, and `CONN_BIND`, followed by a write to `CONN_TRIGGER`, which launches the serialized bind/unbind FSM inside the Connection Manager. Once the operation completes, software reads `CONN_STATUS` to check the `ack` and `full` bits, and then reads `CONN_ID` to obtain the assigned connection identifier.

E. DMA

Data transfer between the UDP Engine IP and the PS is performed using Xilinx DMA IP blocks connected to the PS High-Performance (HP) ports. Both the TX and RX DMA instances are configured identically with the following parameters:

- Write buffer length register: 26 bits
- Address width: 32 bits
- Memory-mapped data width: 128 bits (maximum supported by HP PS ports)
- Stream data width: 128 bits (maximum supported HP PS ports)

The RX DMA is used in write mode to store received packets into PS DRAM. The TX DMA is used in read mode to retrieve data written by the PS user logic for transmission.

Because the internal payload path of the UDP engine is 512 bits wide, an AXI-Stream Data Width Converter IP is included to down-convert each 512-bit payload beat into four consecutive 128-bit beats compatible with the DMA stream interface.

To address this throughput mismatch more effectively, a future enhancement could incorporate a payload splitter and four parallel DMA engines, each operating on a 128-bit stream, thereby sustaining the full 512-bit datapath rate (see Section V).

IV. IMPLEMENTATION

A. Setup

The design operates on three primary clock domains:

- `s_axi_clk` (100 MHz): Used for the AXI-Lite control interface, configuration registers, and the Connection Manager write channel. This clock is generated by the PS.
- `rx_axis_clk` (322.22 MHz): Drives the RX datapath, including the CMAC RX AXI-Stream interface and the UDP Engine RX logic (forward-lookup channel). This clock is derived from the CMAC GTY reference clock.
- `tx_axis_clk` (322.22 MHz): Drives the TX datapath, including the CMAC TX AXI-Stream interface and the UDP Engine TX logic (reverse-lookup channel). This clock is also generated by the CMAC from the GTY reference.

Two AXI High-Performance (HP) ports on the PS (HP0 and HP1) are used for the RX and TX DMA engines described in Section III-E, while an AXI High-Performance Master (HPM) interface from the PS is used to drive the AXI-Lite control interface.

For validation, the TX and RX paths were tested in loopback using two methods: (1) the internal loopback mode of the CMAC IP, and (2) an external QSFP28 loopback module connected to the RFSoC board. Both approaches produced consistent functional behavior and timing results.

In addition, the RX engine includes an optional internal router simulation feature controlled through the configuration registers listed in Table II. When enabled, the TX engine rewires the source and destination MAC addresses, IP address, and UDP ports so that transmitted packets re-enter the RX datapath but the destination and source information is flipped across all 3 layers, allowing end-to-end behavior to be emulated on a single RFSoC the need for two systems to communicate with each other.

B. Latency

The TX engine, RX engine, and the forward (FW) and reverse (RV) lookup channels of the Connection Manager all operate in fully pipelined mode, capable of accepting a new request every cycle. In contrast, the Connection Manager write channel is sequential, processing one bind or unbind operation at a time. The primary latency values are summarized in Table III.

The forward-lookup path of the Connection Manager has a latency of 7 cycles: one cycle for hashing, five cycles for the BRAM access, and one cycle for selecting the matching way among the associative entries. The BRAM latency is relatively high because the table operates at 322.22 MHz, which requires deeper pipelining for timing closure.

The reverse lookup requires only the BRAM access, resulting in a 5-cycle latency, since no hashing or tag matching is needed—the input connectionId directly specifies the index and way.

The write channel latency depends on whether the operation is a bind or an unbind. For an *unbind*, the FSM requires five cycles to read the BRAM entry and one additional cycle to assert the write-enable signal to clear the valid bit. For a *bind*, the FSM performs the same five-cycle BRAM read, one cycle to check for an already existing entry (optional exit), and an additional number of cycles equal to the number of ways to search for a free slot.

The TX engine latency is dominated by the reverse lookup latency plus the time needed to buffer and prepend headers to the payload. In the worst case, the TX latency is:

$$\begin{aligned}\text{Latency} = {} & \max(\text{RV lookup, payload buffering}) \\ & + 2 \text{ cycles (FIFO)} + 1 \text{ cycle (header prepend).}\end{aligned} \tag{1}$$

which accounts for the extra cycle required if header and payload alignment necessitate an additional beat.

The RX engine latency is determined by the larger of the forward-lookup latency in the Connection Manager or the time required to pipeline the incoming payload through the FIFO. In the worst case, the RX latency is:

$$\begin{aligned}\text{Latency} = {} & \max(\text{FW lookup, payload pipeline}) \\ & + 1 \text{ cycle (length-check FSM)} \\ & + 1 \text{ cycle (shift-register fill)} \\ & + 2 \text{ cycles (output pipelining).}\end{aligned} \tag{2}$$

The shift-register term represents the additional cycle required for multi-beat packets before the first valid output word is available. The final two output-pipeline stages were introduced to suppress empty AXI-Stream beats and to allow timing closure at 322.22 MHz while still correctly handling wide `tkeep` patterns.

For more information on the DMA IP latency refer to the official AXI DMA IP performance tables in the product guide. [3]

TABLE III
LATENCY SUMMARY OF MAJOR PIPELINE COMPONENTS

Path	Clock	Latency (cycles)
CM Forward Lookup	322.22 MHz	7
CM Reverse Lookup	322.22 MHz	5
CM Control Write (Bind)	100 MHz	$6 + \text{WAYS}$
CM Control Write (Unbind)	100 MHz	6
UDP RX Engine	322.22 MHz	$\max(7,\ L_{\text{pkt}}+1) + 4$
UDP TX Engine	322.22 MHz	$\max(5,\ L_{\text{payload}}) + 3$
DMA IP (S2MM - Write Mode)	322.22 MHz	39
DMA IP (MM2S - Read Mode)	322.22 MHz	34

C. Utilization

Table IV summarizes the FPGA resource utilization of the UDP Engine on the Zynq UltraScale+ RFSoC 4x2 device.

TABLE IV
FPGA RESOURCE UTILIZATION

Resource	Used	Available	Utilization
LUT	29,725	425,280	6.99%
LUTRAM	3,035	213,600	1.42%
FF	62,526	850,560	7.35%
BRAM	753	1,080	69.72%
URAM	32	80	40.00%
GT	4	16	25.00%

The dominant contributor to BRAM usage is the Connection Manager. The hash function generates a 16-bit index, corresponding to 2^{16} entries per way; across four associative ways, and with duplicated memories required for concurrent forward and reverse lookups, the total storage requirement approaches 3.1 MB. This footprint accounts for both tag storage and valid-bit structures.

To reduce routing pressure caused by BRAM over-utilization, all FIFOs in the datapath were implemented using UltraRAM (URAM). This significantly improved timing closure, especially on the 322.22 MHz RX and TX pipelines.

Further reductions in memory utilization are possible. Options include narrowing the hash width (thereby reducing the table depth), reducing the maximum number of supported concurrent IP/UDP-port bindings, or adopting a more compact hashing scheme that preserves uniform distribution while requiring fewer address bits.

V. FUTURE WORK AND EXTENSIONS

While the current system provides a complete and fully functional 100 Gb/s UDP engine, several architectural enhancements can further increase sustained packet rate, reduce CPU overhead, and improve integration with higher-level applications.

1) ACP Port Integration. The current design supports the Accelerator Coherency Port (ACP) as a drop-in alternative to the High-Performance (HP) ports. As described in Section II-A, the ACP provides cache-coherent access to the APU's L2 cache hierarchy, which can significantly reduce CPU polling latency for control-critical or low-latency

UDP flows. A useful future extension is to classify traffic into *latency-sensitive* and *bulk* categories, routing the former through the ACP and the latter through HP-based DMA paths. This selective routing enables hybrid datapaths in which high-priority flows bypass DDR entirely and interact directly with CPU-resident buffers. Additional considerations and configuration details for ACP-based integration are provided in the Appendix.

2) Multi-DMA Architecture for Full 512-bit Throughput. As described in Section III-E, each HP and ACP interface is limited to a 128-bit AXI data width, while the internal UDP datapath is 512 bits wide. This mismatch constrains peak DMA throughput. A scalable solution is to integrate a 1-to-4 packet arbiter that distributes packets across four independent DMA engines, each mapped to a separate HP port. The arbiter FSM tracks packet boundaries, schedules transfers to available DMA channels, and updates a status register that the CPU can poll to determine which buffer contains the next complete packet. This design restores full throughput while enabling parallel CPU consumption of packet streams.

Although the PS exposes only one ACP port, it may still be dedicated to a specialized DMA path serving high-priority or privileged UDP flows—for example, connections requiring near-real-time CPU interaction or extremely low jitter. This hybrid HP/ACP multi-DMA topology is a promising direction for applications with heterogeneous latency requirements.

3) Scatter-Gather DMA with Ring Buffers. The current implementation uses simple DMA transfers initiated by software, requiring CPU intervention for each packet movement. To support continuous high-rate operation, a Scatter-Gather DMA engine with programmable descriptors may be introduced. Descriptors would index into circular *ring buffers* for both RX and TX paths, enabling continuous packet reception without per-packet CPU triggers. This improvement keeps the CPU almost entirely out of the data-movement loop and transitions the design toward a high-performance SmartNIC-style architecture.

4) Checksum Generation and Validation. For simplicity, checksum computation and verification were omitted in the current prototype. These functions can be added as lightweight, deeply pipelined stages that operate on header bytes. As long as the checksum pipeline depth remains below the BRAM latency of the Connection Manager, it introduces no additional throughput penalty and integrates seamlessly into the RX and TX engines.

VI. Conclusion

This work presented the design and implementation of a full 100 Gb/s UDP engine with integrated connection bookkeeping on an UltraScale+ RFSoC platform. We described the complete datapath architecture and demonstrated how the engine interfaces with the Processing System (PS) through both non-coherent High-Performance (HP) ports and the low-latency, cache-coherent Accelerator Coherency Port (ACP).

The resulting design achieves minimal latency while operating at a high clock frequency, making it well suited for edge-AI preprocessing, serverless cloud services, distributed systems, and high-frequency trading workloads. Additionally, we outlined several future architectural extensions—such as multi-DMA scaling, ACP prioritization, and scatter-gather integration—that can further enhance performance and enable deeper hardware–software co-design on a single SoC device.

Overall, this work demonstrates that tightly integrated FPGA-based UDP acceleration can provide a robust and extensible foundation for next-generation low-latency networking applications.

References

[1] AMD (Xilinx), "Zynq UltraScale+ Device Technical Reference Manual, UG1085 (v2.5)," March 21, 2025. [Online]. Available: https://docs.amd.com/v/u/en-US/ug1085-zynq-ultrascale-trm

[2] AMD (Xilinx), "UltraScale+ Devices — Integrated 100G Ethernet Subsystem, LogiCORE IP Product Guide PG203 v3.1," Jul. 2024. [Online]. Available: https://docs.amd.com/r/en-US/pg203-cmac-usplus (accessed Dec. 10, 2025)

[3] AMD (Xilinx), "AXI DMA LogiCORE IP Product Guide, PG021 v7.1," 2022. [Online]. Available: https://docs.amd.com/r/en-US/pg021_axi_dma

[4] J. C. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," *ACM Trans. Comput. Syst.*, vol. 15, no. 3, pp. 217–252, 1997.

[5] Intel Corporation, "Data Plane Development Kit (DPDK): Programmer's Guide," Intel, 2014. [Online]. Available: https://www.dpdk.org/

[6] L. Deri, "PF_RING ZC (Zero Copy): High-speed packet capture, filtering and forwarding," ntop.org, 2013. [Online]. Available: https://www.ntop.org/products/packet-capture/pf_ring/

[7] S. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2014, pp. 295–306.

[8] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, "No compromises: Distributed transactions with consistency, availability, and performance," in *Proc. 25th ACM Symp. Operating Systems Principles (SOSP)*, 2015, pp. 54–70.

[9] E. Kfoury, S. Choueiri, A. Mazloum, A. AlSabeh, J. Gomez, and J. Crichigno, "A comprehensive survey on SmartNICs: Architectures, development models, applications, and research directions," *IEEE Commun. Surveys Tuts.*, vol. 26, no. 1, pp. 230–271, 2024.

[10] L. Firestone, D. Zhu, K. Ni, S. Kottapalli, and M. Silberstein, "Dagger: Fast and efficient RPCs in cloud microservices with near-memory reconfigurable NICs," in *Proc. 28th ACM Symp. Operating Systems Principles (SOSP)*, 2021, pp. 700–715.

[11] M. Sadri, C. Weis, N. Wehn, and L. Benini, "Energy and performance exploration of the Accelerator Coherency Port using Xilinx Zynq," in *Proc. IEEE/ACM Intl. Symp. Low Power Electronics and Design (ISLPED)*, 2013, pp. 343–348.

VII. Appendix

A. Challenges

1) Timing Closure: Running the UDP engine at a high clock frequency (322.22 MHz) introduced several timing challenges. After verifying the RX and TX engines in simulation, we identified the main sources of timing violations:

- **High BRAM usage:** Large BRAM utilization caused significant combinational routing delays due to connections across multiple banks. This was mitigated by increasing the BRAM latency from 2 to 5 cycles, effectively breaking long routing paths. Additionally, implementing all FIFOs using UltraRAM reduced routing pressure, as URAM primitives allowed contiguous memory blocks without introducing excessive delays.
- **Excessive combinational logic:** Early RX engine implementations included LUT-intensive `tkeep` bitcount logic, which created long combinational chains. We optimized this by counting only the upper 22 bits of packets on the first transaction, which is the only case where variable beat `tkeep` from the CMAC affects whether the shift register buffers incoming data or outputs the full payload immediately. This reduced the worst timing violation from -2.1 ns to approximately -0.43 ns.
- **Costly hash function:** The initial IP/UDP hash used a multiplication for uniform distribution, which was timing-intensive. We replaced it with a simpler XOR-based hash (top bits XOR bottom bits), sufficient for our partially filled address space. This optimization further reduced the worst negative slack (WNS) to about -0.06 ns.

2) Accelerator Coherency Port Configuration: Early in the project, we focused on configuring the Accelerator Coherency Port (ACP) to ensure proper operation. We successfully configured the port to function as a write-mode DMA, transferring data from the PL to the PS. We also verified that the ACP interface could be used from a notebook in the same manner as a standard DMA interface.

The configuration required for correct operation included:

- **AXI Cache attributes:** `AxCACHE[1]` (Cacheable) asserted; optional `AxCACHE[2]` / `AxCACHE[3]` for read/write allocate hints.
- **AXI User bit:** Shareability bit asserted to indicate inner-shareable domain.
- **AXI Protection attributes:** `AxPROT` set for non-secure, privileged access as appropriate.
- **DMA direction:** Configured as write-only from PL to PS.

This configuration enables us to use the ACP port as a drop in replacement for the HP port across the RX datapath.

B. Important Links

- Project GitHub Repository