ECE 427 : TCP Offload Engine

Jack Streck
jstreck2@illinois.edu
UIUC

Vivek Reddy
vivekkr2@illinois.edu
UIUC

Haruto Iguchi
higuchi2@illinois.edu
UIUC

Jack Walberer
johnaw4@illinois.edu
UIUC

Ian Huang
ch134@illinois.edu
UIUC

Roy Liu
tzuchil2@illinois.edu
UIUC

Abstract

In real-time embedded systems deterministic and low latency communication is critical. In these constrained systems, processing the TCP/IP stack in software on a host CPU introduces significant processing overhead and jitter due to OS scheduling and interrupt handling. This project proposes the design of a custom TCP/IP Offload Engine (TOE) to aggressively minimize network latency. By bypassing the host software stack, the dedicated hardware accelerates header parsing, connection management, and payload extraction directly from the GMII data stream. The design aims to eliminate CPU interrupt latency, providing deterministic, microsecond-level response times for critical network packets.

1 Background and motivation

The host CPU overhead of TCP processing depends strongly on where the network stack is implemented. With the conventional Linux TCP stack, approximately 12.13k CPU cycles are required per TCP request. By comparison, FlexTOE[3], a hardware-based TCP offload engine, reduces this cost to 1.67k cycles per request. It suggests that moving TCP processing into hardware can greatly reduce host-side protocol overhead by 86%, thereby improving the suitability of the system for latency-sensitive workloads.

2 Project Idea

This project implements a TCP Offload Engine (TOE) designed for low-latency embedded systems operating over 1 Gb/s Ethernet. The TOE behaves as a lightweight NIC that offloads TCP/IP processing from the host processor. The system utilizes a fully pipelined transmit and receive datapath while minimizing dependence on host-side software processing. Instead of relying on a full software stack, the host provides packet descriptors and payload data directly to the hardware. The TOE is responsible for TCP connection setup and teardown, header generation, checksum and CRC computations, and retransmission and acknowledgment handling. To support constrained on-chip 32KB SRAM, the design has a transmit window, limiting the total amount of unacknowledged payload data stored in SRAM. This prevents the host from issuing descriptors that exceed available buffering resources. This will require lightweight driver modification to have a host system utilize our chip properly.

3 Related Work

SmartNICs. SmartNICs extend traditional network interface cards by integrating programmable processing units such as embedded CPUs or FPGAs. They enable flexible packet processing and allow offloading of higher-level services beyond transport-layer functionality. However, this programmability comes at the cost of increased latency and reduced energy efficiency compared to dedicated ASIC designs, limiting their suitability for highly deterministic, low-latency systems.

Data Processing Units (DPUs). DPUs further generalize SmartNIC architectures by combining programmable cores, hardware accelerators, and high-speed networking interfaces into a single system-on-chip. They are designed to offload full infrastructure workloads, including networking, storage, and security. Despite their flexibility, DPUs rely heavily on software execution, making them less efficient in terms of latency and power compared to specialized hardware offload engines.

Lightweight TCP/IP Stacks. Software-based lightweight TCP/IP stacks such as **lwIP** and **picoTCP**, are designed for resource constrained embedded systems. They reduce memory footprint and computational overhead compared to full operating system stacks. However, because they execute on general-purpose processors, they still incur per-packet processing overhead and timing variability, limiting their ability to achieve deterministic low-latency performance.

4 Design

To balance high-performance networking with stringent die-size constraints, we propose a streamlined TOE architecture optimized for 1 Gb/s line rates. The high-level internal architecture is depicted in Figure 1, while the external system-level interconnections are detailed in Figure 2.

`Transmission Control Block (TCB)`. As shown in Figure 3, the TCB maintains per-connection state required data transfer, which contains sequence and acknowledgment numbers, transmit window size, and connection status. In this design, the TCB is stored in on-chip SRAM and is accessed by both the transmit and receive engines to track progress of each connection.

`Retransmission Timeout (RTO)`. As shown in Figure 3, the RTO logic monitors outstanding (unacknowledged) segments using timers associated with the TCB. If an acknowledgment is not received within the timeout interval, the corresponding segment is retransmitted from the retransmission buffer, ensuring reliability in the presence of packet loss. In addition on the RX pathway there is a Reorder Block which houses packets that are received out of order, until the next corresponding in order packet is received.

`Address Resolution Protocol (ARP)`. The ARP cache stores mappings between IP addresses and corresponding MAC addresses

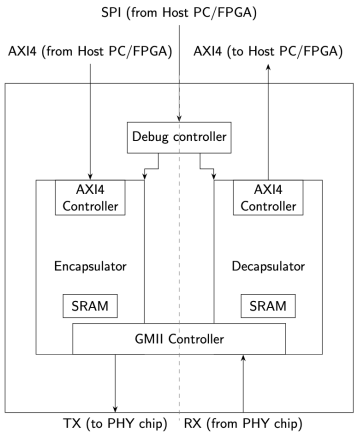

Figure 1: Chip Architecture Diagram

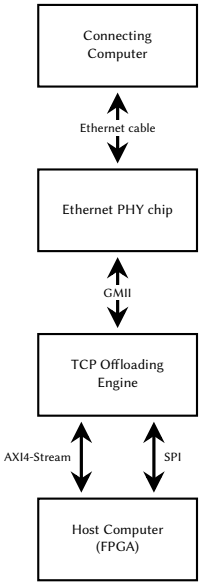

Figure 2: System Architecture Diagram

to enable efficient Ethernet frame transmission. When the transmit engine needs to send a packet, it queries the ARP cache to determine the destination MAC address for the given IP address. If a mapping is not present, the ARP FSM issues an ARP request and temporarily stalls transmission until a response is received. Once resolved, the mapping is inserted into the cache for future use, reducing latency for subsequent transmissions and avoiding repeated ARP broadcasts.

`Encapsulator(TX Engine)`. As shown in the appendix, the Encapsulator processes outbound data starting from the AXI4-Stream interface, where it receives raw application payload from a host PC (or a FPGA in our design). The dataflow proceeds as follows:

- **TCP Segment Generation:** The engine fetches the destination IP and Port from AXI4-Stream interface. It generates the TCP header, calculating the sequence numbers and managing the window size.
- **IP and MAC Framing:** The TCP segment is encapsulated into an IP packet. Concurrently, the ARP FSM resolves the destination MAC address to form the Ethernet MAC frame.
- **Checksum Engine:** Features a high-speed, pipelined adder-tree architecture to perform 16-bit one's complement TCP/IP checksum calculations on-the-fly. This pipelined design ensures that checksum accumulation does not become a bottleneck, maintaining full throughput even during high-bandwidth transmission.
- **CRC-32 Module:** Implements the IEEE 802.3 CRC-32 algorithm for Ethernet frame validation using the standard polynomial (0x04C11DB7). To support the 8-bit GMII data path, the module computes the CRC incrementally, processing one byte per cycle using a parallelized combinational update function derived from the CRC polynomial. This design effectively unrolls the LFSR-based computation into a combinational XOR network. The CRC register is initialized to all ones and inverted at the end of frame transmission to produce the final Frame Check Sequence (FCS).

`Decapsulator(RX Engine)`. As shown in the appendix, the Decapsulator handles inbound traffic from the Ethernet PHY, stripping headers layer-by-layer to recover the payload:

- **Ethernet Parsing:** The incoming bitstream is synchronized, and the Ethernet Parser validates the Frame Check Sequence (FCS) using the CRC-32 checker.
- **Header Validation:** The IP and TCP headers are parsed to verify the destination address and port. The TCP engine validates the checksum to ensure data integrity during transit.
- **Acknowledgment Handling:** Upon successful validation, the Decapsulator triggers the generation of an ACK signal to the Encapsulator's retransmission logic to clear the corresponding buffer entry.

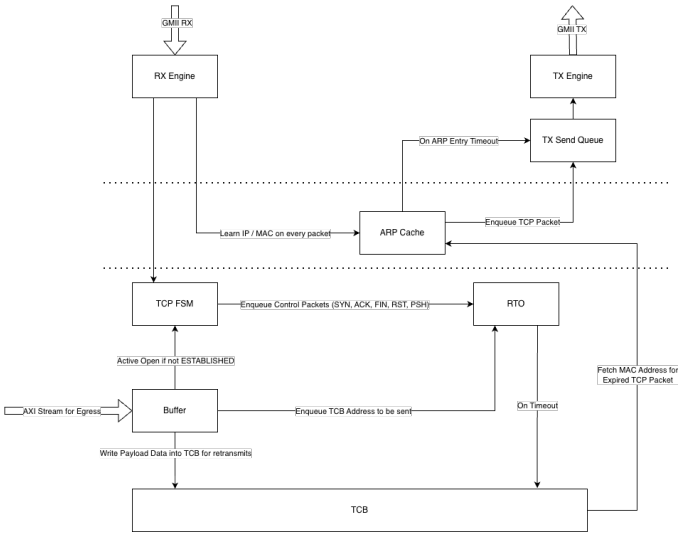

Figure 3: Chip Overview Diagram

- **Payload Delivery:** Once the headers are stripped (decapsulated), the raw data are streamed to the host FPGA via the AXI4-Stream interface for application processing. To minimize latency, payload data streams to the host FPGA in parallel to CRC and Checksum. If either fails, we can tag the payload with an error flag signals the host to discard the invalid frame.

`Three-way Handshake Mechanism`. The TCP connection establishment follows a synchronized three-way handshake protocol, managed by the interaction between the TX and RX Finite State Machines (FSMs):

(1) **SYN Phase (Active Open):** As shown in Figure 4, the process begins when the `TX FSM` transitions from the `CLOSED` state. Upon detecting payload flow from the host, the FSM enters the `SYN_SENT` state and transmits a `SYN` packet to the remote peer.
(2) **SYN-ACK Phase (Passive Open):** As shown in Figure 5, on the receiving end, the `RX FSM` starts in a `LISTEN` state (triggered by a passive open command from the host). When a `SYN` packet is received, the FSM transitions to `SYN-RECEIVED` and responds by sending a `SYN-ACK` packet.
(3) **ACK Phase (Completion):** Once the `TX FSM` receives the `SYN-ACK` while in the `SYN_SENT` state, it sends a final `ACK` and transitions to `ESTABLISHED`. Simultaneously, when the `RX FSM` receives this final `ACK`, it also enters the `ESTABLISHED` state. At this point, both machines are ready to exchange full-duplex data.
(4) **Error Handling and Reset Logic:** Both FSMs include robust error recovery paths. If a `SYN` timeout occurs during the handshake, the system sends a `RST` (Reset) signal to return to the `CLOSED` or `LISTEN` states. Furthermore, the receipt of a `RST` packet at any stage triggers a `TCB flush`.

4.1 I/O Budget

- **4 pins - General Pins:**
  - VDD, VSS, CLK, RST (1-bit each)
- **4 pins - SPI Pins:**
  - MOSI, MISO, CSS, SCLK (1-bit each)
- **22 pins - GMII Pins:**
  - TX_CLK_GMII (125 MHz)
  - TX_EN, TX_ER (1-bit each)
  - TXD[7:0] (8-bit data)
  - RX_CLK_GMII (125 MHz)
  - RX_DV, RX_ER (1-bit each)
  - RXD[7:0] (8-bit data)
- **10 pins - AXI4-Stream Receiver:**
  - TDATA[7:0] (8-bit data)
  - TVALID, TREADY (1-bit each)
- **10 pins - AXI4-Stream Transmitter:**
  - TDATA[7:0] (8-bit data)
  - TVALID, TREADY (1-bit each)

4.2 Design for Test

Scan operations are accessed through the SPI interface, which communicates with an internal test controller. SPI commands configure control registers such as chain selection, scan enable, and capture. Once scan mode is enabled, incoming SPI data is routed to the selected scan chain, and captured data is shifted out through the same interface. This approach provides full internal visibility and control without requiring additional scan pins. An SPI-controlled

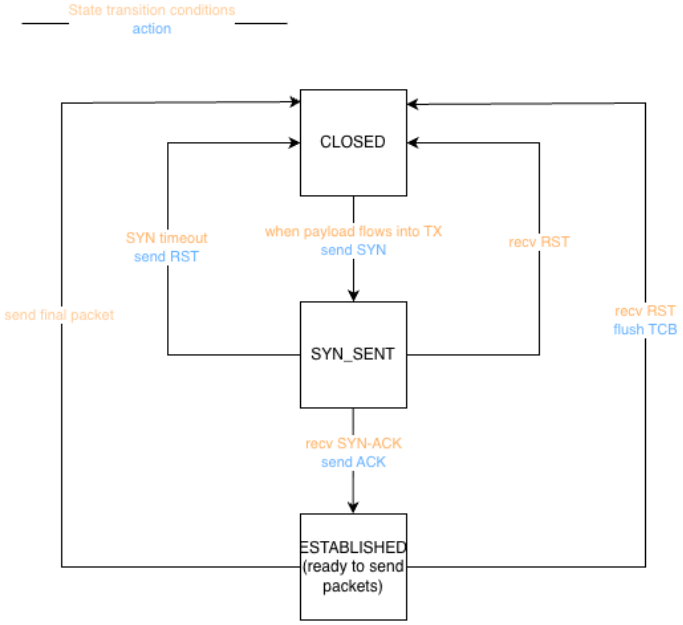

Figure 4: Encapsulator FSM Diagram

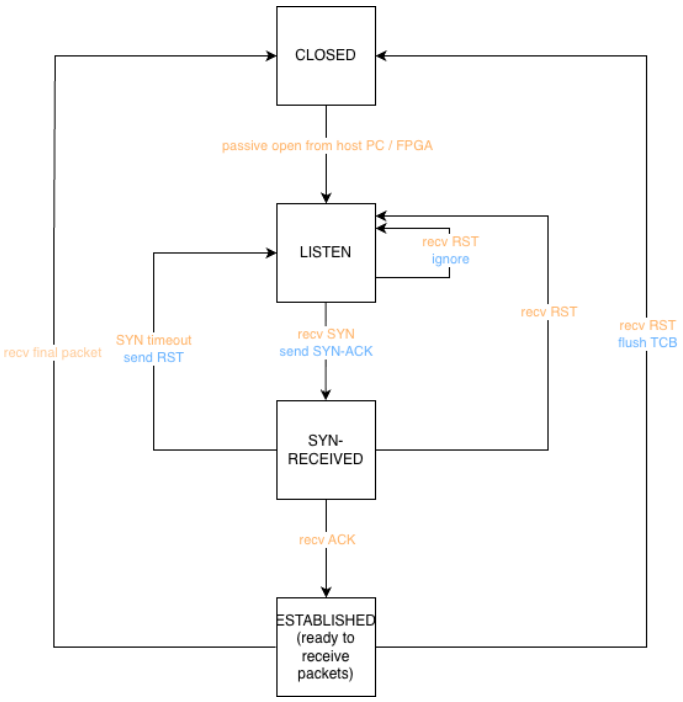

Figure 5: Decapsulator FSM Diagram

test controller will manage scan enable, scan shift, scan capture, and chain selection. In test mode, the controller selects one scan region, shifts in a test vector, applies a capture pulse, and shifts the resulting internal state back out for observation. There will be a select mux for scan chains in the encapsulator, decapsulator, AXI, and GMII. When test mode is enabled, one of these chains will be selected, and inputs can be sent in to registers or read off, depending on scan in or scan out. Each of the modes will have multiple chains. For the encapsulator, we will scan chain the TCP header, checksum generator, and retransmit. For the decapsulator, we can scan chain the ethernet parser, TCP parser, and IP checksum. Additional scan coverage will be applied to AXI handshake and buffering control logic as well as GMII transmit and receive control logic.

4.3 Verification

We will use a SystemVerilog-based testbench with DPI-C integration to connect the RTL TOE to a software reference model. The testbench will include packet generators, protocol checkers, and scoreboards to validate correctness of transmitted and received packets against expected TCP/IP behavior.
A C-based reference model will be used via DPI-C to generate expected outputs and validate correctness of header fields, sequence numbers, checksums, and retransmission behavior.
We will also implement directed test to verify functionality to test each module individually such as TCP Connection FSM, Encapsulator, Decapasulator, CRC-32, and other submodules.

4.4 Advanced features

The following features can be integrated into the baseline architecture to enhance TCP/IP throughput, improve network reliability, and further offload the networking stack entirely into hardware. These enhancements are proposed as a phased implementation roadmap.
- UDP: A lightweight, stateless data transmission protocol path for low-latency, unacknowledged applications
- AIMD Congestion Control: Hardware-driven dynamic bandwidth throttling using Additive Increase/Multiplicative Decrease algorithms to prevent network flooding
- SACK: Advanced retransmission logic to selectively re-fetch only dropped packets from the SRAM buffer, optimizing bandwidth utilization
- MSS Negotiation: Dynamic Maximum Segment Size parsing and configuration during the initial TCP three-way handshake
- DHCP: Automated state-machine logic for dynamic IP allocation, subnet masking, and gateway configuration
- ICMP: Hardware-level packet handling for ICMP Echo (Ping) requests, network diagnostics, and error reporting
- PTP: Nanosecond-precision hardware clock synchronization for highly deterministic distributed systems
- AES Encryption: Wire-speed cryptographic data security

5 Schedule

Specification and Architecture — May to June
Finalize project requirements, define the top-level architecture, assign module ownership, and establish the verification plan.

RTL Development — June to August
Implement the major datapath and control modules, including packet parsing, TCP state handling, buffering, and transmit logic.

Integration and Verification — August to September
Integrate the RTL blocks with a C-based golden model via DPI-C to verify RTL outputs and protocol correctness. Run random tests to validate retransmission and handle out-of-order packets.

Physical Design Preparation — September to October
Prepare the design for backend flow by stabilizing RTL, reviewing synthesis results, and refining timing and area constraints.

Physical Design and Signoff — October to November
Complete floorplanning, placement, routing, timing closure, DRC/LVS checks, and final signoff tasks.

Tapeout — November
Submit the final design for fabrication and finalize the documentation for the implementation and verification flow.

6 Bringup Plan

(1) Configure the chip through SPI with IP, MAC address, etc.
(2) Transmit packets from FPGA through chip to connect to Ethernet PHY chip
(3) Use a secondary device to attempt to receive/transmit packets from/to the TOE
(4) Format a scan chain configuration using SPI to test granular units depending on overall chip functionality
(5) Verify that packets are sent and received using FPGA or wireshark on the secondary device

7 Division of Tasks/Group Strength

7.1 Division of Tasks

- **RTL Design** - Haruto Iguchi, Jack Streck, Vivek Reddy, Roy Liu
- **Verification** - Haruto Iguchi, Ian Huang, Roy Liu, Vivek Reddy
- **Physical Design** - Jack Walberer, Ian Huang, Roy Liu, Jack Streck, Vivek Reddy
- **Physical Verification/Sign-off** - Jack Walberer, Ian Huang, Jack Streck
- **Software Driver Modification** - Haruto Iguchi, Roy Liu, Vivek Reddy
- **Bring-up** - Jack Streck, Vivek Reddy, Haruto Iguchi, Roy Liu

7.2 Group Strength

- **Jack Streck** – MEng, ECE
  - *Relevant Coursework:* ECE 411, ECE 425, ECE 437, ECE 482, ECE 483, ECE 511
  - *Experience:* PLC and GUI development, U55C FPGA Board Bringup, Former ECE411 CA
  - *Technical Strengths:* RTL Design, Physical Design, Verification, SPI and JTAG Debugging

- **Vivek Reddy** – Junior, ECE
  - *Relevant Coursework:* ECE 385, ECE 411, ECE 425, ECE 391
  - *Experience:* Intel DSA Accelerator Inter-Process communication research w/ Professor Kim, CPU Microarchitecture research w/ Professor Wang and Professor Kim
  - *Technical Strengths:* RTL Design, Linux Driver development, Physical Design
- **Haruto Iguchi** – Junior, ECE
  - *Relevant Coursework:* ECE 385, ECE 411, ECE 425, ECE 391
  - *Experience:* TCP Offloading Research w/ Professor Kim, Incoming DV Intern at Apple Japan
  - *Technical Strengths:* RTL Design, Verification
- **Jack Walberer** – MEng, ECE
  - *Relevant Coursework:* ECE 425, ECE 482, ECE 483, ECE 444
  - *Experience:* UIUC EE 25, Technical consulting internship-debugging/ failure analysis on pcbs. Team based project experience in cadence in 482/483.
  - *Technical Strengths:* Physical Design, Layout
- **Ian Huang** – MEng, ECE
  - *Relevant Coursework:* ECE 411, ECE 425, ECE 511, ECE 527
  - *Experience:* Incoming CPU DV Intern at Google; NPU Research w/ Professor Huang; prior ASIC design internship focused on systolic arrays; prior FPGA design internship focused on Ethernet-based systems
  - *Technical Strengths:* System arch design, RTL Design, Physical Design, Verification
- **Roy Liu** – Junior, ECE
  - *Relevant Coursework:* ECE 385, ECE 411, ECE 425, ECE 391
  - *Experience:* NPU Research w/ Professor Huang, NYCU HW/SW NPU Design
  - *Technical Strengths:* RTL Design, Physical Design, Verification

References

[1] Hankook Jang, Sang-Hwa Chung, Dong Kyue Kim, and Yun-Sung Lee. 2011. An Efficient Architecture for a TCP Offload Engine Based on Hardware/Software Co-design. *Journal of Information Science and Engineering* 27 (2011), 493–509.

[2] Jeffrey C. Mogul. 2003. TCP Offload is a Dumb Idea Whose Time Has Come. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*. USENIX Association. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1240966

[3] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. 2022. FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, 87–102.

[4] S. Shishkin et al. 2011. High Performance TCP Offload Engine. In *Proceedings of the IEEE*. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5757731

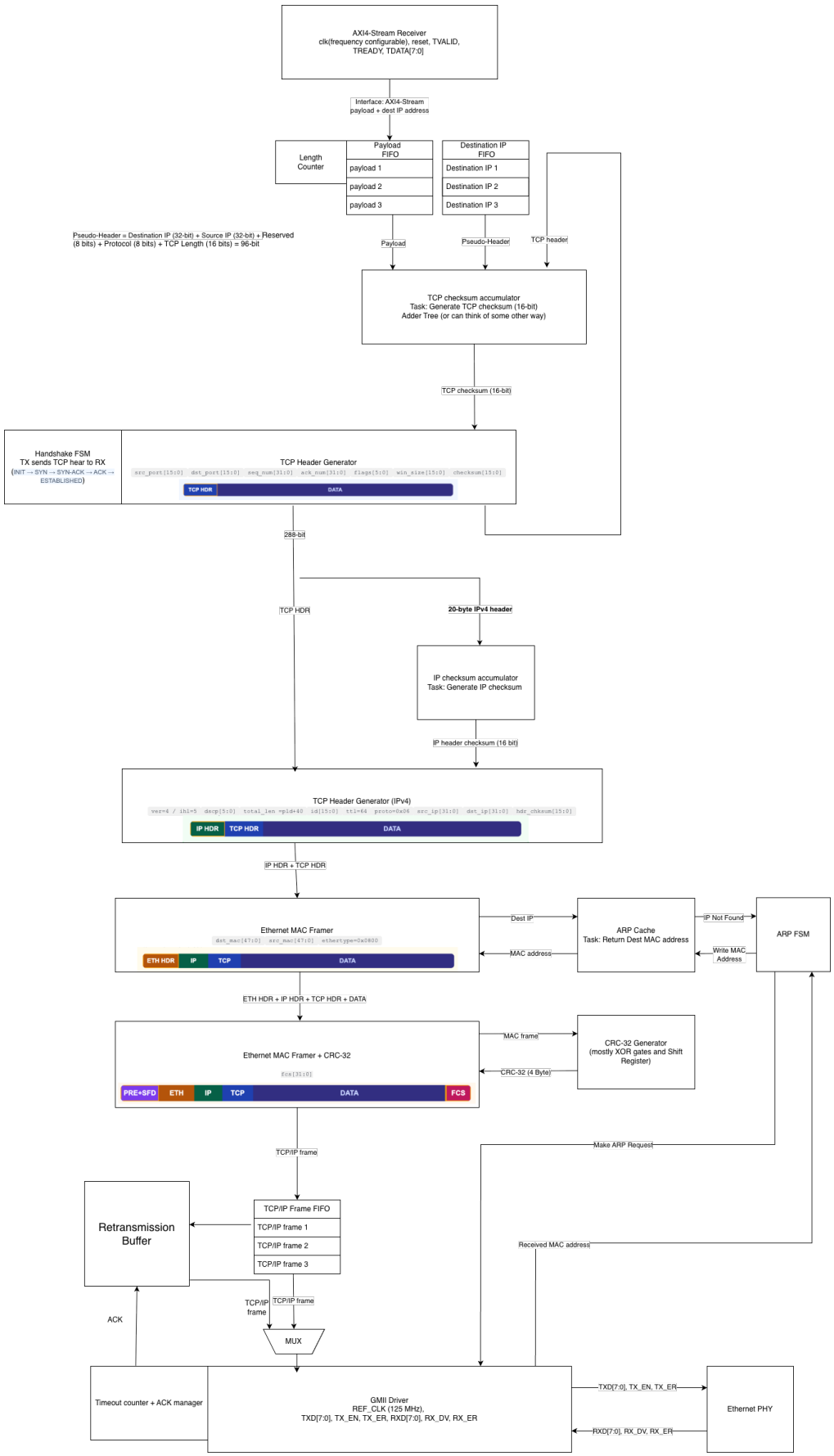

Figure 6: Detailed Architecture of the Encapsulator (TX Engine)

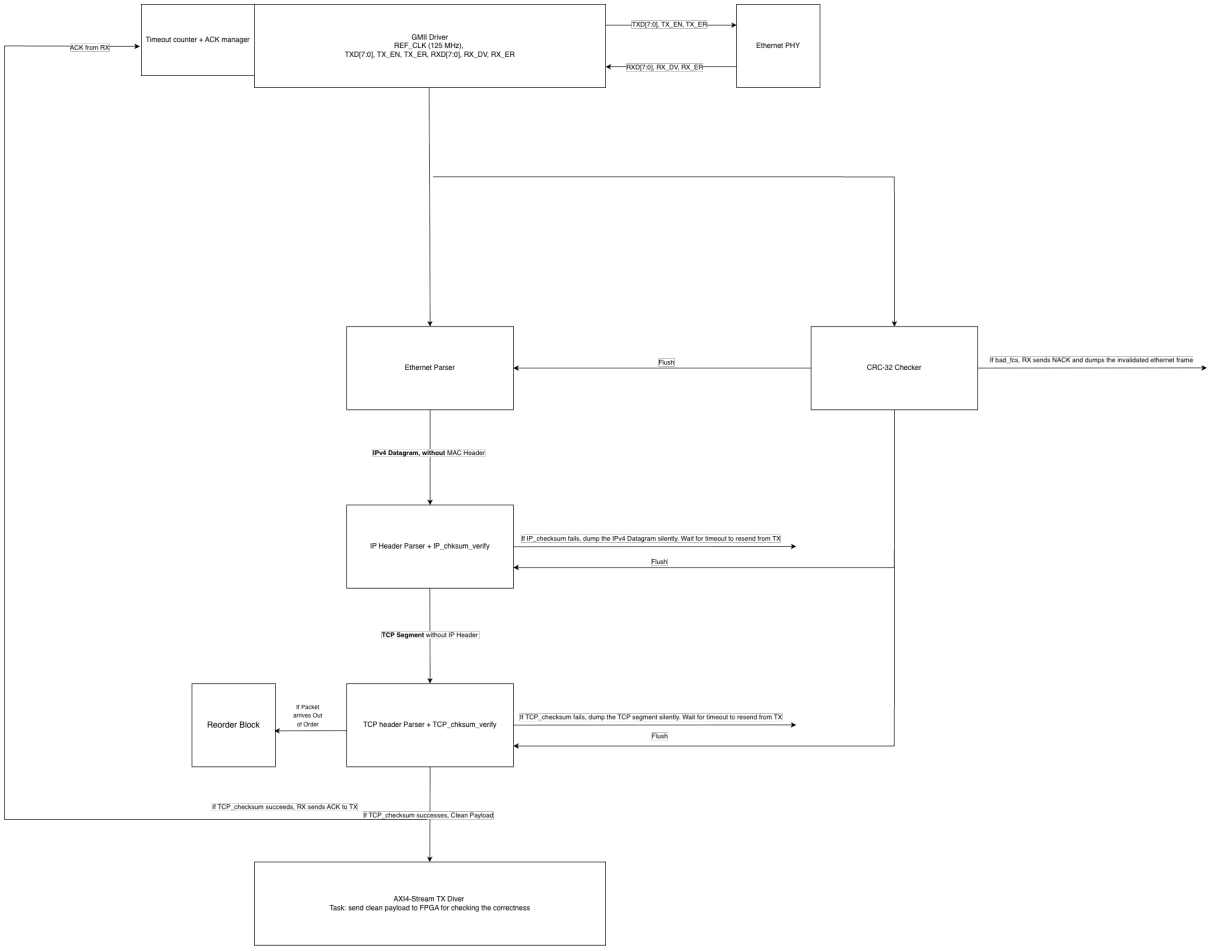

Figure 7: Detailed Architecture of the Decapsulator (RX Engine)