

Hardware-Accelerated BPF: Real-Time Traffic Monitoring and Filtering at the Edge

1st Nour Massri

MIT EECS

mnmassri@mit.edu

2nd Lucas Bautista

MIT EECS

lucas_b@mit.edu

Abstract—This report presents a design for a programmable FPGA-based network accelerator that implements a Berkeley Packet Filter (BPF) bytecode interpreter directly in hardware. Unlike traditional static hardware filters that require re-synthesis for logic changes, this system provides a flexible abstraction layer, allowing users to dynamically upload standard BPF programs to control packet filtering logic. The system monitors bidirectional network traffic between a protected machine and an external network, maintaining real-time packet statistics while selectively filtering incoming traffic. By offloading this logic to dedicated hardware, the design achieves wire-speed packet processing and deterministic latency while retaining the ease of development, safety guarantees, and programmability of standard software-defined BPF bytecode.

Index Terms—Digital Systems, Field-Programmable Gate Arrays, Packet Filter

I. INTRODUCTION

The **Berkeley Packet Filter (BPF)** is a widely adopted mechanism for efficient network packet filtering, supported by most modern operating system kernels. BPF provides a virtual machine architecture with a limited instruction set that executes user-defined filtering programs in kernel space. The BPF virtual machine features simple scratch registers, arithmetic and logic operations, and read-only access to packet data. Critically, BPF programs are stateless, executing independently on each received packet. The key advantage of BPF lies in its ability to filter packets at an early stage in the network stack, before copying packet data to user space. This early filtering eliminates unnecessary memory copies and context switches, significantly improving throughput and reducing CPU overhead for applications that inspect network traffic, such as Wireshark, tcpdump, and intrusion detection systems. BPF's design balances flexibility with safety. Users can upload arbitrary BPF bytecode to customize filtering behavior without modifying kernel code. Safety is ensured through compile-time restrictions: BPF instructions are limited to simple loads, stores, jumps, and ALU operations; backward jumps are prohibited to prevent infinite loops; and all programs are guaranteed to terminate.

Hardware-Based Packet Filtering offers significant performance benefits over software execution, yet traditional approaches suffer from rigidity. Traditional hardware packet filtering requires hardcoding logic directly into the FPGA fabric, a rigid process necessitating hardware redesigns for every

rule change. In contrast, our BPF implementation provides a **flexible programmable abstraction**. Instead of complex hardware description languages, we utilize a standard software interface to offload logic to the FPGA. By implementing a BPF bytecode interpreter directly in FPGA hardware, we offload packet filtering from the host CPU entirely while maintaining the flexibility of software.

This **BPF on FPGA** implementation provides several advantages: (1) deterministic wire-speed processing without CPU intervention, (2) parallel evaluation of multiple packets simultaneously, and (3) reduced system load on the protected machine. Our design maintains compatibility with standard BPF bytecode, allowing existing filtering programs to execute without modification. Because this bytecode is standard BPF, programs can be developed and verified within the host OS kernel and seamlessly deployed to the FPGA, ensuring easy integration and maximum flexibility.

A widely-used **application** for hardware-based packet filtering is UDP multicast feed pruning. In this architecture, servers broadcast information covering a wide range of topics or subscription groups via UDP, requiring the client to ingest the entire stream and filter for relevant data locally. To accelerate this process and prevent the exhaustion of host resources, this filtering task is frequently offloaded to an FPGA, ensuring that unwanted packets never reach the host system. This approach is particularly critical in financial markets, where exchanges broadcast dense market data streams. As shown in Figure 1, the filtering logic is defined using simple, standard bytecode. By offloading this logic, irrelevant securities are dropped at line rate before consuming PCIe bandwidth or host memory.

```
Filter: Allow Nasdaq ITCH for Stock ID 4821
000: ldh [12]                # Load Ethertype
001: jeq #0x800, 2, 8        # IPv4? -> 002, else Drop
002: ldb [23]                # Load Protocol
003: jeq #0x11, 4, 8         # UDP? -> 004, else Drop
004: ldx 4*([14]&0xf)         # Load IP Hdr Len to Reg X
005: ldh [x + 25]            # Load ID @ offset (X+25)
006: jeq #0x12D5, 7, 8       # ID==4821? -> Keep, else Drop
007: ret #1                  # Return 1 (Accept Packet)
008: ret #0                  # Return 0 (Drop Packet)
```

Fig. 1. BPF Assembly for filtering Nasdaq ITCH Market Data. This demonstrates the abstraction level: simple loads, arithmetic, and jumps replace complex hardware logic gates.

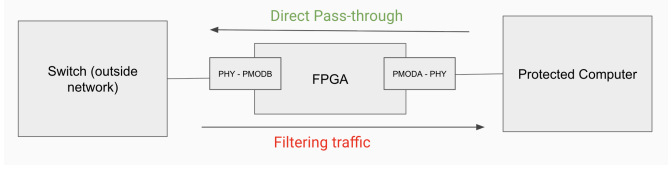


Fig. 2. Physical Setup

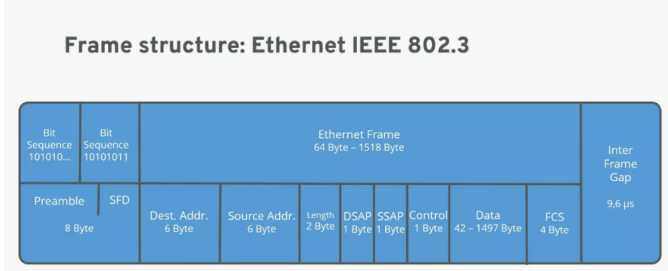


Fig. 3. Ethernet Frame Structure

II. PHYSICAL SETUP

As illustrated in Figure 2, the system is implemented on an AMD/Xilinx Urbana FPGA, interfaced with two LAN8720 PHY transceivers via PMOD expansion ports. One transceiver connects directly to the protected machine, while the second connects to an external switch, simulating the upstream network.

The architecture employs an asymmetric traffic handling model. Outbound traffic (egress from the protected machine) passes through the system transparently without inspection. Conversely, inbound traffic (ingress from the external network) is subject to active BPF filtering. Consequently, only packets validated by the active BPF program are forwarded, while non-compliant traffic is dropped in hardware to secure the internal network.

III. HIGH LEVEL DESIGN

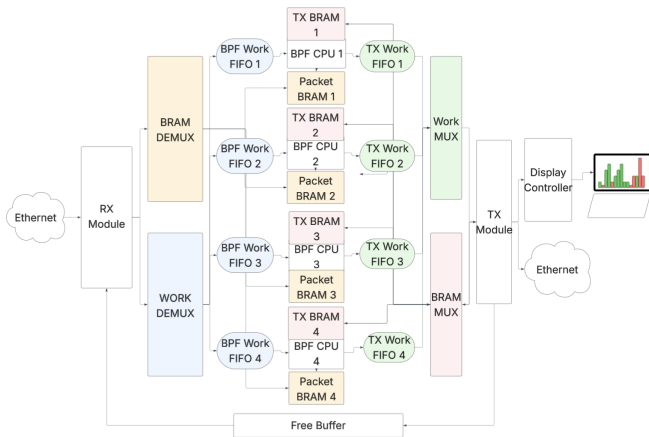


Fig. 4. High Level Diagram

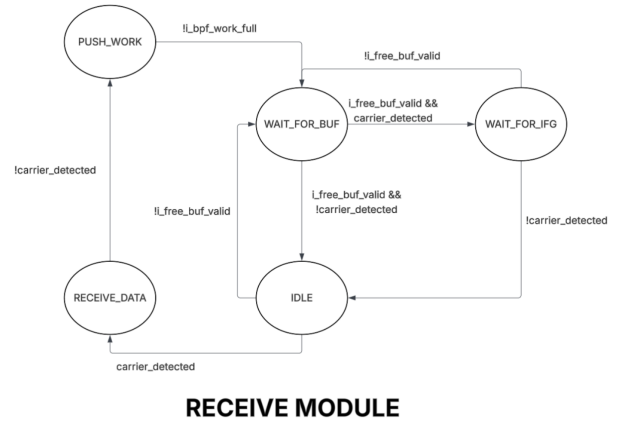


Fig. 5. Receive Module with Parallel Distribution Logic

IV. NETWORK AND BPF PARALLEL ARCHITECTURE

A. Receive Module and Distribution

The LAN8720 PHY transceivers utilize the RMII (Reduced Media Independent Interface) protocol to communicate with the FPGA, providing a 2-bit data stream at 50 MHz. The Receive (RX) module is responsible for assembling these bits into complete Ethernet frames (Figure 3) and distributing them across the parallel processing cores.

Unlike traditional architectures that write to a single shared memory block, this system implements a distributed memory architecture. The RX module interacts with a `free_buffer` manager that maintains available buffer indices in a round-robin fashion corresponding to the N available BPF cores.

Upon detecting an incoming packet preamble:

- 1) The RX module requests a buffer ID from the `free_buffers` FIFO. The ID implicitly determines which BPF core will process the packet (e.g., ID 0 maps to Core 0, ID 1 to Core 1).
- 2) The module writes the incoming packet data directly into the private BRAM of the target BPF core.
- 3) Once the packet is fully received and validated (CRC check), the RX module pushes the packet metadata (length and buffer ID) to a Demultiplexer.
- 4) The Demultiplexer routes this work item into the specific `bpf_work` FIFO of the selected core, triggering the execution phase.

B. Parallel BPF Cores

The packet filtering engine is instantiated as an array of N identical, independent BPF cores. This parallelization increases the system's overall throughput by allowing multiple packets to be processed simultaneously. Each BPF core is a self-contained unit comprising the following local resources:

- **Work Queue:** A FIFO containing metadata for packets waiting to be processed by this specific core.
- **Instruction Memory:** A BROM containing the immutable BPF program instructions.

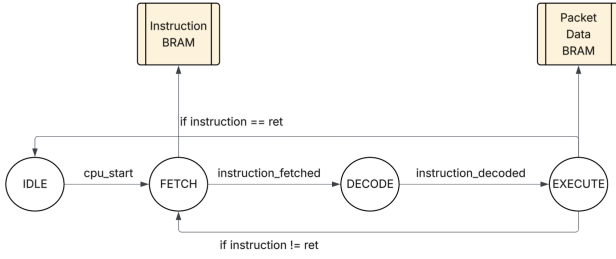


Fig. 6. BPF CPU Finite State Machine

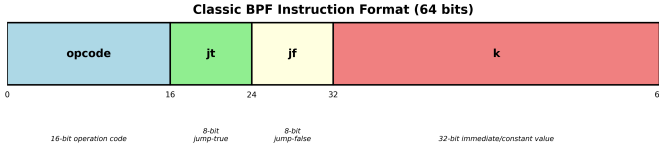


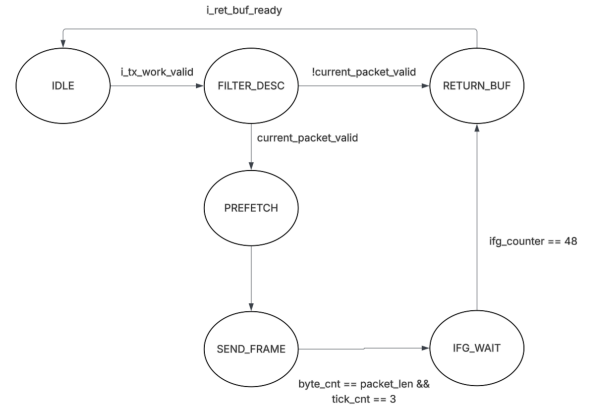
Fig. 7. BPF Instruction Encoding

- **Packet Memory:** The design employs a dual-memory architecture for each core to decouple processing from transmission.
 - **BPF BRAM:** Written by the RX module and read exclusively by the BPF CPU during filtering operations.
 - **TX BRAM:** Written by the RX module (simultaneously with the BPF BRAM) and read exclusively by the TX module during transmission.
- **TX Queue:** A FIFO where the Core pushes the processing verdict (Pass/Drop) and length upon completion.

1) *BPF CPU Execution:* Inside each core, the `bpf_cpu` implements the classic Berkeley Packet Filter virtual machine. It fetches instructions from the local BROM (format shown in Figure 7) and executes them against the data in its local Packet BRAM.

The CPU remains in an idle state until the `bpf_work` FIFO is not empty. It then dequeues the packet metadata, initializes its internal registers (Accumulator, Index, PC), and executes the filter program. The state machine (Figure 6) handles ALU operations, loads, and branches.

Crucially, this architecture deliberately replicates packet data across two distinct BRAMs to simplify the control logic. While this approach increases on-chip memory utilization, it completely eliminates the need for complex bus arbitration or locking mechanisms between the processing and transmission stages. The BPF CPU enjoys exclusive, contention-free access to its memory for instruction execution, while the TX module retains independent access to the replica memory for forwarding. This isolation guarantees that the filtering performance is never degraded by transmission activity and vice versa.



TRANSMISSION MODULE

Fig. 8. Transmit Module

C. Output Multiplexing and Ordering

A critical requirement of network infrastructure is preserving the order of packets. Since parallel cores may finish processing packets of different sizes at different times, a specialized Multiplexer (Mux) module is implemented to enforce strict ordering.

The Mux operates in a strict round-robin sequence (Core 0 $\rightarrow$ Core 1 $\rightarrow \dots \rightarrow$ Core N). It monitors the `tx_queue` of the current expected core.

- If Core 0 is the expected next source, the Mux will wait for Core 0's valid signal, even if Core 1 has already finished and is ready to transmit.
- Once the expected core provides a result, the Mux forwards the buffer ID and verdict to the global Transmit Module and increments its internal counter to wait for the next core in the sequence.

This mechanism guarantees that packets leave the FPGA in the exact order they arrived from the PHY, preventing out-of-order delivery issues at the network layer.

D. Transmit Module

The Transmit (TX) module serves as the unified output interface for the system. It receives ordered work items from the Multiplexer. The transmission process is controlled by the FSM shown in Figure 8.

For packets marked as "Accept" by the BPF verdict:

- 1) The TX module uses the buffer ID to address the specific BRAM located within the originating BPF Core.
- 2) It reads the data via the second port of that core's BRAM.
- 3) It unpacks the byte-aligned data into the 2-bit RMI format and drives the output signals to the LAN8720 PHY.

If the verdict is "Reject," the transmission phase is skipped.

Buffer Recycling: Regardless of the verdict (Sent or Dropped), the TX module performs the final step of the

buffer lifecycle: it pushes the buffer ID back into the global `free_buffers` FIFO. This makes the memory resource available for new incoming packets, closing the loop.

E. Network Statistics

The statistics module aggregates performance metrics from the entire parallel system. It monitors pulse signals from the RX module, the individual BPF cores, and the TX module to track:

- Total packets received.
- Packets accepted/rejected per core.
- Total packets transmitted.

These values are accumulated and passed through an asynchronous FIFO to the 74.25 MHz clock domain, enabling real-time visualization on the HDMI display output.

V. NETWORK STATISTICS + DISPLAY SUB-SYSTEM

A. Clock Domain Crossing

Since the majority of our logic will be running at 50MHz, we need to cross clock domains whenever we pass network statistics to our display (which runs at 74.25 Hz). We use an asynchronous FIFO buffer IP core, and pass in the number of received packets & dropped packets in the last T time units into this FIFO.

B. Packet Count Storage & Cycling

Our display controller polls the network statistics module for updated metrics every T time units. Every new statistic is added to our N length array, where the values at each index i represents the statistics for the ith polling instance. We update this array only when the network statistics FIFO is valid, and have received a *new_frame* high from our signal generator. At this point we set our ready to be high, and pop off the FIFO queue.

Whenever our packet statistics arrays are full, we shift the statistics to the right (eliminating the least recent value) and add a new statistic at the beginning of our array, to visualize the flow of time.

C. Rendering

In order to render our graphics nicely onto the monitor, we partitioned an M pixels x N pixels section of the monitor solely for our graph, where M and N are both divisible by 16. This allowed us to create a coordinate system where any (h_count, v_count) pair ends up in the ($\lfloor h_count / 16 \rfloor$, $\lfloor v_count / 16 \rfloor$) coordinate. This coordinate would represent a 16 x 16 pixel block. Every 16x16 pixel square is then assigned a packet count value. If the time index associated with the x coordinate of this square has a greater packet count value, we color the square green.

VI. EVALUATION

To validate the system, we employed a multi-layered testing strategy ranging from behavioral simulation to real-world stress testing. The evaluation focused on three key metrics: functional correctness, system throughput, and processing latency.

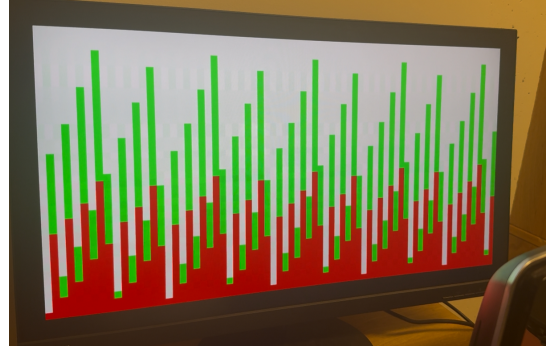


Fig. 9. Packet display with cyclic amount of packets, where green represents total packets received, and red represents the dropped packets

A. Correctness

Before hardware synthesis, we performed extensive behavioral verification. We utilized `cocotb` to write Python-based testbenches for individual modules (FIFOs, BRAMs, BPF CPU) and inspected waveforms to verify corner cases—such as queue wrapping and flag assertions—that are difficult to trigger deterministically in hardware.

Physical verification followed using a “Man-in-the-Middle” setup, where a source computer transmitted packets via an L2 raw socket to a target computer through the FPGA. Using `tcpdump` on the receiver, we confirmed functional exactness: packets matching the BPF drop criteria were strictly discarded, while accepted packets were transmitted without bit errors, corruption, or reordering.

B. Timing

The design successfully met timing constraints across multiple asynchronous clock domains. The system operates primarily on two 50 MHz RMII reference clocks (derived independently from the RX and TX PHYs) and a 74.25 MHz pixel clock for the HDMI interface.

To satisfy timing requirements given the ALU combinatorial paths, the BPF CPU architecture was pipelined. The final design achieved timing closure with a Worst Negative Slack (WNS) of **2.166 ns**.

C. Latency and Throughput

1) *Theoretical Throughput Analysis*: To determine the necessary degree of parallelization (N) required to sustain 100 Mbps performance, we modeled the system using Little’s Law. Little’s Law ($L = \lambda W$) relates the average number of items in a stable system (L) to the average arrival rate (λ) and the average time an item spends in the system (W).

We define the system constraints as follows:

- R_{line} : The line rate of the interface (100 Mbps).
- S_{min} : The minimum Ethernet frame size including overhead (64 bytes packet + 8 bytes preamble + 12 bytes Inter-Frame Gap = 84 bytes).
- f_{clk} : The system clock frequency (50 MHz).
- M_{hist} : The maximum instruction count in historical BPF kernels ($M_{hist} = 4096$ instructions).

- **CPI:** Cycles Per Instruction. We estimate $CPI \approx 4$. This accounts for pipeline overhead and the latency of byte-wide BRAM access.

First, we calculate the maximum packet arrival rate (λ_{max}) the system must handle. This occurs when receiving a stream of minimum-sized packets:

$$\lambda_{max} = \frac{R_{line}}{S_{min} \times 8 \text{ bits/byte}} = \frac{100 \times 10^6}{84 \times 8} \approx 148,810 \text{ packets/sec} \quad (1)$$

The service time (W) required to process one packet of complexity M is:

$$W = \frac{M \times CPI}{f_{clk}} \quad (2)$$

For the system to remain stable, the effective service rate of N parallel cores must exceed the arrival rate:

$$\frac{W}{N} \leq \frac{1}{\lambda_{max}} \implies N \geq \frac{\lambda_{max} \times M \times CPI}{f_{clk}} \quad (3)$$

If we aimed to support the full historical limit of $M = 4096$ instructions, the required number of cores would be:

$$N_{ideal} \geq \frac{148,810 \times 4096 \times 4}{50 \times 10^6} \approx 48.7 \text{ cores} \quad (4)$$

Implementing ≈ 49 cores is not feasible given the Look-Up Table (LUT) and block memory limitations of our FPGA. Consequently, we constrained our design to $N = 4$ cores. Using this constraint, we calculate the maximum program complexity (M_{max}) our system can actually support at line rate:

$$M_{max} \leq \frac{N \times f_{clk}}{\lambda_{max} \times CPI} = \frac{4 \times 50 \times 10^6}{148,810 \times 4} \approx 336 \text{ instructions} \quad (5)$$

This derivation confirms that while we cannot support the theoretical maximum size of software kernels, our parallel architecture ($N = 4$) supports programs up to ≈ 336 instructions. This is sufficient for the standard filtering rules required by most use cases.

2) **Real-world Performance:** We evaluated the system's performance in a live environment to measure throughput and latency under load. **Synthetic Stress Testing:** To prevent the host OS from becoming a bottleneck during transmission, we utilized custom scripts interfacing directly with `pcap` and the kernel to minimize overhead. This optimized generation setup allowed us to verify that the FPGA design successfully sustains a throughput of **100 Mbps** without packet loss.

User Experience and Latency Validation: To validate performance with realistic traffic patterns, we placed the FPGA between a router and a personal computer. In this configuration, we successfully:

- Measured round-trip time (RTT) utilizing standard ICMP pings; the FPGA introduced a negligible additional latency compared to a direct cable connection.
- Surfing the web with no perceptible increase in page-load latency.

- Streamed 4K video content continuously without buffering or interruptions.

D. Memory Utilization

The system employs a distributed memory architecture designed to eliminate contention between the receiving, processing, and transmitting stages.

Packet Storage: To support the parallel architecture without complex bus arbitration, each core maintains two identical, replicated Block RAM (BRAM) instances: one dedicated to BPF CPU reads and one dedicated to TX module reads.

- **Buffer Size:** The standard 1500-byte Ethernet MTU is rounded up to 2048 bytes (2^{11}) to allow for efficient power-of-two addressing and simple bitwise logic.
- **Capacity per Core:** Each core manages 8 buffers.

$$8 \text{ buffers} \times 2048 \text{ bytes} = 16 \text{ KB per BRAM instance}$$

Due to the dual-memory replication strategy, the total packet memory per core is:

$$16 \text{ KB} \times 2 = 32 \text{ KB}$$

Control Overhead (FIFOs): In addition to packet storage, each core utilizes local control queues:

- **Work Queue (Depth 4):** Stores the Packet ID (buffer index) and metadata for incoming tasks.
- **TX Queue (Depth 4):** Stores the verdict and Packet ID for completed tasks.

A global **Free Buffer FIFO** manages the allocation of all available packet slots across the system. Its depth corresponds to the total number of buffers in the system ($8 \text{ buffers} \times 4 \text{ cores} = 32 \text{ entries}$).

Total System Estimate: With $N = 4$ cores, the estimated on-chip memory usage for the primary packet path is:

$$\text{Total Memory} \approx (32 \text{ KB} \times 4 \text{ cores}) + \text{FIFO Overhead} \approx 128 \text{ KB}$$

Converting to bits, this utilizes approximately 1,048 Kb. This fits comfortably within the resources of the target FPGA, which provides 75 Block RAMs of 36 Kb each (totaling 2,700 Kb).

E. Checklist Status

Commitment Deliverables (Achieved):

- Displayed real-time packet statistics on the HDMI monitor.
- Successfully allowed packets to pass through the FPGA (transparent bridge).

Goal Deliverables (Achieved):

- Created a fully functioning BPF processor that correctly filters (drops/passes) packets in both simulation and hardware.

Stretch Goals (Achieved):

- Implemented a parallel architecture with $N = 4$ BPF cores.

- Designed the complex Multiplexer/Demultiplexer logic required to distribute work while enforcing strict packet ordering.

Future Work (Partial Progress):

- **Instruction Set Completeness:** The CPU currently lacks support for complex ALU operations like division and modulo.
- **Runtime Updates:** Loading BPF programs via UART remains a work in progress; currently, programs are uploaded via synthesis.
- **Memory Width Optimization:** As noted in the CPI analysis, the byte-wide packet memory degraded performance. Transitioning to word-aligned memory would significantly lower CPI and increase M_{max} .

VII. IMPLEMENTATION INSIGHTS AND CHALLENGES

A. PHY Interface Noise

One of the earliest hurdles was handling the LAN8720 PHY signals. We discovered that the `eth_crsvd` (Carrier Sense/Data Valid) signal often toggles with "junk" data before the actual packet transmission begins. We implemented robust preamble detection logic that ignores all input until the specific '01' locking sequence is detected, ensuring the RX module effectively synchronizes with the frame start.

B. Parallel Synchronization

Transitioning to the parallel architecture introduced significant complexity in the Multiplexer logic. The primary challenge was enforcing a round-robin order to guarantee that packet N is never transmitted before packet $N - 1$, even if Core N finishes execution first. Additionally, managing the `free_buffer` pool required careful design to ensure that dropped packets (which are never transmitted) still recycle their buffer IDs, preventing memory leaks that would eventually hang the system.

C. Clock Domain Crossing (CDC) and PHY Limitations

A major limitation of the LAN8720 PHY modules is that they only support outputting a clock; they cannot accept a system clock from the FPGA. This forced us to operate the RX/TX logic in the PHY's clock domain. Initially, we attempted CDC at the byte-level (converting 2-bit RMII to bytes, then crossing domains). This approach failed, resulting in synchronization loss and data gaps.

Solution: We moved the CDC boundary to the "packet level" using BRAMs and asynchronous FIFOs. By buffering the entire packet in the PHY domain and only passing stable pointers/data across domains, we eliminated the timing sensitivity.

D. Development Methodology

Separating the CPU logic into a self-contained unit, distinct from the network management and display modules, proved crucial for debugging. This modularity allowed us to verify the BPF core in isolation using `cocotb` simulations, ensuring the processor correctly executed instructions before integrating it with the noisy and asynchronous network interfaces.

VIII. CONTRIBUTIONS

Lucas worked on the design and implementation of the display controller for packet count statistics. He implemented a simulation to see a cyclically changing amount of packets (shown above), and bridged real world network statistics to the display controller for visualizing current changes in packets. He also worked on the design and implementation of the BPF virtual machine CPU, creating simulations, example packets, and example BPF programs to test all instructions.

Nour worked on integrating the FPGA with the LAN8720 board, which we had trouble working with because of its internal clock. Nour also worked on the general design of our project, constructing the infrastructure shown in Figure 4. This design consisted of solving the problem of packet storage, and consistent processing of incoming packets. Nour also worked on the parallel CPU infrastructure taking into account that we must not let packets be reordered.

Both of us researched BPF and communication protocols, to understand how to parse our BPF bytecode and develop a functioning CPU.

ACKNOWLEDGMENT

We thank Joe for his guidance and patience throughout our project.