A Minimal Network Stack on FPGA

Tianhao Huang
tianhaoh@mit.edu

December 12, 2019

1 Introduction

The TCP/IP protocol stack is the de-facto language spoken by networked systems. In a widely-adopted model, the protocol stack consists of four layers from bottom up: link layer, Internet layer, transport layer and application layer. The former three are also referred as L2, L3 and L4 respectively. For hosts (non-router nodes), NICs such as Ethernet controllers provide key link layer functionality, while the middle two reside in OS kernels and the highest layer is left for user-level applications. Core protocols in lower layers are limited. ARP, ICMP, IP, UDP and TCP are sufficient to form the basis of Internet. In this project, we explore opportunities of offloading some of these critical protocols to an FPGA. The resulting design successfully implemented L2, L3 and limited L4 functionalities. The performance objective of this project is a duplex throughput of 10Gbps.

2 High-level Design and Test Plan

2.1 System Organization

The layered protocol design of the network stack easily leads to a layered system implementation. We choose the commonplace Ethernet as our link layer implementation. Like most software stacks, our system focuses on these critical protocols for the bottom three layers:

- ARP, which builds directly on the link layer and provides mappings between physical and IP addresses of remote hosts;
- IPv4, which bases on the link layer and implements essential functionalities of the Internet layer

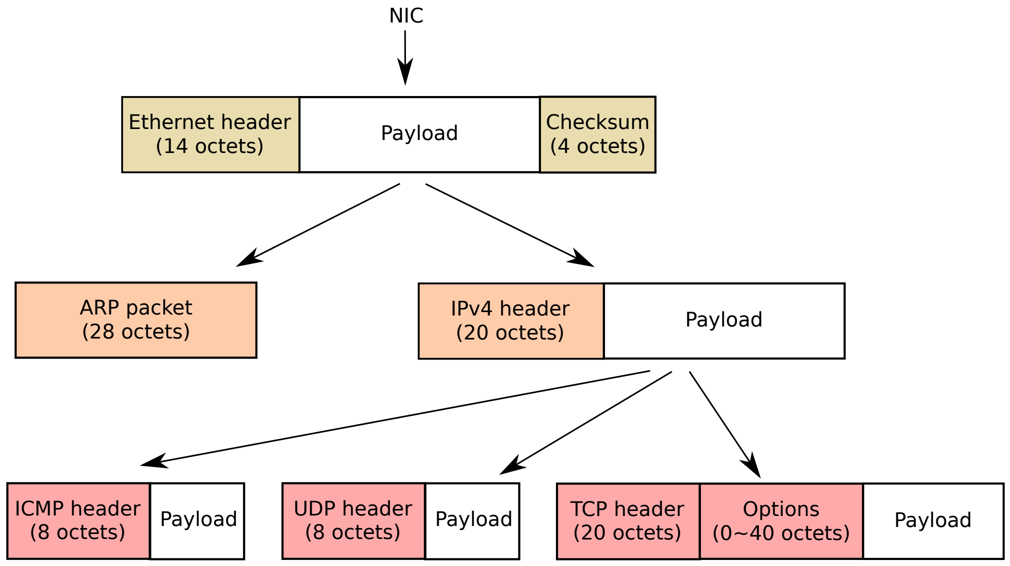

Figure 1: Stack organization of targeted protocols. Arrows points to those protocol packets which payload might contain. 1 octet = 8 bits.

- ICMP, which builds directly on the Internet layer and provides the functionality of the well-known command `ping`
- UDP/TCP, which are part of the transport layer and support application layer

Headers of a higher-layer protocol are capsulated in the payload of a lower-layer protocol. Figure 1 illustrates the header-payload relationship of protocols above.

As the lowest level of data abstraction in our system, Ethernet frames are the input and output data between our hardware stack and external network device. Xilinx's Ethernet module transceives the frames via AXI-Stream interface. In our case, connectal and host-provided facilities are used together to implement a "virtual" Ethernet transceiver to avoid spending too much time on the complicated hardware IP. As incoming packets are streamed into our network stack cycle by cycle, the receiver side of the system will parse the packets to extract headers and redirect the payload data into different packet processing pipelines, until the data is forwarded into transmit side or passed to application. Figure 2 shows our stack design in an eagle view, while Figure 3 demonstrates how the virtual ethernet device is constructed.

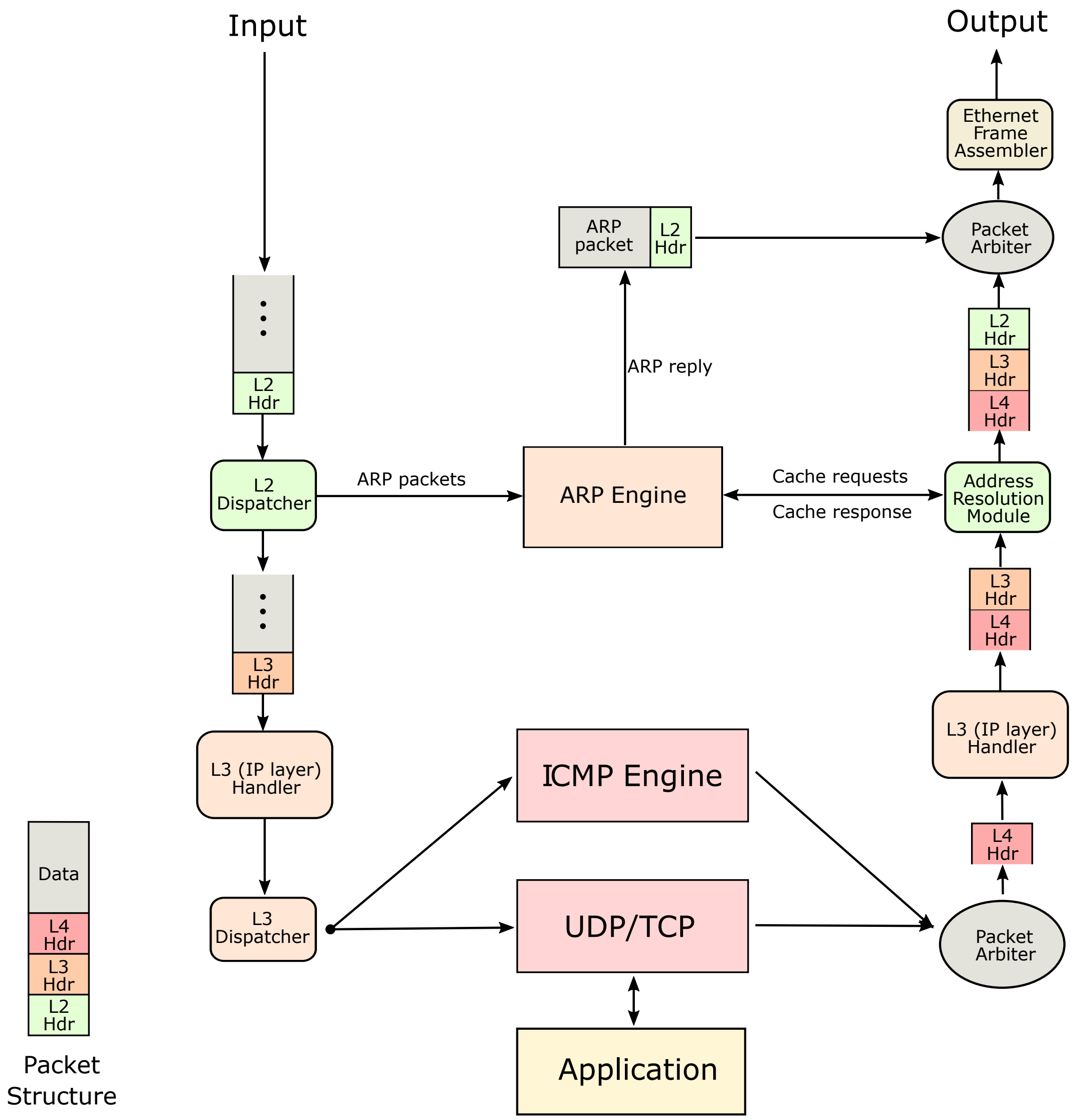

Figure 2: High-level organization of the network stack on FPGA

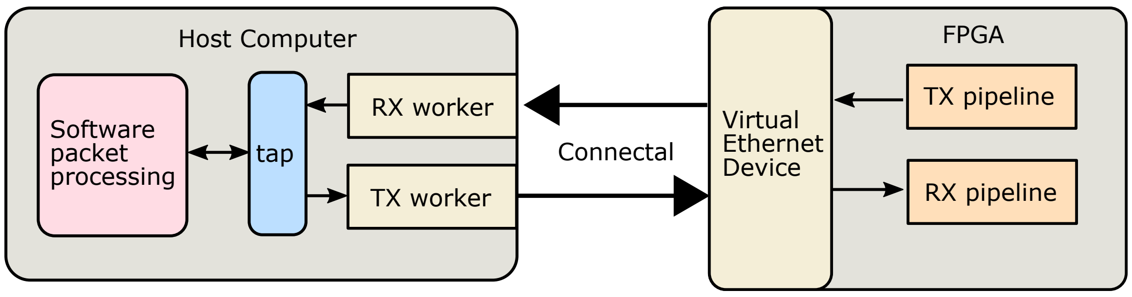

Figure 3: Virtual ethernet device for our network stack

2.2 Test Plan

The system is developed and tested in a bottom-up fashion, starting from the virtual ethernet device, lower layer protocol to higher layer ones. After the virtual ethernet is up and running, *L2 Dispatcher*, *ARP Engine* and *Ethernet Frame Assembler* are developed to set up the first RX-TX loop. We then use the command facility `arping` to generate and examine the implementation of ARP protocol. Then, *IP Handlers* and *ICMP Engine* and *Address Resolution Module* brings up the second RX-TX loop, which is tested by the `ping` command. We added packet schedulers to the TX pipeline to merge two loops. While the third loop via UDP/TCP engines haven't been finished, the test plan would follow a similar method only using different host-side packet generators.

3 Microarchitecture Design

Microarchitecture of major modules in Figure 2 is described in this section. Noted that a RX or TX packet in our system is treated as a combination of metadata and a finite data stream. There is no backward dependency in the RX and TX pipeline. By providing enough depth of FIFOs, our system could accept one data beat (8 bytes) every cycle. Due to time limitations, UDP/TCP engines have not been fully implemented.

3.1 Packet Stream Splitters and Compactors

Although not shown in the high-level diagram, packet stream splitters and compactors are fundamental parts of the modules to parse and compose packets. We use a datapath of 8 bytes throughout our RX and TX pipeline, which allows our design to run at 156.25MHz to achive the throughput goal. However, not all packet headers and payloads are aligned to 8 Bytes. Therefore, a general splitter is designed to extract headers and payloads from incoming 8-byte data beats in a streaming fashion. Oppositely, compactors are responsible for the dirty work of combining unaligned headers and payloads into an 8-byte data stream.

The microarchitecture of a splitter is shown in Figure 4. Compactors have a similar design with reversed data input and output. They both divide one data beat into two parts, store them into FIFO and reorganize them at the FIFO output. Special consideration needs to be done for the last output data beat due to the extra padding. Splitters do not know the total length of incoming packets in advance. Rather, our data beat is marked with a

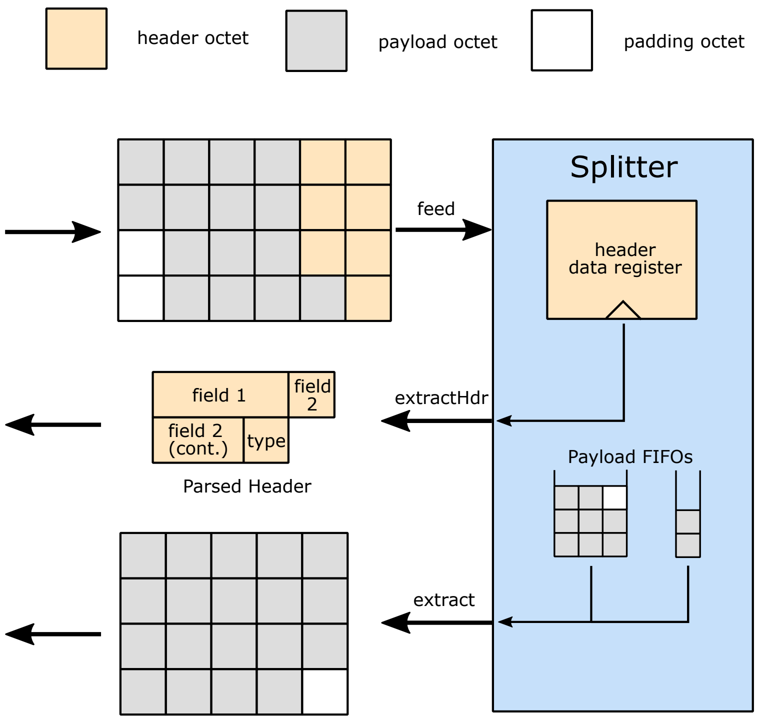

Figure 4: Design of a packet splitter. In this example, a header has 7 octets and one data beat has 4 octets.

flag indicating the end of a packet. The module is implemented in a polymorphic fashion. Given the protocol that a packet belongs to, a splitter provides interfaces to extract a parsed header and its payload stream. Compactors plays the opposite role in the TX pipeline, which hardens the provided header and payload stream into data stream. Since the lengths of a specific protocol header could be determined statically, the widths of FIFOs are known in compile time, enabling cheap circuit implementations.

Main packet processing modules (*L2 Dispatcher*, *Ethernet Frame Assembler*, *IP In/Out Handlers*, *ARP/ICMP/UDP Engine*) build upon one or both of splitters and compactors. The main difference of these modules is how they process the parsed packet headers.

3.2 Packet Arbiter

Packet Arbiter is used at the merge point of two same-level modules in the TX pipeline, such as *ARP Engine* and *IPv4 Out Handler*. It basically arbitrates which module could have their packet data output to the lower layer. We use round-robin arbitration provided by the `Arbiter` libraray in Bluespec.

3.3 General Packet Processing Module

For the RX side, the design of *L2 Dispatcher*, *IP input Handlers*, part of *ARP*, *ICMP* and *UDP* engines could be described as a general packet processing module. It accepts a parsed header and payload stream from a splitter. The parsed header is passed into a protocol-specific processor or FSM for understanding and checking header fields. Depending on the results, the payload stream is either dropped or forwarded into next layer. The structure of such a module is shown in Figure 5. Protocol-specific processing is not described in details here because they could be found in protocol easily found specifications. Noted that we ignored some rare and complicate features, such as IP fragmentation and options.

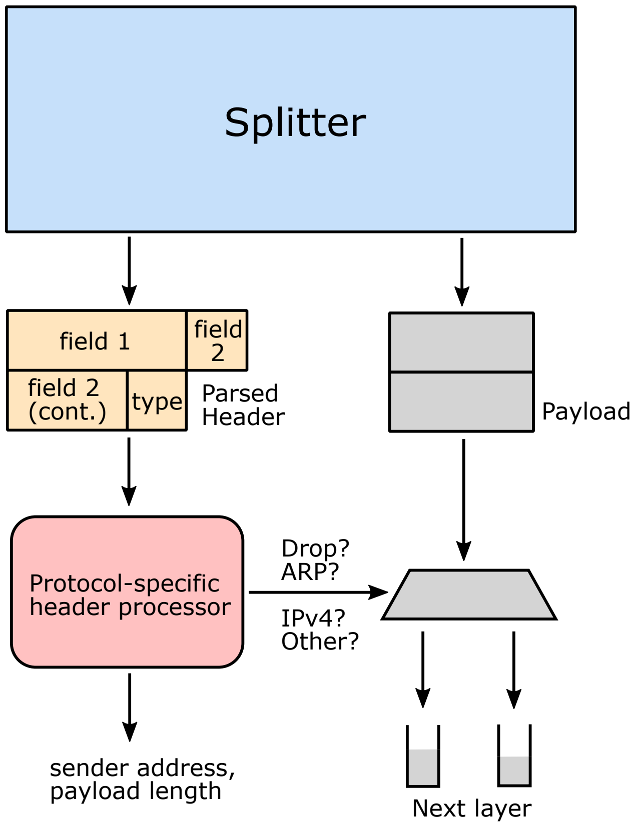

Figure 5: Design of a general packet processing module in L2

Similarly for the TX side, *IP output Handler*, *Ethernet Frame Assembler* and part of *ARP, ICMP* engines use a common structure based on compactors. It accept some key fields to generate a complete protocol header. The header is later merged with input higher-layer data stream to form a current-layer packet. For the following subsections, we omit modules that are nothing more than a general packet processing module and only clarify the design of those with substantial additions.

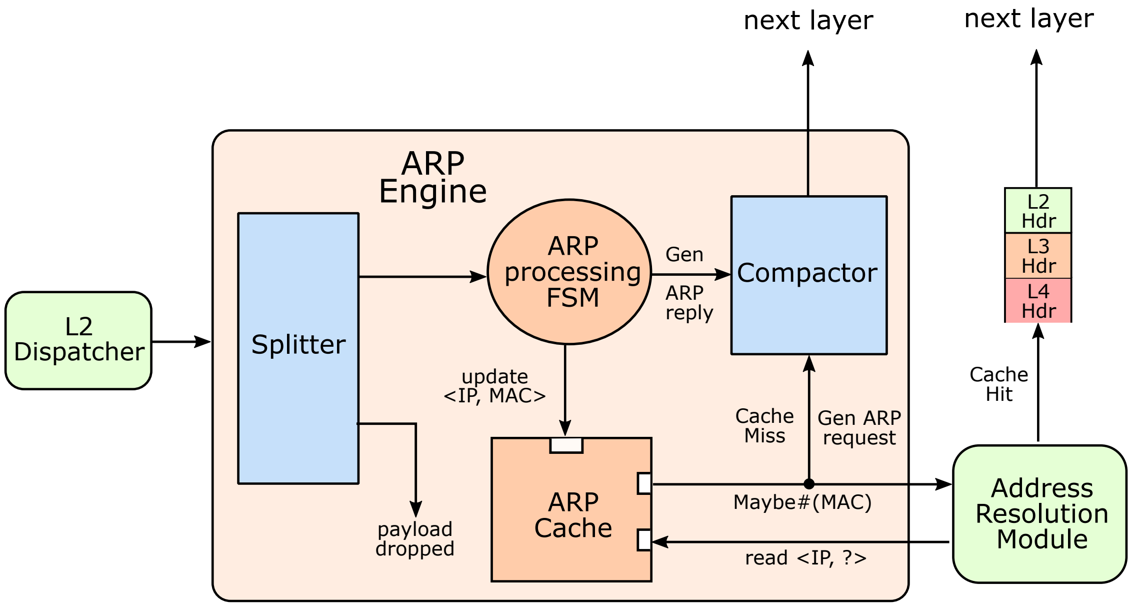

Figure 6: Design of ARP Engine

3.4 ARP Engine

ARP Engine is a moderately complex module with an ARP cache of ⟨IP, MAC⟩ address pairs. It provides physical address to Internet address mappings for higher-layer protocols. The ARP cache resides in the ARP-specific header processor in Figure 5. It is implemented as a nonblocking direct-mapped cache using dual-port BRAMs. Because of the limited storage space, we only use the last 12-bit of IP address as the index. A better solution would be using a hash function but only minor changes are required. On conflicts, the existing entry is invalidated immediately.

There are two types of ARP messaegs. The first one is ARP replies. If targeted for the stack, the ARP cache is updated with sender's ⟨IP, MAC⟩ pair silently. If not, they are dropped. The other one is ARP requests. While the ARP cache is updated the same way, an ARP reply will be generated and sent to the requester. An ARP request could be generated when, say, an IPv4 packet should be transmitted but there is no valid ⟨MAC, IP⟩ entry in the cache. The cache requests are delegated by the *Address Resolution Module*. On cache read miss, the IPv4 packet is dropped. Then ARP engine broadcast an ARP request to local network asking for the desired MAC address. After receiving the reply, later IPv4 address could be resolved successfully.

ARP Engine covers both RX and TX side of the stack. As a result, it contains a splitter near the input and a compactor near the output. Figure 6 illustrates the microarchitecture

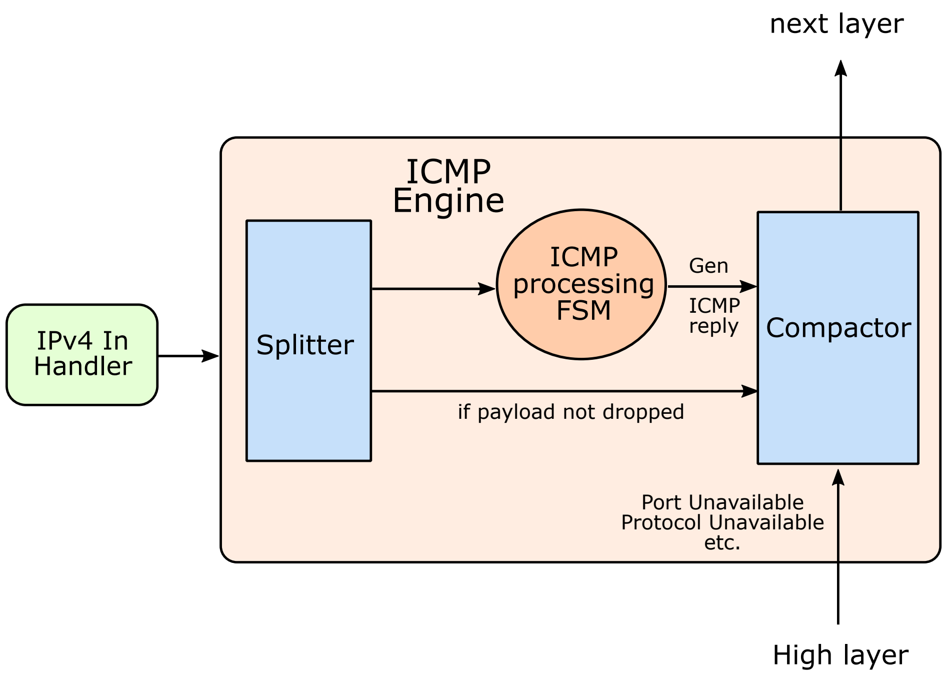

Figure 7: Design of ICMP Engine

of the module.

3.5 ICMP Engine

ICMP Engine looks very similar to *ARP Engine*, but simpler because it does not have a cache. We focus on replying to incoming ICMP echo requests (usually used by the `ping` command). For all ICMP replies and other types of requests, the engine just drops the packet. The ICMP processing FSM would copy the group id and sequence id, and then send an ICMP echo reply header to the compactor. The original payload of the echo request is forwarded to the compactor. There is another path to generate ICMP replies. When a L4 packet is malformed, not supported or not allowed by our stack, a "Network Unreachable" message with some of the packet contents should be sent to the packet sender.

3.6 UDP Engine

UDP Engine usually directly interfaces with applications. Normally, DRAM is used as UDP/TCP payload storage because when the application will consume the data is unknown to the stack. Thus, a memory allocator/deallocator is necessary for managing the packet memory. Due to time limitation, they have not been implemented yet. The following part will discuss the design plan waiting to be materialized into hardware.

Consider the position of UDP modules, it is important to provide a convenient communication mechanism between the engines and applications. A ring buffer, its head and tail pointers are used for such purposes in both RX and TX engines. When the application would like to listen on a certain UDP port, it shall ask for the memory allocator to reserve a continuous memory space to buffer incoming UDP payloads, and then update the rx-side table in the UDP RX engine. Next time when UDP engine receives a packet for the listening port, it could consult the table for DRAM write address (the tail pointer), and then write the payload. Similary for UDP TX engine, when the application sets up a UDP "socket" with a certain outgoing port, the tx-side table entry is updated. When the application is ready to send a UDP packet (payload stored on DRAM), it notifies UDP Tx FSM to generate a header and then sends the data into compactor before output.

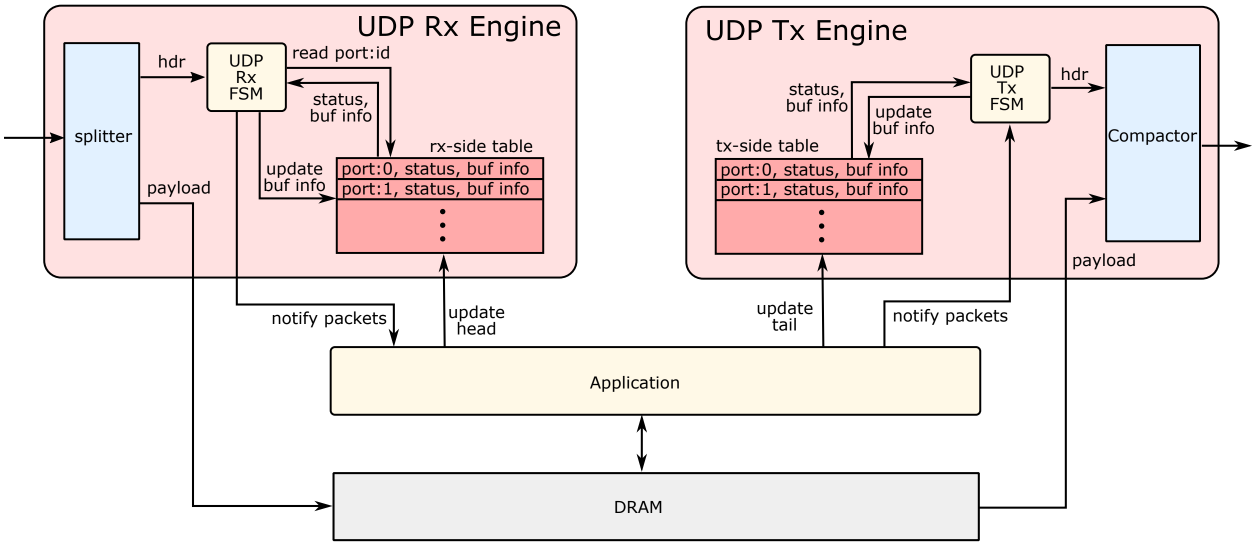

Figure 8: UDP Engine - receiver part

3.7 TCP Engine

Due to time limitations, *TCP Engine* is not implemented as part of the project. *TCP Engine* requires complicated connection state transition and out-of-order packet processing. Its implementation would be an interesting module to study later on.

4 Implementation Evaluation

The network stack in Figure 2 excluding the *UDP/TCP Engine* is fully implemented in hardware. Simulation works great, and it could correctly parse, accept and reply to incoming ARP, IPv4 and ICMP packets. The implementation is also synthesized, place-and-routed using Vivado. The hardware could handle ARP and IPv4 packets, but get into some trouble when replying to ICMP packets. Therefore, for the delay and throughput test I primarily use ARP packets. The following is a summary of results.

Item	Results
Critical path delay	5.332ns
Clocked @	166.7MHz
Theoretical throughput	10.67Gbps duplex
Resource Utilization	30.1k LUTs (9.91%) 35k FlipFlops 277kB BRAM
Loc	2231 BSV 384 C++

Table 1: Results of hardware implementation

The speed of generating packets on the host side is increased on the host side to stress our system. On simulation, the average reply delay shown by the host side command is 0.666 ms, while on FPGA the number is 0.130 ms. On contrast, the average delay of pinging another host computer is roughly 1.05 ms. These results could only be treated as a reference since our implementation nevers use a real ethernet device. Moreover, it seems our system is seriously bottlenecked by the virtual ethernet device. The througput of our system is measured at only 477.1 Mbps. Given a 60-byte packet, even if our system processes one packet at a time, it would only take less than 200 ns, which translates into a throughput of 2.86 Gbps.

Looking back, it is hard to say that creating a virtual ethernet device rather than using Xilinx-provided ethernet IP is 100 percent a good decision. At least a quarter of the time is spent on developing facilities and debugging bugs related to endianness and concurrency. Also, the performance of our system seems seriously bottlenecked by the throughput of the virtual ethernet device.