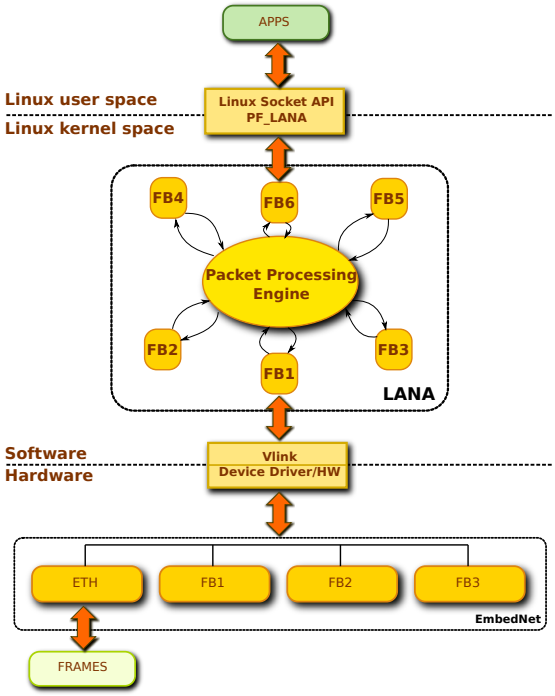
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze



Network Protocols for Embedded Devices

with Dynamic Hardware/Software Mapping

Florian Deragisch

Master's Thesis MA-2011-28
November 2011 until May 2012
Advisors: Ariane Keller, Daniel Borkmann, Dr. Stephan Neuhaus
Supervisor: Prof. Dr. Bernhard Plattner

Abstract

The rise of today's Internet began with the emergence of the *World Wide Web* and hasn't come to an end ever after. The technological advances in the fields of processor architectures and memory storage technologies in the last decade were tremendous. As a result, new devices such as smartphones, tablets or sensor nodes came to market that connect more users and machines to the Internet. This rapid development stands in contrast to the evolution of the Internet which has been going slowly. A static one-size-fits all protocol stack is no longer suited to provide an efficient communication environment that takes the diversity of devices into account. Imagine a sensor node running a full fledged TCP/IP stack which is highly inefficient in terms of battery lifetime. Fortunately, more sophisticated approaches have been proposed to take these resource constraints into account.

There have been substantial efforts in the field of dynamic and reconfigurable communication protocol stacks: The Autonomic Network Architecture (ANA) is a novel network architecture that enables flexible and dynamic rearrangements of the protocol stack. A lightweight version of ANA called LANA makes use of those principles and concepts and even provides a high-performance software framework for network protocol stacks. Another research project, called EmbedNet has been proposed as a hardware design of adaptive network nodes. Like LANA, it relies on ANA and provides means for partial reconfiguration during run-time in hardware.

Those frameworks are meta-architectures which means they lack of protocol functionality. To this end, the scope of this thesis is protocol development for network modules that can be executed in hardware (EmbedNet) or software (LANA). In order to allow for a run-time transition from software to hardware execution and vice versa, we have developed a state transition mechanism.

With the implementation of Huffman (compression protocol) and CRR (reliability protocol), we have shown that computationally expensive algorithms (Huffman) offer more potential for optimized hardware implementations, than simple flow control protocols (CRR). Huffman coding, for instance, offers a lot of parallelism that can be exploited, whereas CRR is limited by the width of its data bus. A Huffman hardware module has been designed and implemented that runs up to 22 times faster than the software module. The expected speed-up for the CRR hardware implementation is rather small, because software and hardware execution is very similar.

Memory access time to fill internal buffers is the common bottleneck that is shared among software and hardware implementations. Additionally, the implemented concept of state transition mechanism, allows flexible task migration for future protocols added to the protocol stack. At last, this thesis brought the involved frameworks one step further with respect to reliability, usability and functionality. New mechanisms and extensions have been designed and implemented that will be of great value for future projects.

Acknowledgements

With this master thesis I complete my studies in Information Technology and Electrical Engineering at the Swiss Federal Institute of Technology (ETH) Zurich.

I would like to express my gratitude to a number of people who have contributed to this thesis. First of all, I would like to thank Prof. Dr. Bernhard Plattner for supervising my thesis.

Further, I would like to thank my advisors Ariane Keller, Daniel Borkmann and Dr. Stephan Neuhaus. I am thankful for all the effort and help of Ariane, especially concerning hardware design and implementation decisions. Special thanks to Daniel who helped me a lot with his software and Linux kernel knowledge throughout the course of this thesis. Thanks to Stephan for sharing his expertise and experience in writing papers and giving talks.

I deeply appreciated the interesting and helpful discussion during our weekly meetings. I learned a lot of new things during the process of this thesis and enjoyed working on this project. It would not have been possible to write this thesis without the support and help of my advisors.

Zurich, June 2012

Florian Deragisch

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The origins of today's Internet reach back into the late 1950's when the U.S. Department of Defense wanted a command-and-control network that could survive a nuclear war. As a consequence, the Advanced Research Projects Agency (ARPA) was founded as a defense research organization. ARPA's focus eventually turned to networking, when it was trying to figure out how to provide remote access to computers. It was not until the late 1960's that an experimental network called ARPANET went on air [2]. As time went by, additional networks were connected to the ARPANET which became increasingly complex. As a result, new protocols that are still being used today such as Domain Name System (DNS), were created to organize hosts into a hierarchical system. The Internet exploded in size with the emergence of the World Wide Web (WWW) in the early 90's and has been growing ever since.

The Internet has changed a great deal since the early days and its primary objective is no longer to survive a nuclear war, but to provide the base for its future growth. The Internet is part of our daily life and changed not only in terms of users, but also concerning usage and networking devices. There is a demand to be connected at all times. Be it to communicate through social media, check emails, do online banking, join a video conference, or browse the Internet for information. In the last decade, new devices with a need to connect to the Internet showed up on the horizon. Mobile phones turned into smartphones that offer more advanced computing ability and connectivity. Sensor networks and other embedded systems are used to gather and evaluate data from sensor nodes, or send data to remote servers, for instance. The PermaSense project is an example for such a network under extreme conditions [3]. Devices as well as their offered services have different requirements to the underlying protocol stack regarding quality of service (QoS), functionality, flexibility, performance, resource usage, power usage, reliability as well as safety and security. A static one-size-fits-all protocol stack is unsuited to fulfill these challenging requirements.

Let's take a look at mobile and embedded devices: A critical requirement for mobile devices is long operation time. Ways to extend operation time are to use bigger batteries, which is rarely an option, or to reduce power consumption. A major part of a sensor nodes' energy consumption is caused by communication and data transmission[4]. A sensor node may not need a full fledged communication stack running on the device at all times. At times only basic connectivity is sufficient to fullfil the needs, for instance by using a low-power protocol stack [5], whereas at other ocasions some specific functionality may be required. The difficulty is

that with current protocol stacks, we are not able to change the stack dynamically. We would need a flexible communication stack that can be changed during run time, depending on our needs and external events. Further energy savings could be achieved through custom-made packet structures which would result in smaller header size and therefore less overhead. Another interesting approach is to migrate functionality from software to hardware and vice versa, as required. Software offers a higher level of flexibility, whereas hardware offers higher performance or lower energy consumption, depending on its design goals.

In contrast to devices and network usage are the underlying protocols such as TCP/IP that haven't changed conceptually since they had been made public. At the time IP was developed, no one expected such a widespread use as we are experiencing today. The Internet and especially IPv4 became victims of their own success: IPv4 address shortage is a real problem, and as of 15 April 2011, APNIC was the first regional Internet Registry to run out of freely allocated addresses [6]. The potential IPv4 address shortage was recognized early and the IEFT began working on the intended succeedor of IPv4 called IPv6 [7]. However, the IPv6 deployment is proceeding slowly [8].

The cases above show both the inflexibility and inefficiency of the current protocol stack and the difficulties to deploy changes. All of these weaknesses can be overcome with a flexible protocol stack in hardware and software: A first step towards this goal has been taken with the Autonomic Network Architecture (ANA) [9]. The objective of autonomic networking is to enable autonomous formation and parametrization of nodes, and to introduce self-awareness and self-expression into the system. This means that the systems will sense its environment through sensors or its internal states (self-awareness) and react appropriately in a smart way (self-expression). The protocol functionality in ANA is split up in so called functional blocks (FB). The functionality of such a FB can vary from a simple forwarding block up to a full TCP/IP stack. Those blocks form a graph-like construct that abstracts the system's protocol stack. FBs can be added, removed, linked and delinked during run time. Therefore functionality can be added, changed or removed from the protocol stack without restarting the operating system or any of its networking components. See Figure 1.1 for a comparison between a legacy network protocol stack and a flexible ANA stack. A better performing and lightweight version of ANA (LANA) that runs in the Linux kernel was developed [1].

To provide sufficient performance or energy efficiency, it is necessary that protocols can not only be executed in software but also in hardware. This implies that not only the software but also the hardware protocol stack can be changed during runtime. First steps towards such an adaptive networking environment have been presented in Lubbers et al. [10] and Keller et al. [11] and follows the concepts developed in the ANA project on a hardware level. Networking protocols are split up into smaller blocks the same way it was introduced for ANA. The core technology to enable flexibility is the utilization of run-time reconfigurable hardware. The target system is a *reconfigurable System on Chip*, an embedded device equipped with an FPGA. Further, an abstraction for programming reconfigurable CPU/FPGA systems was provided [12] that hides software/hardware boundaries. A multithreaded programming model had been extended towards reconfigurable hardware. Thread interaction and integration of software and hardware threads is built upon previous research which presented an execution environment called ReconOS [12],[13]. Further research efforts combing ReconOS and ANA principles, resulted in a reconfigurable networking environment called *EmbedNet*.

This master thesis focuses on the networking aspect of EPiCS[14] (*EmbedNet*). EPiCS uses the network architecture presented from the ANA project as a basis. LANA and EmbedNet are meta-architectures in the sense that they don't offer any protocol functionality. To this end, we develop a compression and reliability proto-

col in this thesis that can be run as functional blocks in either software (LANA), or hardware (EmbedNet). In addition, they can be migrated from software to hardware or vice versa. To enable migration, a transition mechanism was necessary which would transfer the internal protocol state and set up new functional blocks accordingly.

Compression protocols are interesting for devices with limited resources such as sensor nodes or embedded devices for efficiency reason. Compression can help to increase the energy efficiency of sensor nodes when transferring data. Further, it increases the bandwidth utilization, because the same amount of information can be sent with less data. We chose to implement the Huffman coding algorithm, because its algorithm offers a high level of parallelism that can be exploited. This parallelism is required for a fast hardware implementation where we expect to see a big speed-up. Further, the Huffman algorithm is often used as a back-end to other combined compression methods such JPEG or Vorbis [15].

Reliability protocols are critical for correct data transmission between nodes. We chose a Continuous Repeat Request protocol with selective repeat strategy, because it offers flexibility and performance through the choice of its window size. A reliability and compression functional block provide an efficient protocol stack to transfer large amounts of data in a network.

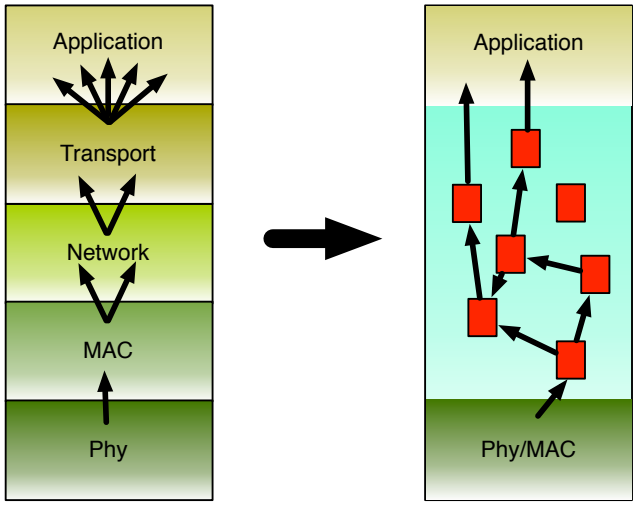


Figure 1.1: Static TCP/IP stack compared to a flexible ANA stack[1].

1.2 Goals

The goal of this master thesis is to develop several networking protocols for *EmbedNet*. Those developed protocols should run both in software and hardware. For protocols to be exchangeable, it is critical that they have exactly the same functionality, such that tasks can be migrated. Those protocols need to be implemented in *C* for execution in software and in *VHDL* for the execution in hardware. To allow for dynamic changes in the mapping of protocols to either hardware or software, a mechanism for transferring protocol state between the two implementations is needed. Depending on the used protocol, this state transition mechanism transfers internal data such as sequence numbers, frames or configuration data between functional blocks.

The implemented functional blocks will be evaluated against their software and hardware counterpart. Two protocols with specific characteristics and different

expenses in computation time were chosen in order to evaluate system bottlenecks as well as strengths and weaknesses of hardware and software implementations.

1.3 Outline

The structure of this thesis is as follows: Chapter 2 (Related Work and Background Information) presents the context of this thesis, the ANA architecture and its principles in more detail. Next, the LANA framework, which was used as the underlying foundation for the protocol stack in software, is presented. ReconOS as the foundation of Embednet is introduced and also current research in the fields of compression algorithms and reliability protocols. In chapter 3 (Design) we look into the design and architecture of software and hardware components, as well as the state transition mechanism. Chapter 4 (Implementation) delves into implementation specific details and presents implementation optimizations. We also point to limiting factors, bottlenecks and faced challenges throughout the implementation process. Functional verification and a performance evaluation where we compare software and hardware execution is done in chapter 5 (Evaluation). Next, we present testbenches and experiments used to test correct functionality. The conclusion and an outlook into future work can be found in chapter 6 (Conclusion and Future Work).

Chapter 2

Related Work and Background Information

Research projects that are related to this thesis are covered in this chapter. We will begin with a quick overview of ANA's architecture and principles. Next, we introduce its lightweight successor LANA, which is used as a software framework to build a flexible software protocol stack. ReconOS provides our software-hardware interface and is handled in the next section. Finally, we talk about compression algorithms and reliability protocols, as well as available hardware implementations.

2.1 ANA

EmbedNet [11] is part of the EPiCS project and focuses on the introduction of self-awareness and self-expression in computer networks and how to realize this goal in hardware. ANA is the underlying concept of our software and hardware frameworks and is presented in this section.

ANA defines a framework and execution environment that enables network stacks to operate in a continuously changing environment. A major principle of ANA is to strive for flexibility and genericity on all levels of its architecture. There are no rigid specifications such as protocols or header fields. ANA is a framework to host and interconnect different multiple heterogenous network instances. ANA, unlike the Internet, does not know a unique and globally shared addressing scheme. The core abstractions that build ANA's architecture and allow to model communication systems with a simple but structured framework concept are described next:

2.1.1 Architectural Abstractions

Network compartments model the ANA world from a coarse-grained point of view. A network compartment is a homogenous network region with respect to addresses, packet formats, transport protocols and other services. Each compartment is free to use whatever addressing or routing protocols it wants. However, each compartment must support a generic *compartment API* in order to offer access to its communication services. Network compartments can be modelled as black-boxes that support a generic communication API. An Ethernet segment, an IPv4 network, or a peer-to-peer-system could be a possible network compartment.

In each network compartment, a distributed set of protocol entities collaborate in order to provide communication services to other compartments. Information channels abstract the nature of those communication services (unicast,

multicast, broadcast, datagram, reliable stream, etc). When interacting with a compartment, an entity really interacts with a software component implementing the compartment's operation. Those components are called function blocks.

Functional Blocks (FBs) abstract any protocol entity generating, consuming, processing and forwarding information in ANA. ANA's protocol stack is built of functional blocks, that can vary greatly in complexity and size. A functional block can be as simple as a forwarding module or as complex as a full fledged TCP/IP stack.

Information Dispatch Points (IDPs) are inspired by network pointers [16] and are similar to file descriptors in Unix systems. Functional blocks are always accessed via one or multiple IDPs attached to it. This binding of an IDP is dynamic and can be changed during runtime which offers the possibility to reconfigure the network stack. The binding between IDPs and FBs are stored in the core forwarding table of the ANA node and are fully decoupled from addresses and names. ANA's next hop entity is always defined by an IDP.

Node compartments are basically ANA nodes that are a network themselves composed by the functional blocks running on the host. Node compartments are similar to network compartments, except that they don't provide information channels. This allows functional blocks to discover each other and interact inside the node compartment like with other network compartments.

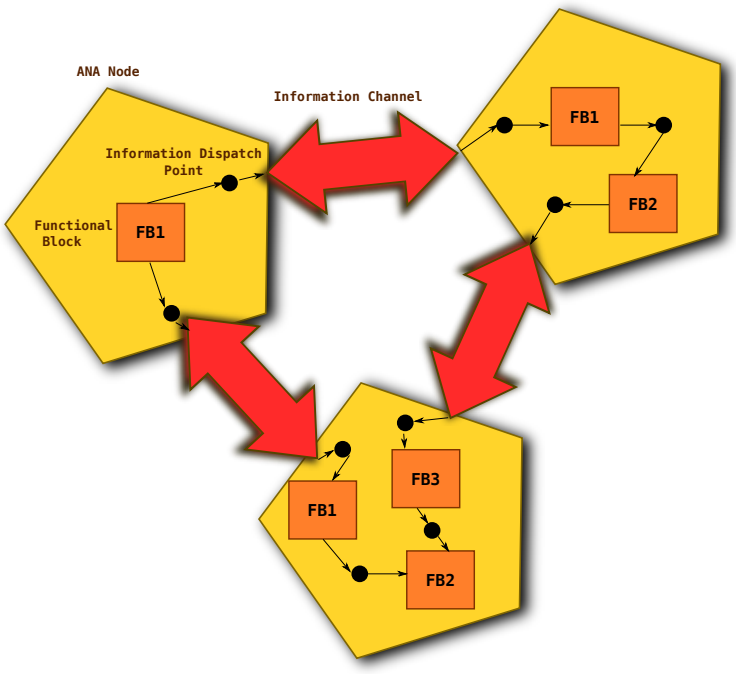


Figure 2.1: ANA sample network

2.1.2 Core Machinery

Information Dispatch Points (IDPs) are identified by a 32-bit integer and are always accessed via their numerical values. The information dispatch table (IDT) stores the bindings between IDPs and FBs in each ANA node. The IDP object structure keeps track of various information such as its owner that created the IDP, its callee that is bound to it, its status and its visibility. A key feature of IDPs is the rebinding ability during run time on demand. Since the data flow should not be disrupted, it is crucial that FBs are able to continue using their IDPs, even during rebinding. When performing IDP rebinding, it is necessary to migrate the state that a FB maintains for a given IDP.

Event notification system allows the autonomic entities of a system to react to networking changes. The system's scope is node-local and allows certain blocks to subscribe to certain events in order to be informed at a later stage, in case any of these events take place. The supported events are such as when an IDP is deleted, redirected, unpublished, busy, or ready to receive data. These events allow a functional block to witness network changes and dynamically adapt to new conditions. Such a behaviour is required to promote autonomic algorithms and reduce human interventions.

2.2 LANA

This chapter covers the lightweight version of ANA which is called LANA [17]. It shares the basics of its architecture and principles with ANA, so we will only point out LANA-specific features, that are relevant to understand how it is operating. We will look more into implementation details this time as compared to ANA, because LANA was the framework used for our compression and reliability protocols in software.

LANA is a complete redesign of ANA from scratch that aims at a slim code size and better optimized code. Various optimizations took place such as pass by reference, instead of value for network packets which are forwarded between functional blocks. The whole code runs in Linux kernel space only which improved the performance and reduced code size, because existing kernel functions and data structures could be reused. As in LANA, functional blocks build up the protocol stack. LANA's core consists of the following elements:

Packet processing engine (PPE) calls all functional blocks that are connected with each other in a graph. It basically works like a switch and forwards packets to their next hop.

Functional block builder creates new functional block instances.

Functional block notifier dispatches event messages between functional blocks.

Functional block registry and other helper data structures manage created functional block instances.

User space configuration interface performs various functional block configuration tasks such as block binding.

Some functional blocks can be part of the *virtual link layer* (vlink), which represents the interface between device driver and the LANA system. The connection between LANA and user space on the other hand is established through the *BSD socket layer*.

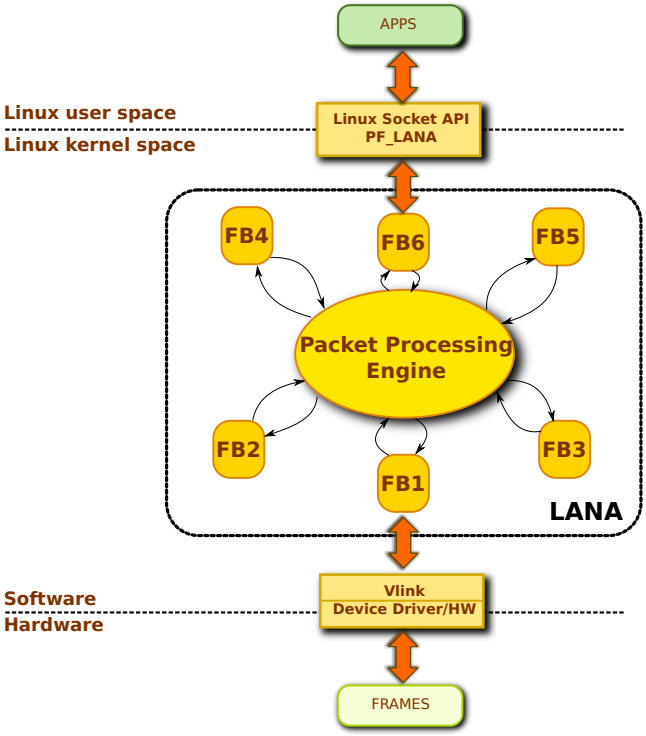


Figure 2.2: LANA sample configuration

2.2.1 Components

Figure 2.2 shows a sample configuration of LANA on our system. We will now briefly discuss the different components and how they interact with each other.

Functional blocks build up LANA's protocol stack. LANA is basically a meta-architecture or a framework without any protocol functionality. Functional blocks are indirectly accessed through their information dispatch point, similar to ANA. The protocol stack is built by connecting different functional blocks with each other and adding up more and more functionality.

Binding and unbinding is realized by event messages. Therefore, functional blocks can subscribe to an arbitrary number of other functional blocks. Connected blocks are automatically subscribed to each other. Event messages can also be used to send protocol specific messages to other functional blocks. The message notifier interface is independent from the packet data interface. During run time, it is possible to replace functional blocks, possibly with a newer version, for maintenance or updating reasons. The IDP can be transfered to the new block without other functional blocks being aware of this.

We distinguish between functional block modules and functional block instances:

Functional block modules are Linux kernel modules that provide a constructor and destructor for its functional block instance and further used functions.

Functional block instances are memory structures with a private data area which is initialized and deinitialized by the corresponding functional block module.

The **functional block builder**'s task is to create functional block object instances. Two functions are registered to the functional block builder, namely the constructor and destructor. Both of these functions need to be implemented specifically for each functional block, because private data parts may vary between FBs. Some struct members of the private data structure are mandatory, because they are used for protocol stack configuration. Once an FB is no longer in use, it is the destructor's task to clean up internal data and to free allocated memory from this FB.

The **functional block notifier** notifies FBs about new events (e.g. binding) or configuration instructions. The functional block notifier construct is distributed across all FBs, which means that each block needs to keep track of its subscribed block and its subscriptions to other blocks.

All active FB instances are stored within the **functional block registry** which consists of two functionalities. A FB's IDP can be retrieved through its name, and a FB's structure though its IDP. We can address a FB instance from user space by its name, whereas in kernel space addressing is done through IDPs.

The core of LANA is called **packet processing engine (PPE)**. Its main task is to call one functional block after the other. Each FB provides a network packet receive function handler that is invoked from the packet processing engine for every packet passing through the protocol stack. It is also the FB's responsibility to either drop or forward a packet. Forwarding is done through writing source and destination IDP into the private area of the socket buffer structure. The destination IDPs are received through configuration notifications and stored internally in their private data. The FB changes the private area of the socket buffer structure according to its internal configuration in a similar way like a network router would change a packet's MAC address. Next, the receive handler returns by telling the PPE what to do with this packet. In case the packet should be forwarded, the PPE maps the IDP address to a pointer to the structure of the functional block instance. After that, the receive handler function of the next FB in line is invoked. This procedure continues until the protocol stack either reaches a dead end or the vlink/pf lana socket.

The PPE further contains a backlog socket buffer queue, which can be used for newly created socket buffers from inside a FB. Newly created packets can be placed in the backlog queue from outside of the context of the PPE.

Besides the socket buffer, information about the path direction is given to the PPE and the FB's receive handler. We differentiate between **ingress** (towards user space) and **egress** (towards PHY).

The **virtual link** can be seen as the entry/exit point for frames from/to the PHY. It represents the glue from the OS to the LANA protocol stack. FBs in this layer are special cases that behave differently than regular FBs and implement the underlying network technology such as Ethernet, Bluetooth or others.

The **BSD socket layer** is similar to the virtual link layer except that it provides the interface between LANA (kernel space) and our application that opened the socket (user space). FBs in this layer need to provide common system calls from the Berkeley sockets API.

Finally, all LANA configurations are controlled from user space through two user space applications, `fbctl` and `vlink`. The `fbctl` tool on the one hand, performs all FB related configurations like creation or deletion of FB instances, binding, unbinding or subscription to other blocks, as well as replacing FB instances with another during runtime. `vlink`, on the other hand, controls FB of the virtual link layer.

2.3 ReconOS

ReconOS [12] is an execution environment which is based on existing embedded operating systems and extends the multithreaded programming model from the software domain to reconfigurable hardware. ReconOS is targeting CPU/FPGA systems for creating flexible multithreaded applications. Software and hardware threads integrate and communicate seamlessly and transparently with the operating system and its services.

2.3.1 Programming Model

Portability and flexibility are two major long term objectives of the ReconOS programming model. Therefore ReconOS tries to reuse established interfaces and functionalities from existing APIs. The following operating system objects are provided for ReconOS applications:

Threads represent the basic units of execution that build an application in ReconOS. Synchronization and communication is achieved by using other operating system objects.

Semaphores and Mutexes control synchronization, protect critical code or manage access to shared resources.

Shared memory, message queues and mailboxes provide means for inter-process communication. Mailboxes and message queues can provide both communication and synchronization at the same time.

Since all interthread activity uses those objects, it is not necessary for a ReconOS-thread to know whether its communication or synchronization partners are located in hardware or software. This not only provides complete transparency but also portability to other platforms, as long as those operating system objects are supported.

ReconOS software threads are POSIX-threads and handled by the standard OS scheduler, and therefore independent from ReconOS extensions. Things are different from a hardware point of view: Hardware tasks run concurrently. Furthermore, hardware description languages such as VHDL offer no mechanisms to implement blocking calls. To this end, a hardware thread's interacting operating system is managed by a single sequential state machine. The state transition in the synchronization state machine depends on control signals from the operating system interface (OSIF). Only after a previous operating system call has returned, the next state can be reached. It is the developer's task to split a hardware thread into a collection of user logic modules and a synchronization state machine.

Listing 2.1 shows the basic idea of such a state machine. We will remain in STATE_GET_ADDR, as long as we haven't received a message from a message box. Only after that will be able to update the current state.

Hardware threads take the same scheduling and stack size parameters as a software thread. These threads are used for the hardware thread's associated delegate thread running in software. Delegate threads will be explained in more detail in Section 2.3.2. It is assumed that the hardware thread is already present in the reconfigurable fabric, when creating a hardware thread in software.

Operating system objects will almost always be shared among different threads. Software threads access shared resources through global variables, for instance, accessible by all threads. This approach does not work when dealing with hardware threads. ReconOS therefore associates an array of resources with every hardware thread. The thread designer needs to define integer constants that act as indices

of the the resource array, and use the symbolic constants as arguments. Further, ReconOS can directly access same memory areas as software threads, which allows for efficient sharing of data among threads.

```
reconos_fsm: process (i_osif.clk,rst,o_osif,o_mem,o_ram) is
        variable done  : boolean;
begin
...
case state is
  when STATE_GET_ADDR =>
    osif_mbox_get(i_osif, o_osif, MBOX_RECV, addr, done);
    if done then
        if (addr = X"FFFFFFFF") then
                state <= STATE_THREAD_EXIT;
        else
                len   <= conv_std_logic_vector(RAM_SIZE,24);
                addr  <= addr(31 downto 2) & "00";
                state <= STATE_READ;
        end if;
    end if;
  when STATE_READ =>
        memif_read(i_ram,o_ram,i_mem,o_mem,addr,X"00000000",len,done);
        if done then
                sort_start <= '1';
                state <= STATE_SORTING;
        end if;
...
end case;
...
end process;
```

Listing 2.1: Code snippet from a ReconOS synchronization state machine

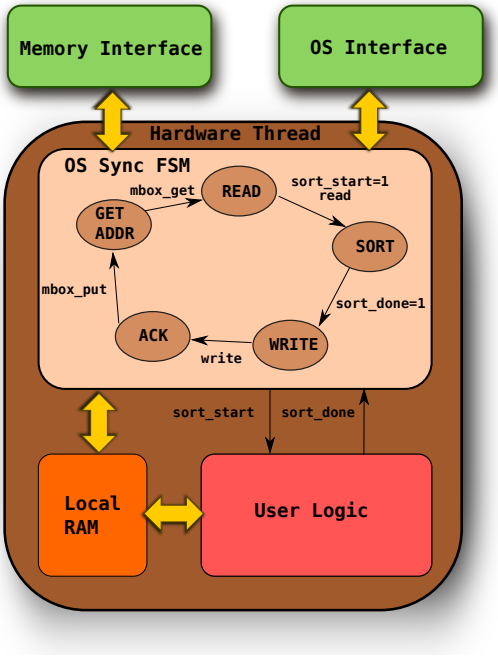


Figure 2.3: Hardware thread

2.3.2 Execution Model

Hardware threads require a run-time environment to connect them to an existing operating system kernel. Carefully defined mechanisms for synchronization and communication between hardware circuitry and the operating system are necessary.

In Reconos this is the task of the operating system interface (OSIF) which connects to the hardware thread's OS synchronization state machine and local RAM. On the other side, it connects to the system memory bus (PLB) and an OS control bus (DCR).

A fundamental aim of ReconOS is transparency of thread-to-thread communication and synchronization. This way we can easily replace a software thread with a functionally equivalent hardware thread. Every hardware thread in ReconOS is associated with exactly one software thread, it's *delegate*. The delegate thread's responsibility is to execute operating system calls on behalf of its corresponding hardware thread. As a result, the hardware thread is hidden behind his delegate thread and appears like a normal software thread to the operating system. Delegate threads are standard OS threads with additional parameters to access the OSIF hardware.

2.4 Architecture

The given development environment for the reconfigurable system on chip (rSoC) is a Xilinx ML605 evaluation board that comes along with a Virtex6 FPGA [18]. Among others, the ML605 offers, among others, the following features:

- Virtex 6 FPGA
- 512 MB DDR3 Memory
- 128 Mb Platform Flash
- System ACE CF and CompactFlash Connector
- 10/100/1000 Tri-Speed Ethernet PHY

Xilinx' Microblaze softcore is used as a CPU in our system. It is a highly configurable 32-bit Harvard architecture processor [19]. This system provides the performance to run a Linux system, which is used for the LANA software framework and the ReconOS software-hardware interface. The FPGA is big enough to leave additional space needed for our functional block modules.

Figure 2.4 shows the whole architecture of the rSoC. In the hardware domain, we have a PHY from the Ethernet interface and several hardware functional blocks. Those FBs are interconnected with each other and pass their frames from one block to another. As in software, the complexity of those functional blocks can vary significantly. The internal structure of those blocks are ReconOS-specific and were presented in Section 2.3.1.

In hardware the passing of frames between FBs is independent from ReconOS and is the task of a Network on Chip (NoC) [20]. The NoC works like a normal network that consists of routers and switches in hardware that forward packets along their way depending on their routing tables. The internal addressing of this NoC follows ANA's principles with each FB having different IDPs. It is basically LANA's counterpart in hardware.

The software hardware interface consists of ReconOS that is used to control the hardware threads through an FSM, and the NoC that can pass frames from software to hardware and vice versa. Frames are passed through a shared memory that resides in RAM and acts as a ring buffer. ReconOS can also be used to configure and initialize FBs. It can also be used to collect the state or internal data from a hardware thread.

LANA that resides in Linux kernel space, builds the software protocol stack. The vlink is responsible to forward and receive frames coming and going to the hardware.

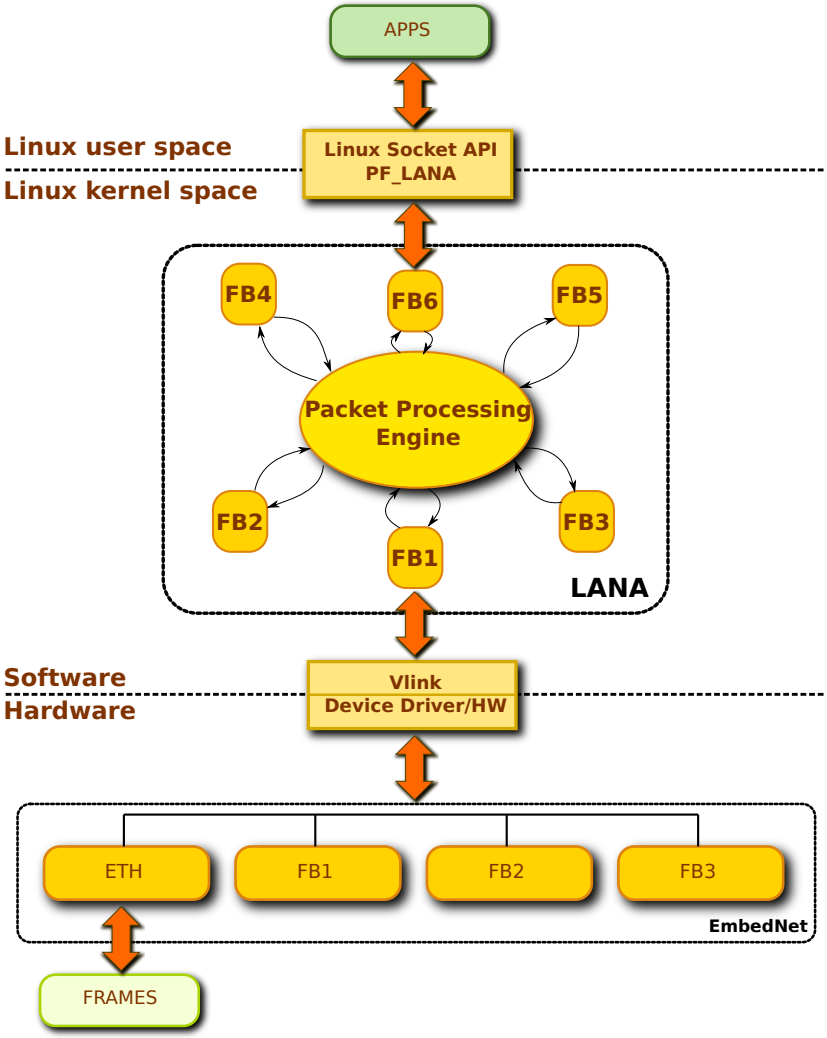


Figure 2.4: System on Chip including software and hardware

Sockets pass frames between kernel space and user space. The structure of the graph can be configured through user space tools that bind FBs.

Finally, there are the applications running in Linux user space, that open LANA sockets, used to interact with the LANA protocol stack.

2.5 Compression Algorithms

In this section we give an overview of suitable compression protocols for network frames and their hardware implementations. What follows is an elaboration of those different protocols and why we chose this specific protocol to implement. Finally, we want to introduce some existing opencores and other research projects and point out how they differ from my implementation.

The reasons to implement a compression protocol for a flexible network stack are diverse: Compressed data not only increases the effective transmission rate, but it also makes more efficient use of bandwidth. Further, smaller packet sizes increase energy efficiency, because power consumption decreases. As a trade off,

the compression and decompression may require additional cycles in software or additional circuitry in hardware. This added complexity results in an additional delay [21]. We argue here that the advantages outbalance the disadvantages, especially when compression and decompression runs in hardware. Besides payload compression protocols, further header compression protocols exist such as described in Degermark et al. [22], and Bormann et al. [23]. As opposed to other available compression algorithms, our protocol needs to be lossless. After decompression, the frame must be exactly the same as it was prior to compression.

2.5.1 IP Payload Compression (IPComp)

The IP Payload Compression protocol (RFC 3173) [24] reduces the size of IP datagrams by compressing datagrams in order to increase the communication performance between two nodes. This is especially useful when communication is over a slow or congested link. It is proposed that compression and decompression are realized through either CPU capacity or a compression coprocessor.

IPComp has the disadvantage that small datagrams are likely to expand as a result of compression overhead. Hence, seperate protocol header is necessary which is responsible for the expansion of small payloads. To this end, the protocol follows a non-expansion policy, that only transmits packets in compressed form, when its size is smaller than the original packet's size. Before utilizing the IPComp protocol, two nodes must first establish a connection, called IPComp Associaton (IPCA). There is no default compression algorithm for IPComp and it is even possible to choose different algorithms for each direction. IPComp is more like a compression protocol framework, because it does not suppose any compression algorithms to be used.

2.5.2 IP Payload Compression Using DEFLATE

RFC 2394 describes how to integrate the DEFLATE compression algorithm into IPComp [25]. The DEFLATE algorithm is widely used today by the famous PKZIP and gzip compressors, or by the freely distributed zlib library [26]. The algorithm was designed by Phil Katz and its details can be found in Deutsch [27].

The DEFLATE algorithm can be used for an arbitrarily long input data stream, using only an a priori bounded amount of intermediate storage, and hence is well suited to be used in data communication. It further compresses data efficiently and comparable to the best currently available general-purpose compression methods and is not covered by any patents [27].

The achievable compression ratio is highly depending on the input data set. English texts usually compress better than executable files [27]. The algorithm is made up from two other compression algorithms, namely **Huffman coding** [28] and **LZ77 compression** [29].

Huffman coding is a form of prefix coding, which means that there is no valid codeword in the system, that is a prefix of any other codeword [30]. For instance `{11, 10}` would be a valid prefix code, wheras `{11, 110}` would not, because after reading two bits it could either be the first code, or the second code. The algorithm starts by sorting and assembling the alphabet depending on their weight. Weight means the frequency or likeliness of a code element. The two elements with the lowest weights are chosen and linked to a parent node. Those two nodes are now the leafs of this parent node with the smaller one being on the left side. The parent node's weight is the sum of his two children nodes. Next, the two nodes with the smallest weight are chosen from all the nodes, including the parent nodes. They are linked to a

parent node again and this continues, until we have one big tree with all the nodes connected to it. In order to extract the code words, the tree needs to be traversed for every character from our alphabet starting from the root node. Taking the left edge results in adding a `0`, wheras the right edge adds a `1` to the bitstream. Figure 2.5 shows the resulting Huffman tree and Table 2.1 the corresponding codes.

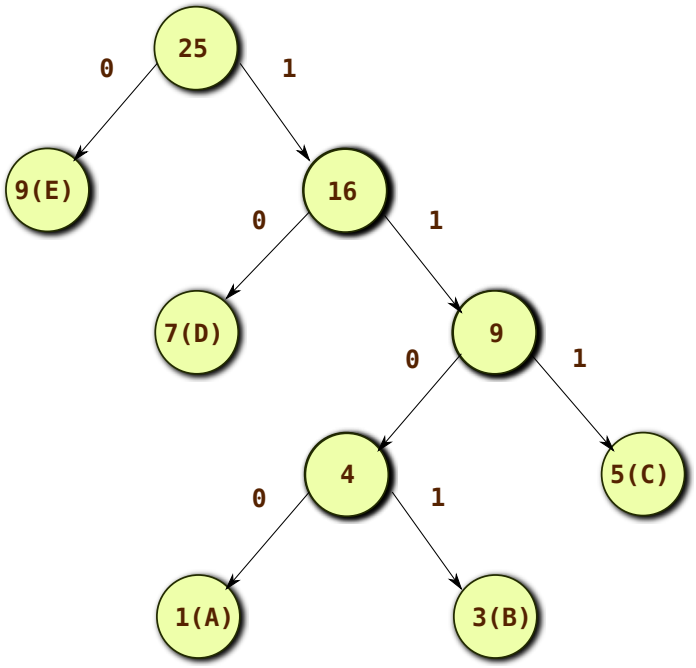


Figure 2.5: Huffman Tree

Char	Weight	Code
A	1	0011
B	3	0010
C	5	000
D	7	01
E	9	1

Table 2.1: Huffman code table

LZ77 compression works by scanning the data for repeated patterns. A sliding window is used to buffer the previous input data stream. When the next sequence of characters from the data to be compressed has been seen before in the sliding window, the input sequence will be replaced by a distance and a length. The distance describes the position of this stream in the sliding window and the length represents the number of characters, that are identical. RFC 2395 proposes using LZS compression algorithm for IPComp, which is similar to LZ77 [31].

The DEFLATE algorithm first uses the LZ77 algorithm to replace sequences that occur multiple times in the data stream. Next, the Huffman coding module is building its alphabet and code table from the already compressed data. In RFC 1951 the following Huffman coding schemes can be selected:

- Compression with fixed Huffman codes : The same code is used for all data frames
- Compression with dynamic Huffman codes : For each data frame a specialized code is created

2.5.3 Huffman

Our main field of application that we had in mind when looking for a compression algorithm was the transmission of different data types between nodes. As a starting point, we are interested in applications such as a chat between nodes, or to transfer books. We want our implementation to be feasible for small and large packet sizes. Therefore, a large header structure, like IPComp is using, was not an option. The same holds true for dynamic Huffman codes, that analyze bitstreams and build the corresponding Huffman table on the fly. This would have resulted in a much bigger design, not only for software, but also for hardware with higher resource requirements and bigger latencies. Each packet would need to carry its own Huffman table, which was used for encoding. The decoder would decode the received data based on the transmitted Huffman table. Fixed Huffman codes are fine for languages, because the frequency variations of each char are rather small.

The implementation of the DEFLATE algorithm in software and hardware was simply out of this thesis' scope. We therefore chose to implement a Huffman module in software and hardware, which is also part of the DEFLATE algorithm. Since functional blocks can be dynamically linked with each other, a later LZ77 module could be used to build up the DEFLATE algorithm on our flexible protocol stack.

2.5.4 Huffman Hardware Implementations

In this section, we cover some existing Huffman implementations in hardware from current research and existing open cores. Our main goal is to maximize the performance throughput of our hardware modules and operate at high frequencies of roughly 100 MHz.

A hardware implementation is described in Rigler et al. [32] for Huffman and LZ77. The implementation was done for dynamic Huffman codes but does not present any raw numbers such as frequency and performance or elaborate on code sizes. [33] presents various Huffman implementations that follow JPEG, MPEG and plain Huffman standards. JPEG and MPEG standards are lossy and therefore not comparable to our implementation. The used code size remains unmentioned in this paper, so it's difficult to compare with our approach. The achieved frequency of this implementation was close to 10 MHz which is too low for our reconfigurable system on chip, and therefore not suitable. In fact most Huffman implementations on FPGA from current research follow the JPEG compression standard such as Acasandrei et al. [34] that achieved 100 MHz in a parallel design. Further examples are Agostini et al. [35] and available OpenCores implementation for a JPEG decoder [36].

Current research focused more on JPEG and MPEG codec implementations. Our approach aims at a highly pipelined and parallel running encoder and decoder, that offer the flexibility to dynamically change our code alphabet during run-time. The architecture of our compression modules will be presented in the next chapter.

2.6 Reliability Protocols

To implement a realibility protocol was a consequence of our first choice to implement a compression protocol. When developing a chat or file transfer application, we need to be sure, that, i.) packets arrive at the destination host, and ii.) that they arrive in the correct order. The transfer of large files is not possible without the guarantee that packets arrive reliably and correctly. Todays most famous network protocol that offers reliability is the **Transmission Control Protocol** (TCP). TCP is the protocol of choice when a reliable link needs to be set up between two nodes. Zandy et al. [37] extends the reliability of TCP by detecting failures to TCP connections and preventing applications from becoming aware of them in the context of mobility. Unfortunately, TCP's implementation is very complex, and thus not implementable in hardware with a reasonable amount of effort. Since TCP is a flow control protocol, it is not well suited to be implemented in hardware anyway, because the expected speed up is negligible. However, an approach to migrate some parts of TCP to hardware gained in popularity in recent years: A so called *TCP Offload Engine* [38] takes over CPU expensive tasks such as checksum and sequence number calculations, sliding window calculations, connection establishment and other key features of TCP. The idea is to designate those tasks to hardware to reduce interrupts to the host CPU, copy fewer bytes over the system bus and reduce CPU requirements for the protocol stack processing.

The **Reliable User Datagram Protocol** (RUDP) is another transport layer protocol that provides reliable data transmission [39]. It's situated between TCP, that adds too much complexity and overhead, and UDP, that lacks guaranteed-order packet delivery. RUDP's list of requirements is as follows:

- Reliable delivery of packets
- Only retransmit segments that were lost and not full blocks
- Sequenced delivery is optional

RUDP is not a formal standard and has not been proposed for standardization. According to Partridge et al. [40] there were some problems with the specifications that have never been fixed.

Most of the current research related to transport and reliability protocols, is done in the context of sensor networks [41],[42]. Stann [42] evaluates on the placement of reliability on different levels of the protocol stack. Those presented protocols are designed especially for wireless links between sensors nodes. We are aiming at a more general approach that fits our flexible protocol stack and can be used for a wide variety of system architectures.

Tanenbaum [43] introduces a variety of reliable network protocols like **Idle Repeat Request** (IRQ) and **Continuous Repeat Request** (CRR) with changing complexity. Some of these presented implementations are similar to our approach, a CRR protocol, which will be presented in the next chapter.

Research in the field of reliability protocols outside of sensor networks seems to be extensions to TCP for wireless/ad hoc communication, probably because of the dominant role of TCP. Most of the presented papers deal with mobility and faulty links in a sensor network context, but those problems are concerning layer one, the physical link, and not the transport layer. Research and implementation towards hardware seem inexistent. This is probably due to the inflexibility of hardware, compared to software, and the small expected speed up from a hardware implementation.

Chapter 3

Design

The following chapter shows the architecture of the used flexible protocol stack in software and hardware. We explain the interactions between software and hardware components and their interfaces. Additionally, design decisions concerning Huffman compression and Continuous Repeat Request modules are presented from a software and hardware point of view. Finally, we present the state transition from software to hardware and vice versa.

3.1 Huffman Coding

3.1.1 Software Module

The Huffman Linux kernel module is encoding or decoding incoming payload data for a given alphabet. This alphabet could be preset as a default alphabet, which is used when the module is loaded, or it can be set dynamically at a later point in time through a proc filesystem call [44]. The module consists of the following components:

- **Packet Reception Handler:** Invoked for incoming and outgoing frames (to/from FB)
- **Event Notifier Handler:** Binding and unbinding to other FBs
- **FB Constructor:** Allocation and intialization of the private data structure. Is invoked when adding the FB
- **FB Destructor:** Deinitialization and deallocation of the private data structure. Is invoked when removing the FB
- **Module Init/Exit:** Linux kernel module constructor and destructor. Are invoked by `insmod` and `rmmod` when loading/unloading the module
- **Huffman Routines:** All Huffman related functions responsible for building the Huffman tree, extracting code words, encoding and decoding data

Next, we describe how the code base is extracted from our alphabet:

Our chosen area of application was the compression of text. We configured our system with an alphabet for English language. However, the underlying system can be used for any language. As a base for our alphabet we chose bytes, because for most programming languages a `char` (1 byte) is used to express characters. There are other languages that contain more characters than our alphabet, or special characters such as the umlauts in the German language. In order to support those

kind of languages, a base change to multiple bytes would be necessary, such as UTF-16 or UTF-32 [45].

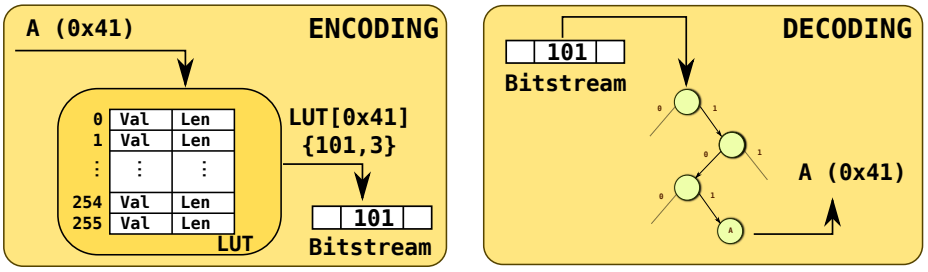
The American Standard Code for Information Interchange (ASCII) is a character-encoding scheme that is based on the English alphabet. The alphabets we used, mostly contain a subset of the ASCII alphabet including all characters, numbers, punctuation characters and other special characters.

In order to obtain the weight of each member in the alphabet, it is necessary to do a letter frequency analysis. We could use a precomputed letter frequency table, or write a script that counts all the different letters in a large textfile, such as a book for example. Next, all our letters are sorted from the smallest to largest weight (frequency) in increasing order.

Once our alphabet with its weight is set up, we build our Huffman tree as it was described in Section 2.5.2. Now that we have the Huffman tree, we extract the Huffman code for each letter in the alphabet. We traverse the tree for all possible leafs. Each leaf is a letter from our alphabet and its corresponding Huffman code is the path, which was chosen to reach it. As mentioned earlier, the path on the left adds a `0` and the path on the right adds a `1`. The whole bitstream is stored as an integer together with its length. The length is required, because the value `5` could be `101` or `0000101`, for instance. The values and lengths are stored in a lookup table with 256 indices. The indices represent the ASCII value of the original character.

Therefore, encoding is just a quick jump to the right index in our table. Once we read the code, we just need to shift it to the correct position and append it to the encoded bitstream. The encoding procedure is shown in Figure 3.1(a).

The decoding is more complicated: We are faced with an arbitrary long bitstream and don't know how long our code word is going to be. Therefore, each time we traverse the Huffman tree according to the bitstream starting from the root. Once a leaf is reached, we can be sure that it is the decoded character, since all characters are leafs in a Huffman tree. The reason being that Huffman is a prefix code. Figure 4.4 shows how the Huffman tree is traversed until the original character is extracted.



(a) Huffman encoding method

(b) Huffman decoding method

Figure 3.1: Hufman Coding in software

3.1.2 Hardware Module

From a hardware point of view, we have more options and freedom to design our architecture compared to software. We assume, the main problem and bottleneck of our software implementation is the decoding. We need to process incoming data bitwise, because we don't know where the boundaries between different characters will be. Possibilities to exploit parallelism are limited on today's processor architectures. There are systems with up to 8 CPU cores, but the size of our code

alphabet will be much larger than this. However, things look different for hardware implementation using concurrent processes.

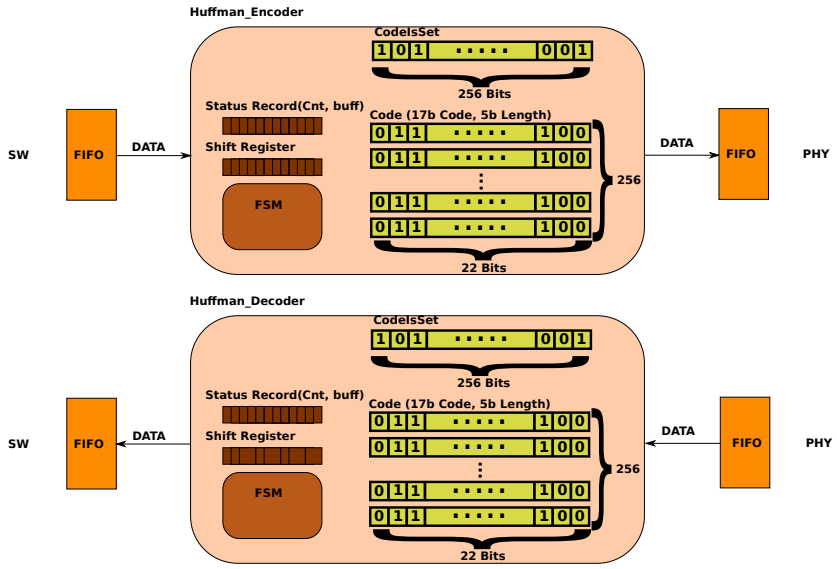


Figure 3.2: Huffman hardware architecture

Figure 3.2 shows the structure of our Huffman FB in hardware. The hardware module contains the following elements:

- **Register Bank:** A bank of 256 rows that store the codewords (comparable to the LUT in software)
- **256-bit Register:** A register that stores which codes were set in the register bank
- **Shift Register:** A shift register is to process the header of a packet
- **Status Records:** Stores internal data
- **Finite State Machine:** Controls the data input and output flow of the module

Register Bank

The most important part is provided by the memory(register bank): In order to fully exploit possibilities of parallelism that we are given in hardware, and fullfil the flexibility requirements, we need to store our codewords in clock enabled D-Flip-Flops. The clock enable signal works like a write enable that makes sure that data is not overwritten accidentally. A register bank is used to store the codewords and their lengths. Those values need to be precomputed in advance and then written to a Huffman module. In our design, we accomplish this through the ReconOS interface from the operating system. Those register banks consist of 256 register rows that store the values that are given from the ReconOS interface and the state machine. Next to the register bank, there is another 256-bit long register that keeps track of the ASCII chars that were set in the code alphabet. Each bit represents

one row of the register bank. The codes are stored in the position of their ASCII code. For instance, Figure 3.2 shows code words being stored in 22 bits. 17 bits are used for the code word, and 5 bits for the length.

Finite State Machine

The finite state machine (FSM) is responsible to detect new incoming frames and change its state accordingly. Before the module starts encoding or decoding, it needs to know whether the packet is a valid Huffman packet. Therefore, it needs to write the incoming data into a shift register to compare the Ethernet type of a packet with its Huffman-specific header. Once a Huffman frame has been detected, the FSM switches its internal state and the coding starts.

Buffer Control

When designing the architecture, one major difficulty was having variable code lengths. The length of a code for our chosen alphabet may vary between 3 and 17 bits (or more for larger alphabets) depending on its weight and the total number of members in the alphabet. The encoding maps a character of 8 bits to a code word of 3 to 17 bits, whereas the decoding maps a code word of 3 to 17 bits to a character of 8 bits, for instance. To overcome this challenge, an internal data structure that keeps track of data buffer, data buffer pointer and packet length was introduced. Those data structures in VHDL, called records, are very similar to the well known `structs` from other programming languages.

I/O

All FBs are directly connected to an asynchronous First-In, First-Out (FIFO) memory. The FIFO not only contains the data of our frames, but it also controls the Huffman-internal FSM through its signals. The FIFO signals when a new frame starts and an old frame ends. On the other hand, our hardware modules need to do the same when writing the data into the output FIFO. A more detailed low-level description of the interface between hardware FB and FIFOs will take place in Chapter 4.

3.2 Continuous Repeat Request

3.2.1 Software Module

The Continuous Repeat Request (CRR) module introduces a reliability protocol to the LANA stack. Every data transfer is acknowledged on the receiver side such that the sender knows a frame was transmitted successfully. The CRR FB in LANA is made up of the following subroutines:

- **Packet Reception Handler:** Invoked for incoming and outgoing frames (to/from FB)
- **Event Notifier Handler:** Binding and unbinding to other FB
- **FB Constructor:** Allocation and intialization of the private data structure. Is invoked when adding the FB
- **FB Destructor:** Deinitialization and deallocation and of the private data structure. Is invoked when removing the FB

- **Module Init/Exit:** Linux kernel module constructor and deconstructor, which are invoked by `insmod` and `rmmod` when loading/deloading the module
- **CRR Routines:** Are responsible for queueing incoming frames, buffering outgoing frames, controlling sequence numbers and windows
- **Packet Timers:** Timers are tracking open (unacknowledged) packets and control the retransmission of lost or delayed packets
- **Open Packets Buffer:** Is a linked list, that buffers all open (unacknowledged) packets. The oldest packet is in the first position. Buffering is necessary, because we may need to retransmit frames in case of loss or delay
- **Packet Queue:** New packets are queued in a linked list, when the transmission window is full

The buffer is used to store packets that have been sent already and are kept in the buffer until the appropriate acknowledgement has been received received. The queue is a simplo FIFO queue that stores packets which are sent at a later point in time. The CRR module was split up in a sender and a receiver. Next, we introduce the architecture and concepts of both modules separately:

The **CRR sender module** is responsible for providing reliable data transmission in our protocol stack. Each packet destined for transmission eventually invokes the sender's packet function handler. In contrast to Idle Repeat Request (IRQ) [46], CRR offers the possibility and flexibility to send several packets at once. The number of open (unacknowledged) packets is defined by the sender's window size. As long as the number of open (unacknowledged) packets is smaller than the window size, arriving packet headers are tagged, processed accordingly, and sent out directly. Note that all open packets need to be buffered, because of potential retransmission. In case that the number of open packets has reached the maximum, which is the window size, then the packet is stored in a queue for later transmission. If the packet has been received successfully, we will receive an acknowledgement (ACK) from the receiver, which states, that the frame was received successfully. The ACK triggers a decrease of the number of open packets. Hence, the next frame can be sent. The next frame will be fetched from the queue and buffered before transmission. Each packet starts a timer before it leaves the sender module. Frames or ACKs that are lost along the way cause a timeout to occur. If the corresponding ACK arrives before the timer runs out, the timer is stopped and restarted when the next frame is sent. If the timer runs out, we reschedule the oldest frame for retransmission and move it to the end of the buffer list. The oldest packets reside in the beginning of the buffer list, the same holds true for the queue. Again, the buffer stores open (unacknowledged) packets, whereas the queue (FIFO queue) stores packets that are sent next. The basic logic of the sender is pictured in Figure 3.3.

The **CRR receiver module** is the counterpart to the sender module and responsible for forwarding data frames in the correct order to the higher level protocols, as well as sending ACKs for each received frame. The architecture on the receiver side is much simpler, because we only need a packet queue to store out of order packets. The receiver expects to receive a frame with a certain sequence number. Sender and receiver need to be configured in exactly the same way, such that they have the same window size. When a frame is received, the sequence number is checked. If it matches the expected sequence number, the frame is forwarded to the user space. The expected sequence number is then incremented. If there are unforwarded packets in the queue (out of order packets), their sequence numbers are compared to the next expected sequence number. If they match, they are forwarded as well and the expected sequence number is incremented again. This continues until the queue is

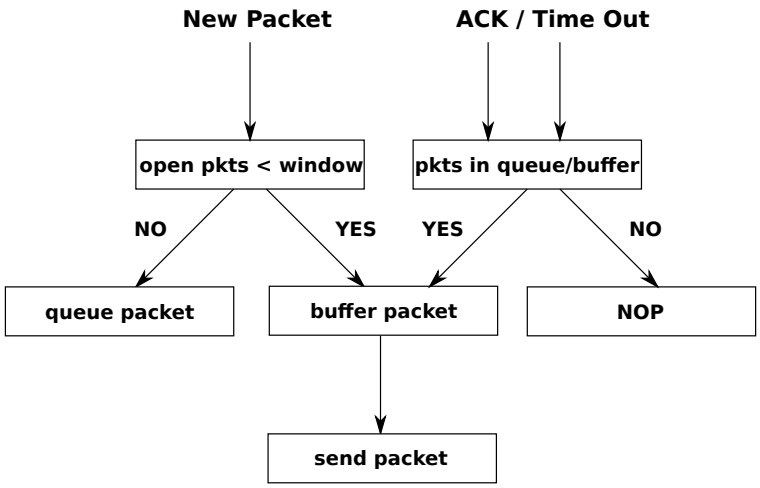


Figure 3.3: CRR sender procedures

either empty or the sequence number doesn't match. Each received frame that fits the CRR frame structure and Ethernet type is acknowledged, even if it is an old frame that has been received before. ACK frames could have been lost or delayed along the way. This way, the sender doesn't know that the receiver has already received it successfully and retransmits the frame again. Figure 3.4 shows the basic control flow from the receiver side for valid frames.

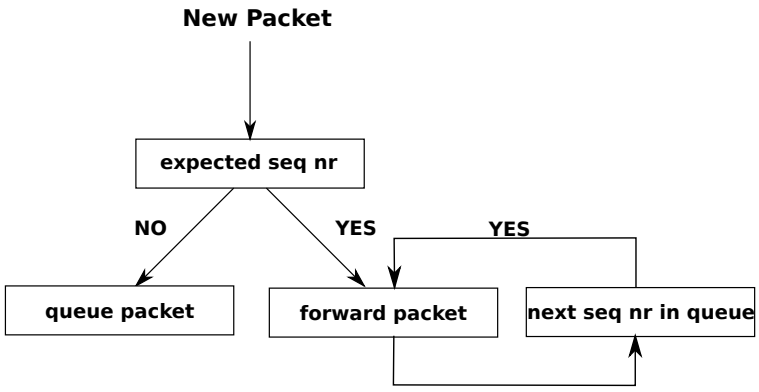


Figure 3.4: CRR receiver procedures

3.2.2 Hardware Module

The CRR hardware implementation differs from the Huffman module in many ways: CRR is a control flow protocol that introduces reliability into our data transmission. Unlike Huffman, it doesn't involve computationally expensive operations that could be done much more efficiently. CRR contains a lot of memory accesses e.g. frames need to be buffered for potential retransmissions. The hardware module, in contrast to the software module, does not contain a queue. Memory is sparse on an FPGA and we simply cannot queue frames the same way as we did in software. Therefore, we only buffer open (unacknowledged) packets and read a new packet every time an ACK is received.

There is a small delay introduced through the CRR module, because the header needs to be received completely, before the module decides, whether it is a valid CRR frame or if it's an old CRR frame. However, the predominant part of the delay is caused by the transmission time from sender to receiver and back, due to the protocol design.

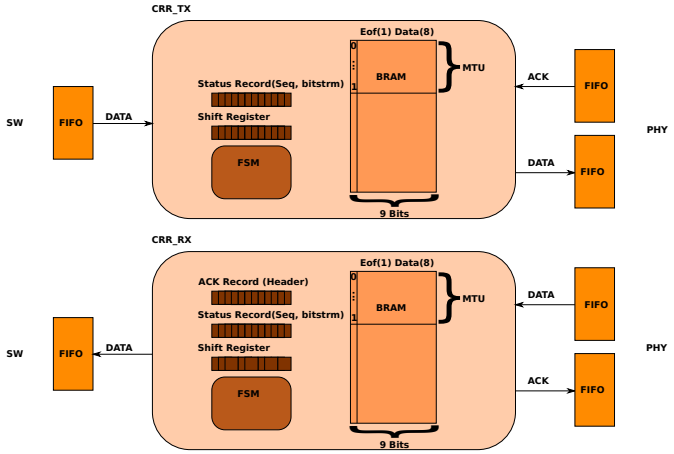


Figure 3.5: CRR hardware architecture

Figure 3.5 shows the structure of our CRR FB in hardware. The following elements make up for its functionality:

- **BRAM:** An area and performance optimized memory that is used to buffer frames
- **Shift Register:** A shift register is used to verify the header of a packet
- **Status Records:** Stores internal data
- **Finite State Machine:** Controls the data input and output flow of the module
- **Timer Processes:** Counters are used to control the timeout and retranmission of open packets

Memory is the most important resource of the CRR hardware module. It is implemented using block RAM (BRAM) which is embedded on the FPGA. The **sender module** uses the BRAM to buffer frames, because in case of a timeout they would have to be retransmitted.

The **receiver module** needs to store out of order packets, that are forwarded at a later stage, when the packet's sequence number matches the expected sequence number. Figure 3.5 shows the structure of the memory:

The memory's data width is 9 bits. The first bit is used to mark the end of the packet and the remaining 8 bits are used for the frame. Ethernet frames may vary between 64 bytes and 1518 bytes including CRC checksum [47]. Therefore, the dimension of the memory depends on the maximum transmission unit (MTU) and the window size. In order to prevent possible problems of overlapping windows, we set the maximum sequence number to twice the window size. This way, the receiver can clearly differentiate between consecutive windows. The whole memory

is $MTU \times 2 \times WIN\_SZ$. For simple addressing of all frames in our memory, the packets start at multiples of the MTU. The address of the frame with sequence number i is:

$$Address[i] = MTU \times (i - 1)$$

The shift register has the same functionality as for the Huffman module: It is used to verify the header for a CRR frame and to extract the sequence number. The FSM changes its state depending on the sequence number or on the type of the frame.

The **sender** does not evaluate the header, because all frames arriving in the sender module are tagged as CRR frames. This is done by changing the Ethernet type and adding a small CRR header. For efficiency and performance reasons, the frame is forwarded and stored in the BRAM in parallel.

A timer is started for each open packet that decrements its value each clock cycle. When the timer runs out, thus it reached `0` before the corresponding ACK arrived, the frame is retransmitted and the timer restarted. In case of several timeouts at once, they are processed from oldest to newest packet. In order not to waste bandwidth, the sender would read potential ACK frames first, before retransmitting a packet that has timed out. It may be possible that an open packet times out and at the same time, a new frames arrives that may be the awaited ACK frame. Thus, it makes more sense to read the newly arrived frame first before retransmitting the open frame.

The **receiver**, on the other side, needs to analyze the header of the frame. Invalid frames (invalid sequence number) are dropped without ACK. CRR frames with correct or incorrect sequence numbers are both acknowledged, because the sender may not have received previous ACKs, hence, the retransmission of an old (previously received) frame. Old frames are ignored internally, whereas out of order frames, frames with sequence numbers larger than the expected number, are acknowledged and stored in the memory. ACKs are sent while the corresponding frame is still being received. The hardware receiver module works exactly the same as the software module with regards to queued frames. Each correctly received frame is forwarded to higher level protocols and results in the examination of the queue. Thus, consecutive frames stored in the memory are forwarded subsequently.

The CRR modules share the same FIFO interface as we have seen for the Huffman module. One difference to be noted is the fact that there are 3 interfaces for **sender** and **receiver** each. The sender receives frames from the software side, ACKs from the Ethernet-side, and sends frames to the Ethernet side. The receiver receives frames from the Ethernet-side, sends ACKs to the Ethernet side, and forwards frames to the software side.

3.3 State Transition

To be able to change and map functionality of our protocol stack dynamically to either hardware or software, a mechanism for the state transition is needed. State transition means the migration of a protocol from software to hardware, or vice versa. The transition must guarantee that no frame loss occurs and that the end user of the system doesn't even notice the migration of protocols. For the continuous and error-free processing before and after a state transition, a transfer of the protocol state may be necessary. Further, it may be required to migrate some internal data (buffered packets) of the module as well.

Let's take a look at CRR, for instance: Our sender module runs in software and we would like to change the execution to hardware for performance or energy efficiency reasons. The software module stored some internal data (buffered packets), which needs to be transfered to the CRR module in hardware before it can start

operating. The state transition mechanism needs to transfer data such as next sequence number, open packets, buffered packets and packets in the queue. The hardware module then needs to initialize its internal structure properly, based on the internal data from the software module. The whole process of swapping tasks from software to hardware and vice versa is pictured in Figure 3.6.

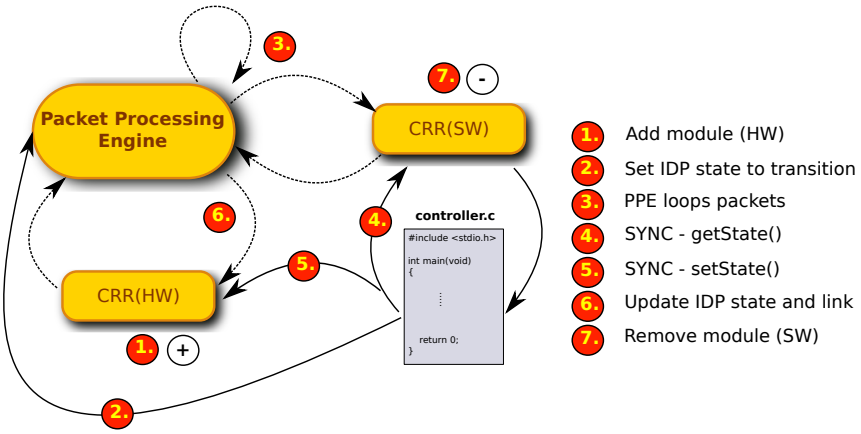


Figure 3.6: State transition mechanism

The component responsible for the state transition is a software component, called `controller.c` in Figure 3.6, that is in charge of the whole process. The first step **(1)** is to add the FB in hardware and to generate a hardware thread and its delegate thread. The FB is now present in hardware, but still not linked to our protocol stack or the packet processing engine.

Next, the state of the IDP belonging to the FB in software is set to `TRANSITION` **(2)**. As a consequence the PPE is not forwarding new packets to the FB with the corresponding IDP anymore. Instead, the PPE keeps those packets in a loop **(3)** until the `TRANSITION` state is no longer active. The binding of the FB to the PPE depends on the FB's protocol. To be correct here, FBs are not directly connected to the PPE. They are connected to other functional blocks, but the PPE is responsible for the forwarding of the packet. The Huffman module, for instance, could keep its connection to the PPE. There may still be a packet inside the module that needs to be processed and has just invoked the functional block receive handler, before the state transition took place. It is cruical that any packet still being processed is forwarded correctly from the Huffman module. However, things look different for the CRR module.

The CRR sender may have several packets that time out during state transmission. Those packets could be caught in the loop **(3)** and would result in retransmissions of those packets. Imagine the case when sender and receiver continue normal operation. The receiver would send an ACK for each retransmitted frame, even though it is an old frame. The sender on the other hand, receives all those ACKs and doesn't know what to do with it, because they don't belong to open packets. Even worse, they could be mistaken as an ACK of a newly sent frame. As can be seen, this situation could potentially cause problems and malfunctioning of the CRR modules. Contrary to the Huffman module, it is necessary for the CRR modules to loop the packets and remove the link between PPE and CRR modules to avoid problems. The two options are to either leave the link to the next FB as is, or unbind it.

Next, the internal data needs to be extracted through a `getStateSW()` **(4)** and a later `setStateHW()` **(5)** call. Totally, there are 4 different calls to migrate data

from software to hardware and vice versa. The controller invokes a function of the software module that starts creating the specific data structure. FBs may have different internal data, thus resulting in custom functions responsible for the creation of a data block. There is a specific data structure for those data blocks that is used to encapsulate the internal data. The exact implementation specific details will be discussed in the following chapter. The data is then passed back to the controller module.

Now that the data is ready to be transfered to the hardware module, we call the `setStateHW()` function. The data block is sent through the ReconOS interface and stored in the local RAM of the hardware FB. As in the software module, the process of extracting the internal state from the data block depends on the data's structure. It is the developer's task to change the existing sample FSM to his own needs. The FSM reads the data from the local RAM, and updates its registers accordingly. In the case of CRR, it is necessary to transfer buffered packets into the designated BRAM buffer of the module. Once the block data has been read, the hardware module finished its initialization and is ready for operation.

Now it's time to change the IDP's state from `TRANSITION` back to normal operation **(6)**. The hardware module was initialized and is ready to start processing incoming packets and continue the operation. Therefore, the FB's IDPs are updated accordingly and all the packets that were looping before and destined for the FB's IDP are now being forwarded to the hardware module.

The only thing which is left to do now is to remove the module running in software **(7)**. This is the basic process of migrating modules from software to hardware. The migration from hardware to software is exactly the same, but the `getStateHW()` call from a hardware point of view, controls the FSM to fill the local RAM accordingly. `setStateSW()` is reading the data block and updating its internals in software.

Chapter 4

Implementation

This chapter describes the implementation of the Huffman coding and Continuous Repeat Request modules in software and hardware. We give a closer look into data structures, header formats, interfaces and hardware logics. Finally, we discuss the state transition mechanism, used to migrate tasks, and its implementation in more detail.

In order to provide error-free communication between nodes with different processor architectures, the order of transmission for bits and bytes was described in RFC1700 [48] with the term *network order*. The network byte order uses big-endian representation, which sends the most significant part first. Intel, for instance, uses the little endian representation, which requires an endianess conversion for data types with multiple bytes. The Berkeley socket API provides a set of functions to convert 16-bit and 32-bit integers to and from network order [49]. Those functions convert between host order (little endian) and network order (big endian). **htonl(3)** transforms a 32-bit integer from host order to network order, whereas **ntohs(3)** converts a 16-bit integer the other way around. Note that all Linux system functions are referenced in **bold** letters with a reference to the section at which they are listed in the Linux man pages. Those function calls are architecture dependent: **htons(3)** will swap the bytes on an Intel architecture, but does nothing on a big endian CPU architecture. All our modules expect to receive data in big endian format.

4.1 Huffman Coding

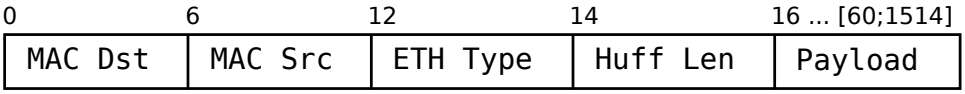


Figure 4.1: Ethernet frame and Huffman header format (length in bytes)

The frame format of a Huffman packet is shown in Figure 4.1. It contains the following fields:

- **MAC Dst:** Contains the MAC destination address of the receiver node
- **MAC Src:** Contains the MAC source address of the sender node
- **ETH Type:** The Ethernet type field **0xACDC** is used to mark Huffman frames
- **Huff Len:** Indicates the length of the original payload

- **Payload:** The data part of the frame

The Ethernet Type field **0xACDC** is used to tag Huffman frames, such that only Huffman modules process frames of their kind. Other frames will not be processed. The length field is required, because there can be bit or byte padding at the end of the frame. Encoder and decoder cannot be sure when to stop processing otherwise. Zero byte padding could be caused by frames shorter than the minimum Ethernet frame size of 60 bytes, excluding the CRC [47]. Bit padding, on the other hand, is often encountered, because it is unlikely that the length of the encoded stream will be a multiple of bytes. Hence, to be able to stop decoding in time, the decoder needs to know the length of the original payload.

A more sophisticated approach concerning headers, encapsulation and decapsulation is needed to connect several functional blocks with each other. Header design decisions have been made at an early stage of the thesis when the main goal was single unit testing. To choose a suitable header structure and stripping technique is an important design decision to be made for future functional blocks.

4.1.1 Software Module

Data Structures

Next to LANA specific members, each functional block has some specific private data members. This data can be shared between CPUs, or can be CPU-local data. Either way, proper locking techniques are necessary when reading from or writing to this shared memory. Critical code sections need to be protected by locks to prevent data inconsistency or race conditions. The private data struct of our Huffman module is presented in Listing 4.1.

```
struct fb_huffman_priv {
        idp_t port[2];
        seqlock_t lock;
        rwlock_t tree_lock;
        struct language_book *mybook;
        struct huffman_root *english_first;     /* Huffman Tree */
        struct code_book *code_en;              /* Encoding Table */
};
```

Listing 4.1: Huffman private data structure

The `port` member is responsible for the binding and unbinding to other functional blocks. There are two IDP ports: One for the ingress and one for the egress path. The `lock` member is a seqlock that provides fast and lockless (nonblocking) access to shared resources. They work for situations where the protected resource is small, simple and frequently accessed, whereas write access is rare. The basic idea behind this lock is to allow readers to access the resource freely, but check for collisions with writers. In case of a collision, they retry the access. Seqlocks cannot be used to protect data structures involving pointers, because the reader might follow a pointer that is invalid while the writer changes the data structure [50]. What follows now are Huffman-specific members of the struct:

The `tree_lock` is a reader/writer form of spinlocks. They allow any number of readers into a critical section simultaneously, but writers need to have exclusive access [51]. This lock protects all shared Huffman-specific resources. During normal operation, all accesses to Huffman resources are read only. However, in the destructor or when changing the Huffman tree, write access is required. Since our Huffman data is more complex than LANA's private data, we cannot use a seqlock. Private data that involves pointers, could result in dereferencing a NULL pointers, otherwise. The language book, containing the number of characters, the actual characters and their weights, is stored in a separate structure which can be accessed through

the `mybook` pointer. The data to build the tree is extracted from this structure. The `english_first` pointer represents the root of the Huffman tree. The Huffman tree is mainly accessed for decoding purposes, when traversing the tree is necessary. The encoding part, on the other hand, accesses the `code_en` structure. This look-up table is used in exactly the same way as it was presented in Section 3.1.1.

The most important data structures for the Huffman module are presented next:

```
struct huffman_root {
        struct huffman_node *first;
};

struct huffman_node {
        unsigned char character;
        unsigned int frequency;
        struct huffman_node *next[2];
        struct huffman_node *previous;
};
```

Listing 4.2: Huffman tree data structure

Listing 4.2 shows the structure of the root of the Huffman tree and its nodes. Huffman nodes store their `character`, if any, and their `frequency` (weight). The `next` pointer is used to reach a node's children and the `previous` member points to a node's parent. The `previous` field was introduced for the non-recursive deletion and traversion of the tree. Recursive functions may need a lot of stack memory depending on the recursion depth. Linux kernel stack size is limited to 8K [52] on x86 64-bit CPU architectures. It might be even lower on our reconfigurable system so it is necessary to avoid any recursive functions with variable recursion depths. Recursive functions were rewritten as iterative functions and that required a possibility to traverse to previous nodes. This feature is required to traverse the tree and extract the code words for each character in the alphabet. The idea is to store a pointer to each character from our alphabet.

Now we begin with the node of a character and traverse the tree backwards, after the Huffman tree has been created. We store the poiner of our start node and move to the parent node. Now, we compare which of the two nodes matches our pointer and write either a `0` when it is the node on the left or a `1` otherwise. We keep doing this until we reach the root. Finally, we store our obtained code word in the code book. This step is repeated for each element in the alphabet.

```
struct schedule_node {
    struct schedule_node *next;
    struct huffman_node *huffman;
};
```

Listing 4.3: Schedule node data structure

The `schedule_node` structure is needed, when building the Huffman tree and is shown in Listing 4.3. The algorithm to design the tree was described in Section 2.5.2. We basically choose the two `schedule_nodes` with the lowest weights. Those two nodes are placed first in the schedule list, because it is a sorted list. Next, they are linked to a parent node whose frequency is the sum of its children's frequencies. After that, this new parent node, which is actually a small Huffman tree, is inserted in the schedule list, again at the right position. The `next` member points to the next schedule node with a higher frequency. The `huffman` field is either a single node with a character or a subset of the final Huffman tree. The algorithm ends when there is but a single schedule node left, which carries a pointer to the complete Huffman tree.

```
struct language_book {
        unsigned char length;
        unsigned char character[ASCIISZ];
        unsigned short frequency[ASCIISZ];
};
```

Listing 4.4: Language book data structure

Our alphabet is stored in the structure of Listing 4.4. The number of characters in our alphabet is defined in the `length` field. The remaining two fields define the characters and their corresponding frequencies.

```
struct code_book {
        unsigned char alphabetsz;
        unsigned int *code;
        unsigned char *length;
};
```

Listing 4.5: Code book data structure

Finally, we come to our code book used to store code words and lengths. The idea of the code book is basically a LUT and was described in Figure 3.1(a). Its software data structure is displayed in Listing 4.5. The `alphabetsz` field has the same meaning like the `length` field from the previous structure. The `code` and `length` pointers are used to access arrays with 256 elements. The character's ASCII value is used to access the LUT.

Huffman Functions

The following functions provide the functionality for the Huffman module:

- **unsigned char struct_ctor(struct huffman_root *root, struct schedule_node *sched, struct code_book *book, unsigned char len)**: Initializes the Huffman tree, the schedule and also allocates memory for the code book (LUT). The function returns with `0` in case of memory allocation failure
- **struct schedule_node *construct_schedule(struct language_book *book, struct schedule_node *first, struct huffman_node **ptrArray)**: Builds a linked list (schedule) out of all characters from the alphabet, which need to be sorted in increasing weight order
- **void insert_schedule_node(struct schedule_node *node, struct schedule_node *tree)**: Inserts a schedule node at the right position in the list
- **struct huffman_node *extract_huffman_tree(struct schedule_node *first)**: Extracts and builds the Huffman tree from the existing schedule
- **void traverse_treev2(struct code_book *code_en, struct huffman_node *node, struct huffman_node **ptrArray)**: Traverses the Huffman tree iteratively and extracts codes and their lengths from the leafs. The name v2 refers to the iterative function
- **unsigned char append_code(unsigned int code, unsigned char length, unsigned char free, int *bitstream, unsigned char mod)**: Adds a code with a certain length to the bitstream
- **unsigned int encode_huffman(struct sk_buff * const skb, char *output, struct code_book *code_en)**: Encodes the original payload by looking-up the character's code and length in the LUT

- **unsigned int decode_huffman(struct sk_buff * const skb, char *output, struct huffman_node *node)**: Decodes the payload of a frame by traversing the tree according to the bistream
- **int fb_huff_proc_show(struct seq_file *m, void *v)**: Prints the language book, when the module's proc file is read
- **ssize_t fb_huff_proc_write(struct file *file, const char __user * ubuff, size_t count, loff_t * offset)**: Creates a new Huffman tree from the language book, which is written to the module's proc file in increasing order
- **void delete_treev2(struct huffman_node *tree)**: Frees all elements of the Huffman tree iteratively
- **void deconstruct_schedule(struct schedule_node *first)**: Destructs the linked list (schedule) and frees all elements

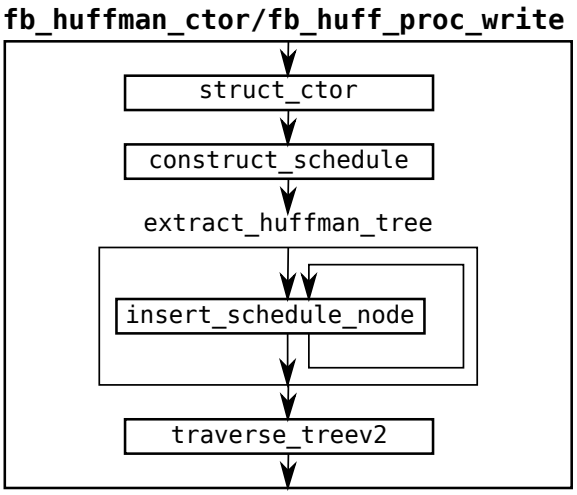


Figure 4.2: Overview of the Huffman constructor

Figure 4.2 shows in which order those functions are called to build the Huffman tree. Afterwards the module is ready to process incoming and outgoing frames.

Encoding

All frames with `EGRESS` direction (PHY direction) that are forwarded to the Huffman module from the PPE and have the correct `ETH TYPE` (0xACDC) invoke the **encode_huffman** function. First, the length field is extracted from the header. Next, the payload is encoded byte by byte. For each char the corresponding code and length are extracted from the LUT. These values are passed to **append_code** where the code is appended to the new packet payload. After all chars have been decoded, the function returns with the size of the new payload. It is possible that the new payload length is much larger than the original. This could be the case for unlikely characters. Depending on the outcome we either call the Linux kernel function **skb_trim**, which cuts the length of a buffer, or **skb_put**, which extends the data area of the buffer. Figure 4.3 shows this procedure for involved Huffman functions.

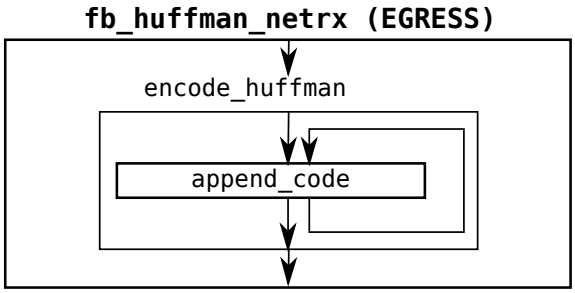


Figure 4.3: Involved Huffman functions for encoding

Decoding

Huffman frames that invoke the receive handler through the `INGRESS` direction (SW direction) are decoded. The process is similar to the encoding and starts with extracting the length field as well. Now the decoder follows the payload bit by bit and traverses the tree accordingly. Every time a leaf is reached, the according character is written to the new payload. After all characters have been decoded, the data buffer area is either reduced or enlarged in size. Figure 4.4 displays this behaviour for participating Huffman components.

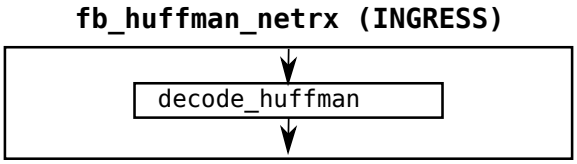


Figure 4.4: Involved Huffman functions for decoding

Optimizations

In the following section, we present optimization techniques that improve the performance of our software module.

There are many branches in our code, hence the program flow can vary greatly. A CPU uses branch-prediction to improve its performance, which means that it will try to guess which way an if-then-else structure will go before it is known. If it turns out later that the correct branch was chosen the CPU continues executing, whereas a misprediction flushes the pipeline and fetches the correct instruction this time. Mispredictions cause a delay and therefore reduce performance. To reduce the overall delay, it is possible to tell the compiler (gcc only) which branch should be favoured and is taken most of the time. This helps to reduce the number of mispredictions. We can use the `likely` and `unlikely` macros from the Linux kernel in Listing 4.6 to indicate which branch is likelier to be taken during run time [53].

```
#define likely(x)       __builtin_expect((x),1)
#define unlikely(x)     __builtin_expect((x),0)
```

Listing 4.6: Likely and unlikely macros

The encoding and decoding options were written on a low level and close to CPU opcode including `AND/OR/SHIFT` instructions. The compiler should be able to create optimal and fast code for these parts.

LANA can be run on multicore systems and therefore a proper locking technique is very important. In Section 4.1.2 we explained our lock choice. After building the Huffman tree and extracting the code words, we require read access to our private data most of the time. An exception is a new alphabet which is written into the module. The use of a per CPU-local data structure enables us to run the Huffman coding in parallel on several CPU cores in a fast way. A `rwlock_t` allows an unlimited number of readers into the critical section, but still allows to block readers in case of write accesses.

4.1.2 Hardware Module

Before starting with specific hardware implementations, we introduce some implementation guidelines and concepts that were used.

We mostly followed the VHDL naming conventions presented from the Microelectronics Design Center to introduce a logical and clear naming concept [54].

Another inspiration was the structured VHDL design method proposed by Jiri Gaisler [55]. The LEON3 softcore processor was developed and implemented according to this design method. It suggests implementing every VHDL entity around two processes. One combinational and one sequential process that contains only registers.

Further, the use of record types is highly recommended for readability reasons, especially for large designs with a huge amount of signals. Not only the port list becomes shorter and more readable, but making changes to the interface becomes much easier. The same holds true for registers: When adding more registers, we only need to add a new element to our record type definition instead of adding new signals and adding them to the sensitivity list.

Not all designs are equally well suited to be implemented according to those guidelines mentioned in the paper above, but some points, such as the use of records hold true for most designs.

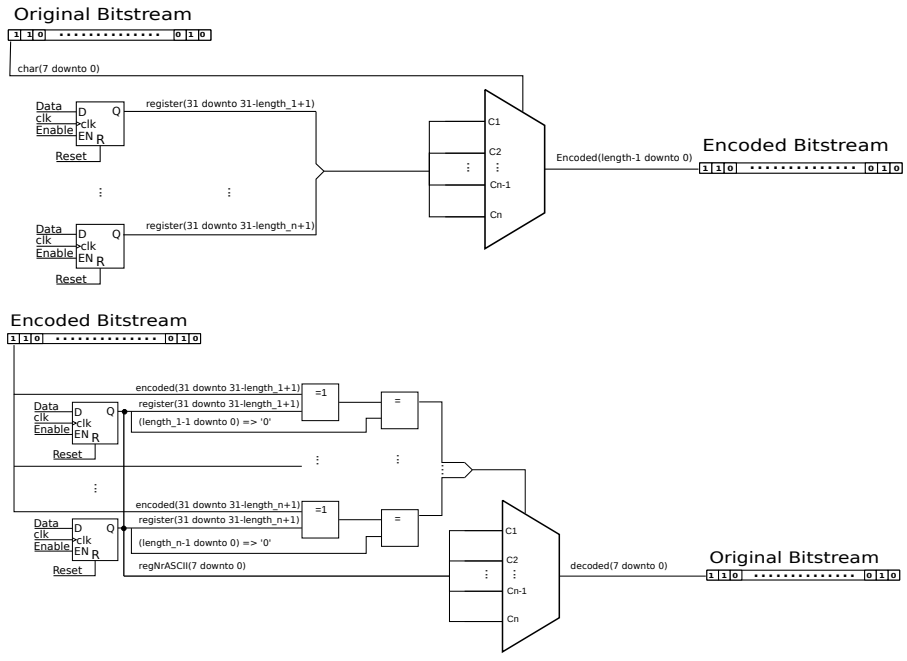


Figure 4.5: Encoding and decoding in hardware

The hardware level design of our Huffman coding modules is presented in Figure 4.5. One byte at a time is read from the original bitstream and is used as the index to choose the correct code word from the register bank. The retrieved code is then appended to the encoded bitstream.

To decode the encoded bitstream, we need to compare the bitstream with all, possibly 256, code words in parallel. Those code words have different lengths, but we can be sure to have at most one match, because Huffman is a prefix code. This means that each character is a leaf in the Huffman tree, which makes it impossible to have more than one match. The correct character is then written to the original bitstream. The big advantage of this hardware architecture is the exploitation of parallelism, which cannot be achieved to that extent in software with today's CPU architectures to the same degree.

Data Structures

The encoder and decoder hardware module use the internal data structures shown in Listing 4.7. It should be mentioned here that the current implementation accepts code lengths between 3 and 17 bits. This is, because for our current design we use a default code that uses code lengths between 3 and 17 bits. The maximum code length is a constant that can be changed easily in the code. For the following figures and discussion we assume a maximum code length of 17 bits. The `bufferIn` field and its counterpart `posbufferIn` are used to buffer data. `bufferIn` holds 3 bytes which is enough to store the longest code word. `posbufferIn` keeps track of read and written bits and changes its position accordingly. `len` stores the number of characters to be encoded or decoded and is extracted from the header. The `eof` bit is set after the last data byte has been read from the FIFO. Even though encoder and decoder share the same record, its usage is different. The process of encoding and decoding will be presented later in this section. What follows are the interfaces of our functional blocks:

```
type huffman_enc_type is record
        bufferIn        : STD_LOGIC_VECTOR(23 downto 0);
        posbufferIn : integer range 0 to 24;
        len                     : integer range 0 to 1600;
        eof                     : STD_LOGIC;
end record;

type huffman_dec_type is record
        bufferIn        : STD_LOGIC_VECTOR(23 downto 0);
        posbufferIn : integer range 0 to 24;
        len                     : integer range 0 to 1600;
        eof                     : STD_LOGIC;
end record;
```

Listing 4.7: Huffman encoding record

Interfaces

Every FB in hardware has a ReconOS interface and a NoC interface. The basic connection between those two interfaces were shown in Figure 2.3 and Figure 3.2. We focus on the interface between FB and NoC because it is responsible for the delivery of frames to our module. It is important to know that ReconOS provides access to a shared memory and uses `mailboxes` for communication and synchronization.

The interfaces to a FIFO for input and output can be seen from Figure 4.6. The interface for incoming frames is as follows:

- **DataInxD:** The data signal with a width of 1 byte
- **ValidInxS:** Indicates the data's validity

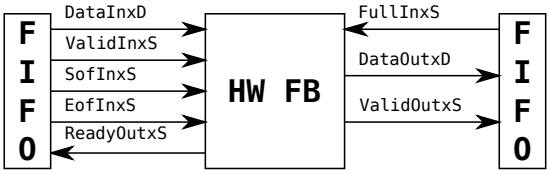


Figure 4.6: FIFO interface between NoC and FB

- **SofInxS:** Goes high at the start of a frame
- **EofInxS:** Goes high at the end of a frame
- **ReadyOutxS:** Goes high when the modul is ready to fetch data and controls the FIFO's read enable

The interface for outgoing frames is similar:

- **FullInxS:** Is high when the FIFO is full
- **DataOutxD:** Is 10 bit wide and contains Sof & Eof & Data
- **ValidOutxS:** Indicates the validity of the data

Encoding module processes

The following processes are responsible for the main functionality of the encoder module:

- **encodeComb**: Reads a character from DataInxD and returns the encoded value which was read from the register bank
- **controldataComb**: Reads the data from encodeComb, updates internal data and prepares the output of decoded data
- **SYNC_PROC**: A sequential process that updates all registers
- **Counter**: A counter used to write the header into a shift register
- **NEXT_STATE_DECODE**: Determines the next state of the coding FSM
- **reconos_fsm**: The synchronization FSM which controls the ReconOS interface

The encoding is done in the **encodeComb** process, which reads a character, and checks if its ASCII value has been set before. If so, the length of its code word is read from the register bank at the correct index. Next, a code word with the correct number of bits is read and assigned to a signal.

This code word is later read from **controldataComb** and buffered in the record as in Section 4.1.2. This process also controls the `ReadyOutxS` signal: The length of an encoded character may be between 3 and 17 bits, but we can only write 8 bits per cycle at the output. Therefore we need at least 17 free bits in our buffer, before reading a new character from the input. Each cycle 1 byte is written to the output, whenever possible, which is, when there are more than 8 bits in the buffer.

`SYNC_PROC` is responsible for resetting and updating registers, states and records. The **reconos_fsm** process is used to initialize the register bank and mark each entry as set. The memory structure has already been presented in Figure 3.2. The initialization is done through ReconOS message boxes. The codes `0xDEADBEEF` enables code reception and `0xDEADDA7A` disables it. This is to avoid unwanted changes to our codes from other hardware modules, that may receive message boxes.

Decoding module processes

The decoder's functionality is made up from the following processes:

- **decodeComb**: 256 processes compare the encoded bitstream with all codes from the register bank
- **hot2binComb**: This process performs a *one hot to binary* conversion
- **controldataComb**: Reads data from the input and controls a separate decoding FSM
- **OUTPUT_DECODE_DECODE**: Feeds decodeComb with new data and acts as the output decoder of the decoding FSM
- **SYNC_PROC**: A sequential process that updates all registers
- **Counter**: A counter used to write the header into a shift register
- **NEXT_STATE_DECODE**: Determines the next state of the coding FSM
- **reconos_fsm**: The synchronization FSM which controls the ReconOS interface

Encoder and decoder both share the same coding FSM, which controls the arrival of new data and the analyzation of header fields. They also share the same ReconOS synchronization FSM, because both need to initialize their memory (register banks) before operation.

The decoder has another FSM which is solely responsible for decoding. In order to meet design requirements, we had to introduce pipeline stages that shortened our longest path which results in a higher maximal frequency. This decoding FSM is controlled by **controldataComb**. Before starting a new decoding cycle, we need to have at least 17 bits in our buffer. This is to be sure, that we have a match when comparing all code words. The FSMs will be presented in the next section. The **decodeComb** process was generated 256 times and `process(i)` compares the input stream with `code(i)`. In case of a match it sets a bit number `i` in a 256-bit long bit vector to 1. The conversion from binary value of this vector to the position value is done in **hot2binComb**. The decoding process was illustrated in Figure 4.5.

Finite State Machines

We will start this section with the introduction of the coding FSM, which is present in the encoder and decoder module. Its task is to recognize new incoming frames, check their headers, extract the length of the payload and detect the end of frames. Figure 4.7 shows the existing states and their edges.

NEUTRAL is the starting state of the coding FSM. The next state `ANALYZE` is reached upon the detection of a new frame.

ANALYZE is responsible for checking the header of incoming frames. Depending on the Ether Type the packet is either forwarded as is through the `FORWARD` state, or encoded/decoded through the `PRECODE`/`WAIT_PRECODE` state.

WAIT_PRECODE is a transition state that is waiting for the module to become ready to read new data. This may be the case when the output FIFO is full. The length of the payload is also extracted in this state.

PRECODE extracts the length of the payload and moves on with the `CODE` state similar to the `WAIT_PRECODE` state.

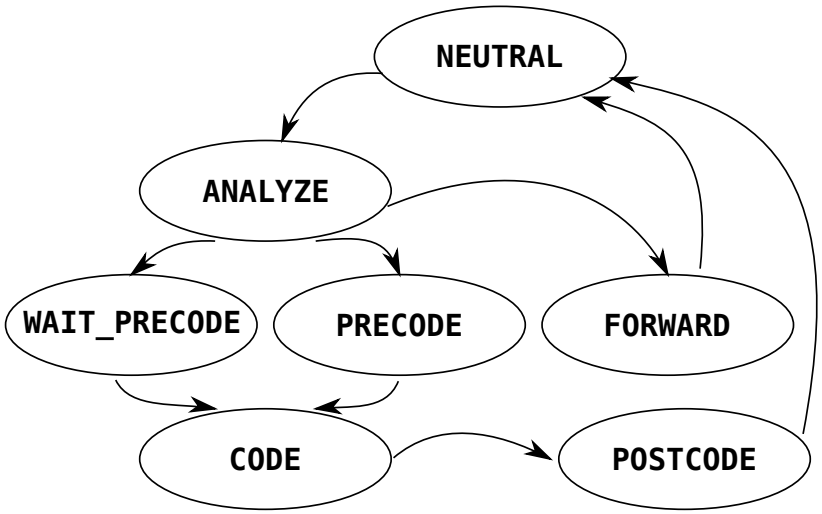


Figure 4.7: Coding FSM

CODE is the state during which the coding takes place. We remain in this state until the end of the frame has been detected which results in a change to `POSTCODE`.

POSTCODE has to continue coding until all characters from the payload have been processed. After this state a new cycle starts with `NEUTRAL` state.

FORWARD simply forwards the incoming packet byte by byte until the end of frame has been detected, which results in a change of the state to `NEUTRAL`.

The decoder's FSM will be presented next: Its task is to control the different pipeline stages of the decoding process. It also controls the inner ready state of the module, because during decoding it cannot fetch new data. Figure 4.8 shows the order, states and connections of the FSM.

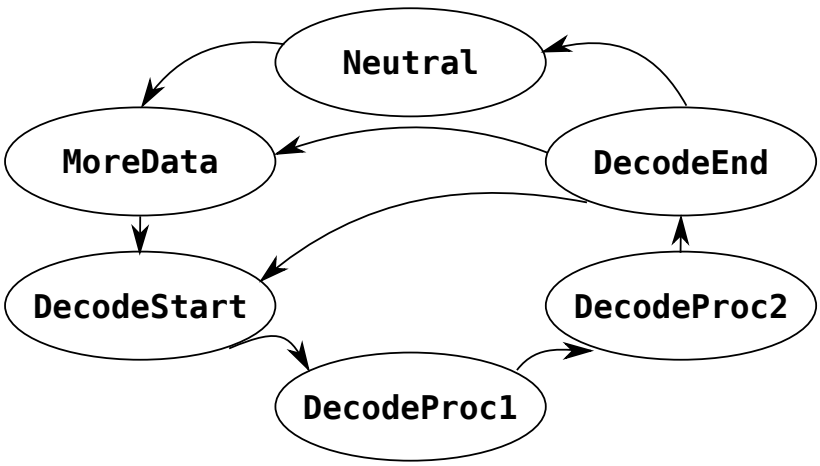


Figure 4.8: Decoder FSM

Neutral is the initial state used to reset internal data and change the state to `MoreData`.

MoreData is responsible for filling the internal data buffer and changing the buffer pointer accordingly. More than 17 bits in the buffer forward the state to `DecodeStart`.

DecodeStart is the first pipeline stage and starts the decoding by writing the internal data buffer into a separate decode buffer.

DecodeProc1 is the second pipeline stage.

DecodeProc2 is the third pipeline stage and marks the data from the decoding as valid.

DecodeEnd is the final pipeline stage. If there are still at least 17 bits in the internal buffer, we continue decoding with `DecodeStart`. Less than 17 bits result in the `MoreData` state, whereas the end of the payload results in `Neutral`.

4.2 Continuous Repeat Request

There are two different kind of frames for CRR: Data frames and ACK frames. The structure and interaction of those two packet types are presented next:

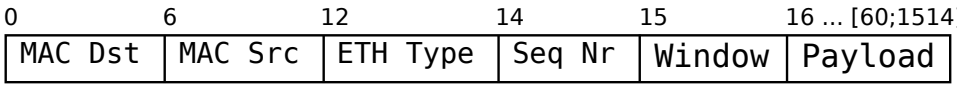


Figure 4.9: Frame and CRR sender format (Data) (length in bytes)

Figure 4.9 shows the frame format sent from a CRR sender module, which contains the following fields:

- **MAC Dst:** Contains the MAC destination address of the receiver module
- **MAC Src:** Contains the MAC source address of the sender module
- **ETH Type:** The Ethernet type field **0xABBA** marks CRR data frames
- **Seq Nr:** Represents the sequence number of the current frame
- **Window:** Specifices the current data transmission window
- **Payload:** The data part of the frame

The Ethernet Type field value **0xABBA** represents CRR module frames. The sequence number is used to keep track of open frames. Frames start with the number `1` and end with the number `2*WIN_SZ`. Once all sequence numbers in this range have been acknowledged, we start with a new transmission and the sequence number `1`. In order to have a similar header structure for sender and receiver module, and to improve the performance, we chose to allow consecutive windows to overlap. This way we only need to wait for missing ACKs after two windows (the maximum sequence number), before starting over again with the first sequence number. As a consequence, a window field is required. The window field contains a value different from the ACK tag, to differentiate between CRR data and ACK frames. It is changed every time we begin a new window and restart the sequence number. This field helps the receiver to detect new windows. Otherwise, it wouldn't know whether the received frame is a retransmission, or belongs to a new window.

Same header length of the two different CRR frames is mandatory, because payloads, with a certain sequence of bytes, could cause a problem otherwise, which would lead to mistaking data frames for ACK frames.

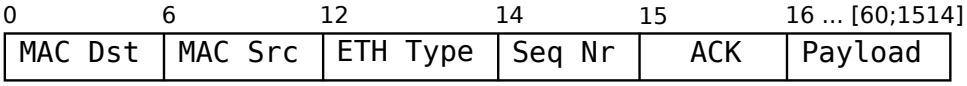


Figure 4.10: Frame and CRR receiver format (ACK) (length in bytes)

Figure 4.10 shows the remaining frame format for CRR receiver modules, which are built in the following way:

- **MAC Dst:** Contains the MAC destination address of the sender module
- **MAC Src:** Contains the MAC source address of the receiver module
- **ETH Type:** The Ethernet ype field **0xABBA** marks CRR ACK frames
- **Seq Nr:** Represents the sequence number of the frame to be ACK'ed
- **ACK:** Is used to mark the frame as an ACK for a certain sequence number
- **Payload:** The data part of the frame

ACK frames have the same structure as data frames, except, that the window field changed. The ACK field carries a value of **0xFF** in order to distinguish between the two different frames. The window field of data frames cannot reach the same value as the ACK field.

4.2.1 Software Module

Data Structures

The private data structure of our CRR sender module can be seen in Listing 4.8. In contrast to the Huffman module, we need to have a single private data structure shared among all CPUs. The reason being, that CRR involves heavy buffering and queueing and each CPU needs to work with the same queues and buffers. Therefore, we used pointers for all CRR specific data structures that point to the same memory location on all CPU cores. This results in a single lock and private data structure which is shared among all CPUs. Next, we present the private data of sender and receiver module:

The same `rwlock_t` structure was used as for the Huffman modules. CRR involves not only read accesses, like the Huffman module did, but also a lot of write accesses. Read and write accesses are mixed up during the program flow which made it more difficult to seperate the code into pure read and pure write sections. To reduce any overhead caused by switching back and forth between **read_lock** and **write_lock**, we chose to protect the whole critical section with a single **write_lock**.

The `mytimer` structure is explained in more detail later on. We have `2*WIN_SZ` sequence numbers and therefore need the same amount of timers. `tx_open_pkts` defines the number of unacknowledged packets that are outstanding, `tx_seq_nr` presents the next sequence number to be used, and `tx_win_nr` the current window number. `bitstream` is responsible to keep track of which sequence numbers have still not been acknowledged. The bit at the first position is used for sequence number 1. The state of the interface between user space and kernel space is expressed through the `wait_active` value. We will talk about this requirement in Section 4.2.1.

There are two different linked lists involved in the CRR sender module: A buffer list and a queue list that use the Linux kernel structure `sk_buff_head`.

`sk_buff_head` is a doubly linked list and its structure members can be seen in Listing 4.9. The buffer list is called `tx_stack_list`, because packets that are sent, are stacked up on top of each other with the oldest packet at the bottom of the stack. ACKs remove the packet from the stack and add newly sent packets on top of the stack. The name is in no way related to the stack data structure, which pushes and pops data on its stack. or the program stack. The `tx_queue_list` is important for packets that can't be sent, because of a full tranmission window, for instance. Those packets are added to the queue and will be processed one by one at a later time.

```
struct fb_crr_tx_priv {
        idp_t port[2];
        seqlock_t lock;
        rwlock_t *tx_lock;
        struct mytimer my_timer[2*WIN_SZ];
        unsigned char *tx_open_pkts;
        unsigned char *tx_seq_nr;
        unsigned char *tx_win_nr;
        unsigned int *bitstream;
        unsigned char *wait_active;
        struct sk_buff_head *tx_stack_list;
        struct sk_buff_head *tx_queue_list;
};
```

Listing 4.8: CRR sender private data structure

```
struct sk_buff_head {
         struct sk_buff  * next;
         struct sk_buff  * prev;

         __u32           qlen;
         spinlock_t      lock;
};
```

Listing 4.9: sk_buff_head kernel structure

The CRR receiver private data structure in Listing 4.10 is very similar to its counterpart. The only difference is the `list`, a linked list used to store out of order packets. Each time a packet with the correct sequence number arrives, we look if the following sequence number has been received before and is stored in this linked list.

```
struct fb_crr_rx_priv {
        idp_t port[2];
        seqlock_t lock;
        rwlock_t *rx_lock;
        unsigned char *rx_win_nr;
        unsigned char *rx_seq_nr;
        unsigned int *rx_bitstream;
        struct sk_buff_head *list;
};
```

Listing 4.10: CRR receiver private data structure

As mentioned before, every sequence number between `1` and `2*WIN_SZ` has its own timer structure presented in Listing 4.11. The pointers are equivalent to the struct members of Listing 4.8. They all point to the same memory addresses. The Linux kernel structure `tasklet_hrtimer` was chosen to handle timeout routines. It is used to initialize a tasklet for a softirq callback.

```
struct mytimer {
        unsigned char *open_pkts;
        unsigned int *bitstream;
        struct tasklet_hrtimer mytimer;
        struct sk_buff_head *stack_list;
        rwlock_t *tx_lock;
};
```

Listing 4.11: CRR sender timer data structure

CRR Functions

The following functions provide the CRR functionality:

- **struct sk_buff *skb_get_nr(unsigned char n, struct sk_buff_head *list)**: Returns the address of the `sk_buff` structure with sequence number `n`
- **enum hrtimer_restart fb_crr_tx_timeout(struct hrtimer *self)**: Packets that timeout invoke this handler which retransmits the packet
- **int notify_fblock_subscribers(struct fblock *us, unsigned long cmd, void *arg)**: Is used to control the throughput between user space and kernel space (LANA)
- **__u32 skb_queue_len(const struct sk_buff_head *list_)**: Returns the length of the queue (Linux Kernel)
- **void skb_queue_tail(struct sk_buff_head *list, struct sk_buff *newsk)**: Inserts an `sk_buff` at the end of the list (Linux Kernel)
- **struct sk_buff *skb_dequeue(struct sk_buff_head *list)**: Removes the head of the list (Linux Kernel)
- **void skb_queue_head(struct sk_buff_head *list, struct sk_buff *newsk)**: Inserts an `sk_buff` at the start of the list (Linux Kernel)
- **struct sk_buff *skb_copy(const struct sk_buff *skb, int priority)**: Makes a copy of an `sk_buff` and its data (Linux Kernel)
- **int process_packet(struct sk_buff *skb, enum path_type dir)**: Schedules an `sk_buff` for immediate processing by the PPE (LANA)
- **void skb_unlink(struct sk_buff *skb)**: Removes an `sk_buff` from the list (Linux Kernel)
- **void skb_insert(struct sk_buff *old, struct sk_buff *newsk)**: Place a packet before a given packet in the list (Linux Kernel)
- **struct sk_buff *skb_get_pos(unsigned char seq, struct sk_buff_head *list)**: Returns the correct position for a frame with sequence number `seq` in the receiver's `list`

CRR Sender

The CRR sender module provides reliable data transmission between nodes. The basic **data transmission** routine is as follows:

All frames with `EGRESS` direction and 0xABBA as `ETH TYPE` that invoke the receive function handler are being sent as CRR data frames. At first, the packet is tagged with the current sequence number which is incremented afterwards for the next packet arriving at the module. Next, the window identifier is written into the header. If the number of open (unacknowledged) packets is smaller than the modules window size, we check the number of packets waiting in the queue. If the queue is empty, we get to send the packet right away but not without buffering it in the `stack_list` first and updating `open_pkts` and `bitstream`. However, if the queue is not empty, we queue our packet at the end of the `queue_list` and dequeue the first packet. This packet is then being sent the same way as in the previous example. A fully used transmission window queues all incoming packets in the `queue_list`. Those packets will be sent at a later time.

There is a special case for frames carrying the sequence number `1`: Sending a packet with the first sequence numbers starts a new transmission window, which should not overlap with the old one. Most CRR implementations don't allow consecutive windows to overlap. This means that all packets within the window need to be acknowledged first, before starting a new window. Our implementation allows two consecutive windows (big window) to overlap with each other, but not more than two. Therefore, before starting a new window with sequence number 1, the previous two windows must have been acknowledged. The advantage of this little tweak is a trade off between performance and memory. The sender needs to wait for all ACKs to arrive only once per `2*WIN_SZ` packets, instead of twice, but the receiver needs to have enough memory to store `2*WIN_SZ-1` packets, instead of `WIN_SZ-1`. As a consequence, before a packet with sequence number `1` is sent, we need to make sure, that all ACKs of the previous two windows have been received. If not, the packet is added to the queue.

Each packet, leaving the sender module, starts a timer, which stops when the corresponding ACK has been received. If the ACK does not receive within time, the time out of the timer will call its function handler. The **time-out** function **fb_crr_tx_timeout** is invoked with a `self` pointer of the `hrtimer` structure seen in Listing 4.11. Through the **container_of** Linux kernel function we are able to access our `mytimer` struct. The `sk_buff` at the head of the `stack_list` is dequeued, copied and resent again. The original is inserted at the end of the list again, because it will be the last sent packet now. Before returning from the function, the appropriate timer is restarted.

The remaining subfunctionality of the sender module is the **ACK reception**: Packets on the `INGRESS` path with `ETH TYPE` 0xABBA, a correct sequence number, and the ACK field set to 0xFF, are treated as ACKs. First of all, the corresponding timer of this sequence number is stopped. Next, **skb_get_nr** returns the `sk_buff`'s address that carries the sequence number which has been acknowledged. It is no longer necessary to buffer this packet, hence it is unlinked, freed and its `bitstream` bit is unset. The `open_pkts` value is decremented by one as well, which gives opportunity to a new packet to be sent. From here things are exactly the same as in the **transmission** part above. In the next section we present the program flow of the receiver module.

CRR Receiver

The receiver's responsibility is forwarding packets in the correct order and the transmission of ACKs for each successfully received CRR data packet: The **forwarding** routine reads the incoming packet's window and compares it with the window of the previously received frame. If they match, it means that they belong to the same big window. A different window on the other hand, means that the sender module has started a new window with sequence number `1`. Therefore, all packets from the previous window had been successfully acknowledged. As a consequence, the `bitstream` value is reset to zero, which states that no packet from this new window has been received yet. Next, the receiver module looks at the sequence number and compares it with its own expected sequence number. If they match, the packet is forwarded, the `rx_seq_nr` incremented and the corresponding bit in `bitstream` set. It may be possible that some higher sequence numbers have arrived before and were stored in the `list`. Therefore, we search this list for consecutive sequence numbers and forward them as well. Each forwarded frame increments `rx_seq_nr`. Out of order frames, on the other hand, are stored in the `list` and remain there until it is their turn to be forwarded.

The receiver module sends an **ACK** for each successfully received frame. Therefore, we copy the received frame and swap MAC addresses. Afterwards, we mark

the packet as an ACK and let the PPE process it. The receiver module does not make a difference between packets carrying the expected sequence number, out of order packets, or previously received packets. An ACK is sent either way, because we don't know for sure if the ACK has been received by the sender side, or if it has been lost.

Optimizations

We start this section with two problem descriptions which were two major challenges when implementing the CRR modules:

Each **send(2)** call in a user space application sends a packet via the `AF_LANA` socket directly into Linux kernel space. Now imagine an application that keeps sending packets: The first couple of packets are sent directly, because the transmission window offers free packet slots. After a while the transmission window fills up and the first number of packets are inserted in the queue. There will be more packets arriving in kernel space than leaving it, so the queue will grow steadily. The queue can only hold a certain number of packets, before we run out of memory. It is crucial to introduce an upper bound for the queue length.

We came up with the following solution: The CRR sender module keeps track of it's queue's length at all times. In the event of a full queue, we need to block the connection between user space and kernel space. The code snippet of Listing 4.12 shows the idea:

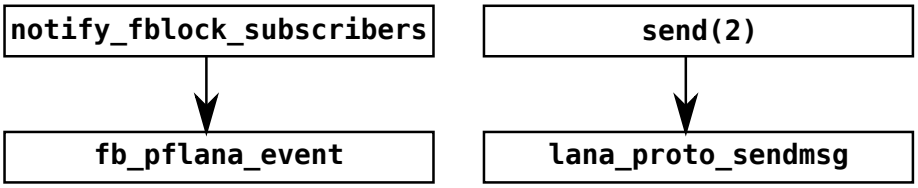
```
if (queue_len == MAX_QUEUE_LEN && *fb_priv_cpu->p->wait_active == 0) {
  notify_fblock_subscribers((struct fblock *)fb, FBLOCK_SRC_WAIT, NULL);
  *fb_priv_cpu->p->wait_active = 1;
  //printk(KERN_ERR "[TX]\tWAIT ACTIVATED!\n");
}
else if (queue_len == MIN_QUEUE_LEN && *fb_priv_cpu->p->wait_active == 1) {
  notify_fblock_subscribers((struct fblock *)fb, FBLOCK_SRC_WAIT_DONE, NULL);
  *fb_priv_cpu->p->wait_active = 0;
  //printk(KERN_ERR "[TX]\tWAIT DEACTIVATED!\n");
}
```

Listing 4.12: Blocking and unblocking incoming packets

When the queue length reaches the maximum length and `wait_active` has still not been set, we call the **notify_fblock_subscribers** function which calls all FB subscribed to our block. This results in the function **fb_pflana_event** being evoked. Figure 4.11(a) pictures this connection. `FBLOCK_SRC_WAIT` sets a private struct member of the FB `fb_pflana` named `spinwait`, whereas `FBLOCK_SRC_WAIT_DONE` unsets it.

lana_proto_sendmsg evaluates the value of `spinwait` for each packet that is sent from user space. As we can see from Figure 4.11(b) **lana_proto_sendmsg** is invoked through **send(2)**. Whenever `spinwait` is set, the packet is freed and **lana_proto_sendmsg** returns with an error message telling the **send(2)** routine that the packet could not be transmitted. The user space application needs to be ready for this and either verify or retry transmission. Now the connection between user space and kernel space is blocked, which results in an emptying queue. When the queue length reaches a certain level called `MIN_QUEUE` in Listing 4.12, `spinwait` is unset again and new packets are transfered to kernel space.

We faced the same problem for the opposite direction as well: Packets coming from kernel space to user space are written to socket buffers. If the user space application doesn't read packets from the socket buffer fast enough, the socket buffer will eventually fill up and discard incoming packets. To avoid this problem the socket buffer size may need to be increased [56]. The socket buffer's size should be proportional to `MTU*2*WIN_SZ`, which is the maximal amount of data two consecutive windows can carry.



(a) Notification hierarchy (b) Send hierarchy

Figure 4.11: Hierarchy of involved functions

Proper locking technique and branch prediction optimizations that were already presented in Section 4.1.1 were reused for the CRR modules as well. We tried to reuse existing Linux kernel functions and data structures, because on one hand they made life a little easier during implementation, and on the other hand they are optimized and perform well.

ACK frames were chosen as small as possible to not unnecessarily waste bandwidth. Data flow control protocols don't offer much freedom and flexibility for optimizations. There is no parallelism to exploit, a rather long critical section and plenty of memory accesses. The Huffman modules benefit much more when using multicore systems, than CRR, because they can process packets in parallel, whereas for CRR one core in the critical section, blocks all other cores from entering.

4.2.2 Hardware Module

Data Structures

The data structures in VHDL are closely related to the private data structures from the software modules this time. The reason being, that CRR, unlike Huffman, is all about control flow with many different branches and states. Computational efforts are negligible, so CRR sender and receiver module resemble one huge FSM. The records of sender and receiver module are presented in Listing 4.13. Most record members are analogous to the software data structures and should be self-explanatory. However, `nextstate` from `CRR_RX_STATUS` is used to store a future state. We will elaborate on this future state, when talking about the FSMs. `store` indicates that an out of order packet arrived and needs to be stored. Values different from `0` for `store` define the sequence number and the address at which the packet will be stored.

```
type CRR_TX_STATUS is record
        bitstream : std_logic_vector(2*WIN_SZ-1 downto 0);
        window    : std_logic_vector(7 downto 0);
        nextseq           : std_logic_vector(7 downto 0);
        openpkts  : integer range 0 to MAX_SEQ_NR;
end record;

type CRR_RX_STATUS is record
        bitstream : std_logic_vector(2*WIN_SZ-1 downto 0);
        oldwindow : std_logic_vector(7 downto 0);
        nextseq           : std_logic_vector(7 downto 0);
        nextstate : crr_rx_type;
        store             : integer range 0 to MAX_SEQ_NR;
end record;
```

Listing 4.13: CRR sender record

There is a separate ACK record for the CRR receiver module, that stores the MAC source and destination address as well as the sequence number. The data

record structure is required for the ACK process.

```
type ACK_HDR is record
    macsrc        : MAC_ADDR;
    macdst        : MAC_ADDR;
    seqnumber : std_logic_vector(7 downto 0);
end record;
```

Listing 4.14: CRR ACK record

Interfaces

The same interfaces to ReconOS and NoC are shared like in Section 4.1.2. There is a difference in the number of interfaces, which we have seen earlier in Figure 3.5. The **sender module**, for instance, requires an `EGRESS` input and output path, as well as an `INGRESS` input path. The reason being that ACK and data frames require different paths is because otherwise potential deadlocks could occur when waiting for ACK frames that are stored somewhere in the queue between data frames.

The **receiver module**, on the other hand, has `INGRESS` input and output path as well as an `EGRESS` output path.

CRR sender module processes

The following processes contribute to the sender's functionality:

- **SYNC_PROC**: Updates registers, states and records
- **SHIFT_REG_PROC**: A shift register process for the CRR header
- **BRAMAddrCnt**: Addressing counter for storing and reading packets from the BRAM
- **HeaderCnt**: Keeps track of the CRR header in the shift register
- **RESENDCnt**: One of the generated time-out counters
- **ResendSeq**: Detects packet time-outs
- **NEXT_STATE_DECODE**: Controls the module's FSM
- **OUTPUT_DECODE**: Controls the modules internal state, BRAM access and written output data

The sender contains 3 subroutines, namely receiving ACKs, retransmitting old packets and transmitting new packets. Those three routines get executed in this particular order. Receiving ACKs has the highest priority, because it may prevent packets from timing out and saves potential retransmissions. Packets that time out need to be retransmitted as fast as possible, because unacknowledged packets may block our module from sending further packets when restarting with sequence number `1`, for instance. Finally, the transmission of new packets is scheduled last.

For the most part, the CRR sender module follows the software approach and accesses similar data structures. A timer in hardware is a simple counter that, when reaching a certain value, triggers an event. In our case there are `2*WIN_SZ` counters in parallel that are initialized to `0`. When a new packet is sent, `bitstream` is updated, accordingly. Each of those counter processes examines the value of its bit position in `bitstream`. If the bit is set to one, the process' counter value is increment by one on each clock cycle. When the counter reaches a certain time-out value, a retransmission of the packet is scheduled. The processes **RESENDCnt** and **ResendSeq** are responsible for the timer functionality and to detect time-outs.

RESENDCnt is the counter process and **ResendSeq** schedules retransmissions. If more than one packet has timed out, the packet with the smallest sequence number, namely the oldest packet, is retransmitted first.

Transferring and storing packets happen at the same time and minimize delay when sending CRR data frames.

CRR receiver module processes

Processes responsible for the receiver's functionality are listed below:

- **SYNC_PROC**: Updates registers, states and records
- **SHIFT_REG_PROC**: A shift register process for the CRR header
- **AckCnt**: This counter controls the transmission of ACK frames
- **BRAMAddrCnt**: Addressing counter for storing and reading packets from the BRAM
- **HeaderCnt**: Keeps track of the CRR header in the shift register
- **ProcShiftCnt**: Another counter used to process the CRR header
- **ACK_PROC**: Builds the ACK frames based on the ACK record structure
- **NEXT_STATE_DECODE**: Controls the module's FSM
- **OUTPUT_DECODE**: Controls the modules internal state and BRAM access

Most processes are self-explanatory. We give a more thorough explanation of the interaction between processes when introducing the state machines involved in the next section.

Finite State Machines

Figure 4.12 and Figure 4.13 show the state machines responsible for correct operation of our CRR modules.

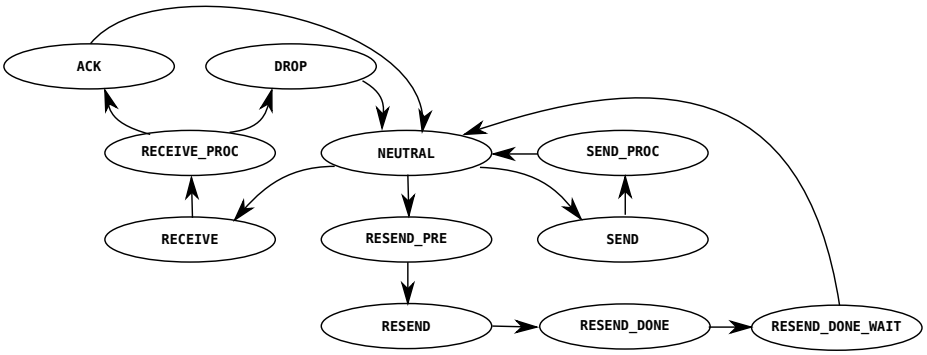


Figure 4.12: CRR sender FSM

NEUTRAL is the starting point of the CRR sender FSM. Next state is either `RECEIVE` if there are packets to be received, `RESEND_PRE` if a packet timed out and needs to be retransmitted, or `SEND` in case there are packets to be sent.

RECEIVE resets the counter of the process **HeaderCnt** and moves to `RECEIVE_PROC` in case of a new frame waiting to be read.

RECEIVE_PROC waits for the complete header to show up in the shift register and decides which state to go next depending on the header fields. Invalid `ETH TYPE`, sequence number, or non ACK packets result in the packet being dropped in the following `DROP` state, otherwise the FSM continues with the `ACK` state. `bitstream` and `openpkts` are also adjusted in this state.

ACK reads the remaining part of the frame and returns to the `NEUTRAL` state in the end.

DROP reads and clear the packet from the queue and returns to the `NEUTRAL` state afterwards.

RESEND_PRE is a transit state responsible for setting up the BRAM address.

RESEND increments the address and retransmits the packet stored in the BRAM. Changes its state to `RESEND_DONE` after writing the last byte.

RESEND_DONE resets the counter of the timed out packet and advances to `RESEND_DONE_WAIT`.

RESEND_DONE_WAIT is a synchronization state and changes the state back to `NEUTRAL`.

SEND awaits the start of an incoming frame and changes the window number every time sequence number `1` is sent. Starts sending the packet and storing it in BRAM. `SEND_PROC` will be the state that follows.

SEND_PROC is updating `bitstream`, `nextseq` and `openpkts` after the end of file byte has been detected. Afterwards we start over with `NEUTRAL` state.

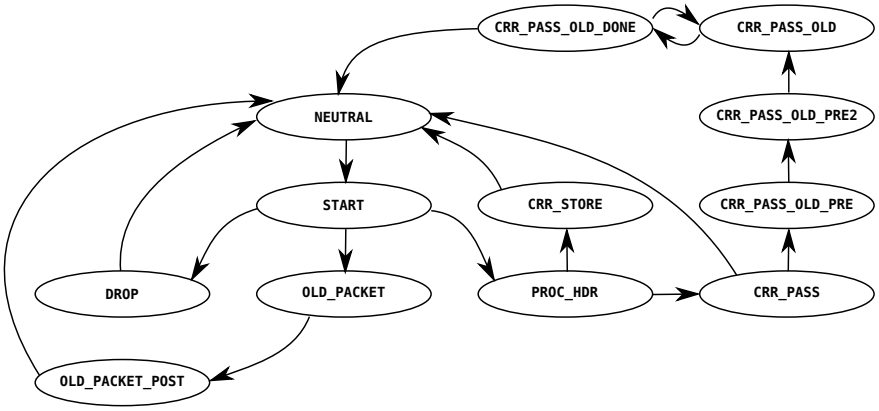


Figure 4.13: CRR receiver FSM

NEUTRAL resets the internal data structures such as `store` and `nexstate`. Arriving packets change the state to `START`.

START reads the packet header and extracts sequence number, `ETH TYPE` and the window. Further, it stores the MAC addresses for potential ACK frames that need to be sent later. Packets with invalid `ETH TYPE` or sequence number are

dropped in the DROP state. Other possibilities are old packets that result in a state change to OLD_PACKET, out of order packets move on to PROC_HDR and set nextstate to CRR_STORE for later, or the expected sequence number which also continues with PROC_HDR and sets nextstate to pass.

DROP Awaits the end of the packet and continues with NEUTRAL.

OLD_PACKET reads and clears the old packet from the FIFO.

OLD_PACKET_POST waits for ACK to finish sending and starts a new reception round with NEUTRAL.

PROC_HDR waits for the complete header to be processed, which means either to be stored or forwarded. Afterwards we change the state to the value of nextstate which was set in the START state.

CRR_STORE continues storing the packet until the end of file byte has been written to the BRAM. Changes state to NEUTRAL after that.

CRR_PASS forwards the packet to the next FB. There are two possibilities upon the end of file event: The following sequence number has been received before and is stored in the BRAM, which results in CRR_PASS_OLD_PRE, or otherwise we jump back to NEUTRAL state.

CRR_PASS_OLD_PRE is a transit state to initialize some internal data.

CRR_PASS_OLD_PRE2 is the second transit state.

CRR_PASS_OLD starts reading the BRAM and forwarding the packet to the next FB. The end of file flag causes a state switch to CRR_PASS_OLD_DONE and an update of nextseq.

CRR_PASS_OLD_DONE checks if consecutive packets are stored in the BRAM as well (CRR_PASS_OLD). If not, the next state is changed to NEUTRAL.

4.3 State Transition

We described the order of a state transition in Section 3.3 before. Now we will talk about implementation specific details: The state transition mechanism enables the migration of tasks running in software to hardware or vice versa. Modules with internal protocol state need to transfer this state to their software or hardware counterpart, which continues operation after initializing the internal state. Stateless protocols, on the other hand, don't need to collect and set the state before changing the task execution from software to hardware. In order to transfer the protocol state, a specific data structure is needed. To this end we created a data structure, which can be used for all possible data structures and FBs.

The data structure can be seen in Figure 4.14. All existing data structures in C have data sizes in multiple of bytes. Therefore, we can express every possible data structure with the number of elements and the number of bytes for each element, including padding elements. We chose to address 32-bit integer values instead of bytes and introduced byte padding for unused memory space. This simplified the state machine in hardware later on. The first two bytes at address 0 are used to describe the number of elements. The following two bytes are padded. Afterwards the data description starts with length of the data in bytes and the corresponding data follows. This scheme is repeated for each element.

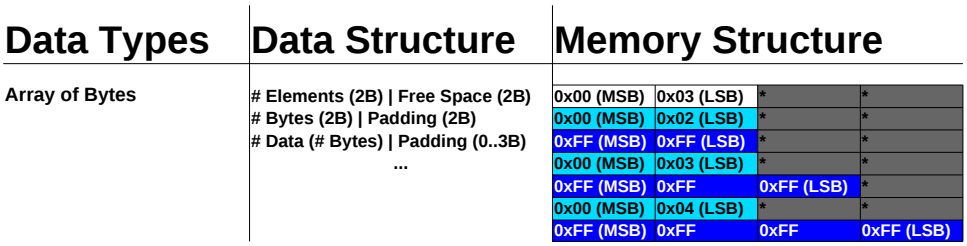


Figure 4.14: Protocol state data structure

4.3.1 Software Module

The state transition mechanism is a LANA extension that provides the needed functionality for **getStateSW()** and **setStateSW()** calls. The basic idea behind it was to develop a single miscellaneous device driver, which is responsible for such operations on all functional blocks. Reading from the device represents **getStateSW()** and writing it results in **setStateSW()**. The extensions are located in the file `xt_conf.c` and include the following functions [17]:

- **long ei_conf_ioctl(struct file *file, unsigned int cmd, unsigned long arg)**: Is used to change the underlying configuration device parameter, for instance the name of the functional block. Returns `0` when successful
- **ssize_t ei_conf_read(struct file *file, char __user *buff, size_t len, loff_t *ignore)**: Calls a specific function on the FB, which was configured on the device to write its internal data to a buffer. Returns the data length when successful
- **ssize_t ei_conf_write(struct file *file, const char __user *buff, size_t len, loff_t *ignore)**: Calls another specific function on the FB that extracts its internal data from a buffer. Returns the data lenght when successful

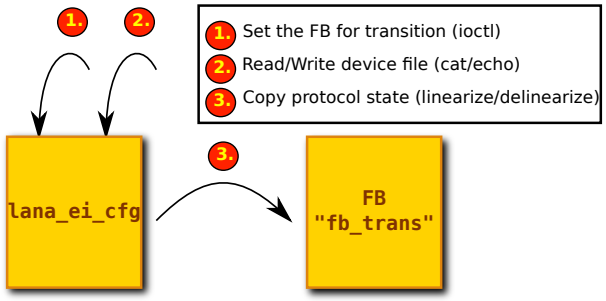


Figure 4.15: State transition in software

Figure 4.15 shows how the state transition is implemented in software: The first step is to tell the `lana_ei_cfg` device the name of the functional block that should write or read its internal data. Therefore we need to open the device file from user space and access its file descriptor using **ioctl(2)** [57] with a device driver specific message and the name of the FB, "fb_trans" in our example.

Next, we either read or write to the device file, depending if we set or get the state. As a result, one of the above mentioned functions will be called. Internally, the read and write functions first need to figure out the address of the earlier configured functional block. With the extension of the state transition mechanism, two

new functional block callback handler were introduced and added to the functional block data structure. Those two functions are called **linearize** and **delinearize**. **linearize**'s responsibility is to copy the FB's internal state and data to a buffer, whereas **delinearize** copies the buffer and sets its internal state and data accordingly.

ei_conf_read allocates memory, calls the **linearize** function of the FB and copies the retrieved protocol state data back to user space. **ei_conf_write** behaviour is analogous except that data is copied from user space to kernel space. Note that it is the developer's task to change **linearize** and **delinzearize** functions to extract or create the earlier mentioned data structure. The exact internal data structure is unique for each FB.

4.3.2 Hardware Module

A hardware implementation of the state transition mechanism for CRR sender module was implemented. The following examples and discussion should introduce the basic operation, but in the end, it is the developer's task to tailor the example FSM to his specific needs. The existing design should be a good starting point for future FBs.

Data Structures

Listing 4.15 shows the record structure that was used for the state transfer:

```
type TRANSITION_STATUS is record
        addresscounter : integer range 0 to C_LOCAL_RAM_SIZE;
        elemcounter : integer range 0 to 65535;
        bitstream : std_logic_vector(2*WIN_SZ-1 downto 0);
        len_index        : integer range 0 to C_LOCAL_RAM_SIZE;
        pktindex : integer range 0 to MAX_SEQ_NR;
        pktbuffer  : std_logic_vector(31 downto 0);
        buffercounter : integer range 0 to 1600;
end record;
```

Listing 4.15: State transition record for CRR

The dual port RAM (DPR) of every hardware FB that is connected to ReconOS' operating system and memory interface is the starting point of our module. Other RAMs belonging to our FB that store protocol specific data need to be updated to DPR, because we need to be able to read and write to all involved RAMs from two different processes at the same time. The `addresscounter` is used to read or write from and to the ReconOS DPR. The total number of elements is stored in `elemcounter`. The `bitstream` field is used as a copy of the original bitstream and is updated every time a packet has been processed and stored in the ReconOS DPR. It acts like a packet copy schedule. All length values from the data structure are stored in `length_index` in order to read the correct amount of bytes for each data structure member. The current packet that is processed is written to `pktindex`, which is responsible to write packets at the correct position in the CRR DPR. ReconOS's DPR's data width is 32 bit and the DPR to store CRR packets has a data width of 9 bits. The `pktbuffer` is used to buffer data while we write it bytewise to the CRR DPR or from CRR DPR to ReconOS DPR in chunks of 32-bit. Finally, `buffercounter`'s responsibility is the addressing for read and write accesses to CRR DPR.

Finite State Machines

The state transition mechanism introduces an additional FSM, responsible for the extraction and initialization of data, as an extension to the ReconOS FSM. It's called an extension, because it may just add those states discussed later to a preexisting

FSM as an extension. We will begin with the extension to the ReconOS FSM that is pictured in Figure 4.16:

GET_OP is the intial state of the ReconOS FSM. In this state the operation is determined, that could result either in `GET_ADDR`, or `SET_ADDR`.

GET_ADDR receives the memory location at which we store our protocol state later.

PROC_GET_STATE is the processing state of the **getStateHW()** call and will lead to `STATE_WRITE` when finished.

STATE_WRITE writes the content of the ReconOS DPR to the memory location specified earlier and moves to `STATE_ACK`.

SET_ADDR receives the memory location from which we need to read our data block structure later.

STATE_READ copies data from the previously given memory location to our ReconOS DRP.

PROC_SET_STATE is the processing state of the **setStateHW()** call and leads to `STATE_ACK` later.

STATE_ACK is the final state of the ReconOS FSM and is responsible for feedback delivery after the command has been executed successfully.

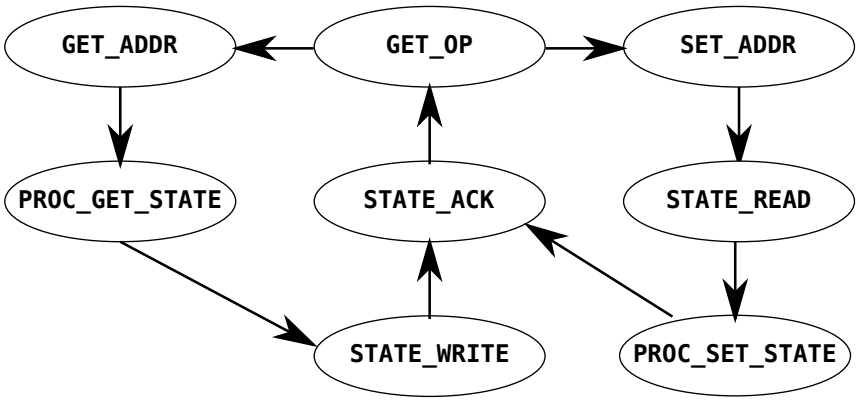


Figure 4.16: ReconOS state transition FSM

The remaining FSM reads the FB's internal state and creates a data block structure as seen in Figure 4.14, or reads such a data block structure and initializes internal record data structures and memories accordingly. The CRR state transition FSM can be seen in Figure 4.17 and is presented next:

NEUTRAL is the initial state and resets internal data records. Next state is either `GET_PROC_HDR` in case of a **getStateHW()** call, or `SET_PROC_HDR` when **setStateHW()** was detected.

SET_PROC_HDR extracts the number of elements and writes the value to `elemcounter`.

SET_PROC_BITSTREAM extracts the length of the bitstream value and stores it to `bitstream` in the next cycle.

SET_PROC_OPENPKTS does the same as the `SET_PROC_BITSTREAM` state but for openpkts (internal record) that are stored in `pktbuffer` temporarily.

SET_PROC_WINDOW does the same like `SET_PROC_OPENPKTS` for window (internal record).

SET_PROC_NEXTSEQ does the same like `SET_PROC_OPENPKTS` for nextseq (internal record).

SET_PROC_RECORD initializes the internal data record structure for CRR with the values written to `pktbuffer` register. Unacknowledged packets change the state to `SET_PROC_LEN`, whereas no outstanding ACKS results in the state `DONE`.

SET_PROC_PKT_LEN extracts the packet's length.

SET_PROC_PKT_START acts as a transition state before regular processing starts.

SET_PROC_PKT writes the `pktbuffer` bytewise to the CRR DPR and also controls the continuous reading from the ReconOS DPR. When `buffercounter` reaches `len_index-2` we change to `SET_PROC_PKT_END`.

SET_PROC_PKT_END sets the end of file bit at the beginning of the CRR DPR data signal to indicate the end of the packet.

GET_PROC_HDR writes the number of elements in the first 16-bits of the ReconOS DPR.

GET_PROC_BITSTREAM stores length and value of CRR's internal record member bitstream.

GET_PROC_OPENPKTS stores length and value of CRR's internal record member openpkts.

GET_PROC_WINDOW stores length and value of CRR's internal record member window.

GET_PROC_NEXTSEQ stores length and value of CRR's internal record member nextseq. The next state depends on the fact if any packets need to be stored. If so, the packet with the lowest sequence number is processed first in `GET_PROC_PKT`. If there are no packets to store, we continue with `DONE`.

GET_PROC_PKT collects data from the CRR DPR in our buffer and writes the data in 32-bit chunks to the ReconOS DPR. The state changes to `GET_PROC_PKT_LEN`, when the end of file has been detected.

GET_PROC_PKT_END writes the buffered data to the ReconOS DPR and pads unused bits with `0`.

GET_PROC_PKT_LEN writes the length of the processed packet to the correct address. If there are no other packets left to be processed and stored in the ReconOS DPR anymore, we jump to the final state `DONE`. Otherwise we identify the next packet in line to be stored in ReconOS DPR, reset some internal state transition record members and return to `GET_PROC_PKT`.

DONE communicates with the ReconOS FSM and signals that the processing has come to an end. The ReconOS FSM will either move on with `STATE_WRITE` when it was a **getStateHW()** call, or with `ACK` when it was **setStateHW()**, to signal that the processing finished.

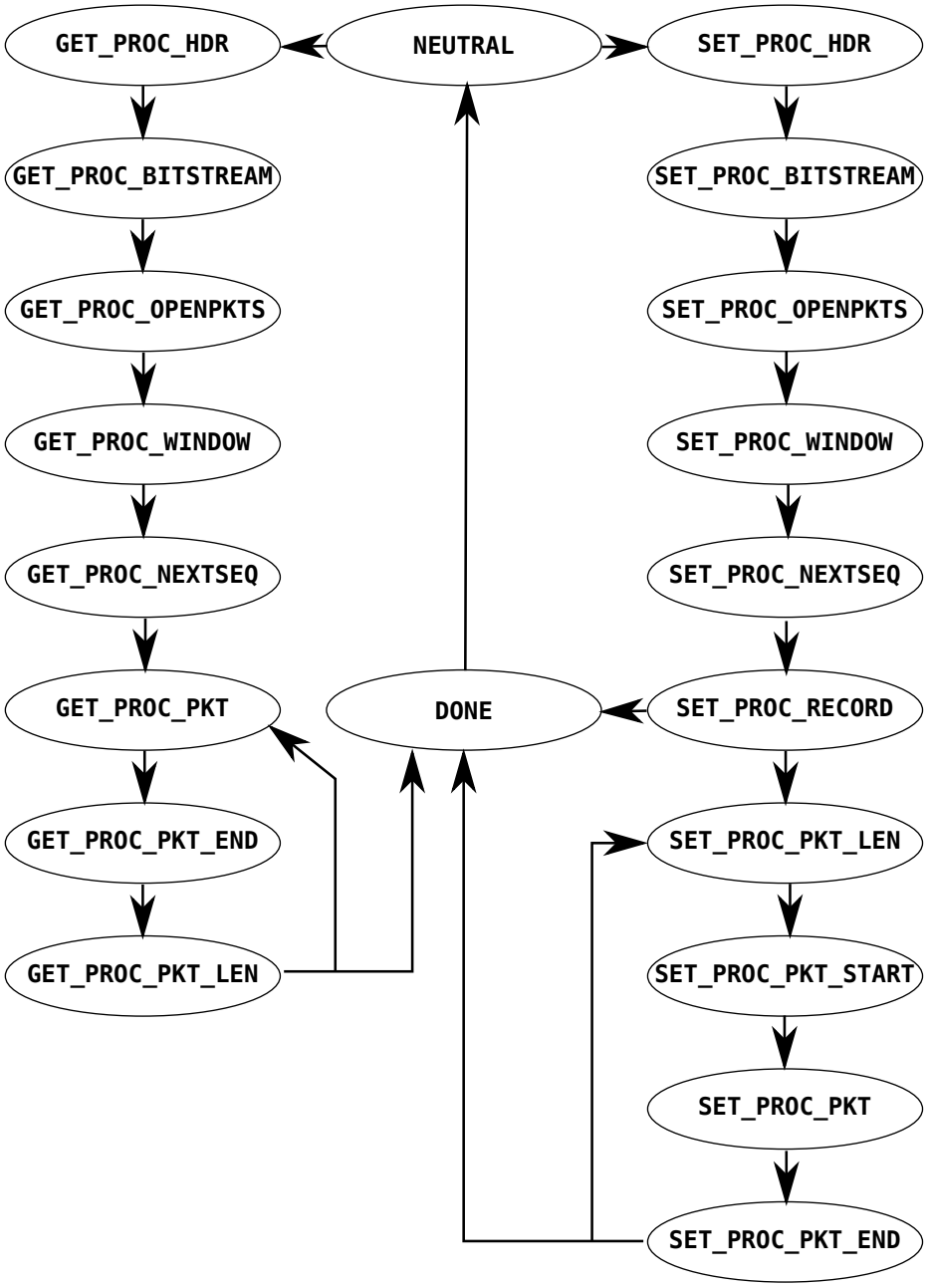


Figure 4.17: State transition FSM

Chapter 5

Evaluation

In this chapter, we elaborate on the methodology of functional verification and present a performance evaluation of our implemented functional blocks. This chapter is structured in hardware and software sections for Huffman and CRR modules.

5.1 Testing Platform

We used two machines with exactly the same hardware and software components to run performance benchmarks and functional tests. Those systems include the following main components:

- **CPU:** Intel Core 2 Quad, Q6600, 2.4GHz
- **RAM:** 4 GB
- **NIC:** Intel 82566DC-2 Gigabit Ethernet Controller
- **OS:** Linux Version 3.0.0, Debian 4.4.5-8

We ran all software benchmark on a transmitter-receiver setup that was connected directly through Ethernet.

5.2 Testing Methodology

The performance values measured in this section represent the statistical median of a series of measurements. Packet sizes were chosen as proposed in RFC2544 [47], except that the maximum transmission unit was set to 1500 bytes. Complexity tests in software and hardware were measured in terms of cycles. The advantage of cycles as the unit of time are various:

- Smallest and most accurate unit of time to measure performance and complexity
- Independent of processor frequency and should yield comparable results for different processors of the same architecture
- Suitable to measure software execution time in Linux kernel space and hardware execution time in simulation

This reasons justify cycles as the unit of time when comparing software and hardware modules. The other chosen unit was packets per second to express and measure the throughput.

Hardware benchmarks run in simulation only need a single measurement, because the result is exact and deterministic. Software benchmarks were tested in a series of at least 6 measurements with a time duration of 10 seconds to compute the median. This testing approach gave us an accurate number for most measurements and the duration of at least 1 minute compensated for irregularities. Strong variations during measurement resulted in increasing the measurement series to obtain more accurate results. The exact testing process will be explained in the appropriate chapter, because depending on the protocol, the involved tools and setup changed.

5.3 Huffman Coding

What follows is the presentation of our testbench and benchmark setups and their results. We start the discussion with an examination of software modules and continue with hardware implementations afterwards. Finally, we put the results in relation to each other and discuss the speed-up and limiting factors.

5.3.1 Software Module

Compression and encryption protocols are challenging to verify and debug. When things go wrong we end up debugging encoding and decoding on bit-level. Encoded data does not contain a lot of information that can be used to conclude what went wrong when things fail. One approach is to use data that is encoded into repeating patterns. Once encoder and decoder work as single units they can be connected which simplifies the debugging process immensely. The output should be exactly the same as the input in an encoder-decoder setup. This was the approach for our functional verification experiments in hardware and software.

Functional Verification

When designing and implementing the Huffman coding algorithm for the software module, we wrote a user space application first in order to do the testing. Debugging and running tests in a user space application has several advantages: Dereferencing null pointers, or segmentation faults end up in program failures instead of kernel crashes. As a consequence, it is easier to follow the program flow and analyze program paths and diagnose outputs with the help of **printf(3)** calls, for instance. There are kernel debuggers such as `KGDB` [58], but those are controversially discussed in the Linux kernel hacker community [59]. Therefore, we decided to use `gdb` in user space.

Finally, the testing had to be done in kernel space as a LANA FB module. The setup can be seen in Figure 5.1.

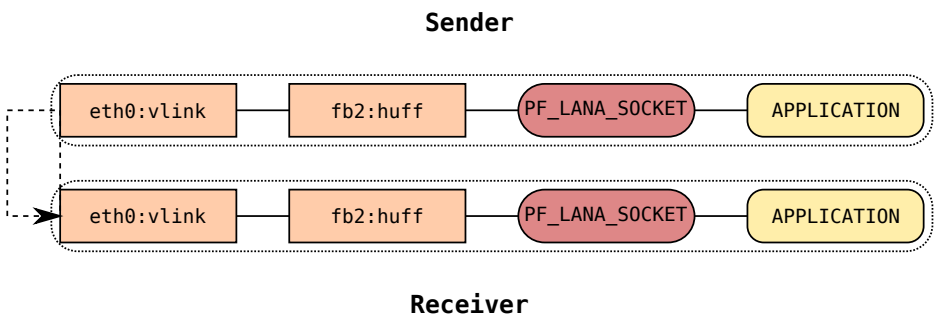


Figure 5.1: Huffman setup

The basic setup consists of a sender and receiver node that have both loaded the LANA framework in kernel space. Additionally, there is a Huffman coding module linked between Ethernet and `PF_LANA` sockets. The socket was opened by a user space application and is responsible for forwarding packets between user space and kernel space. The application on the sender node creates packets and sends them to the socket. The socket receives those, changes the IDP addresses in the control block field of the `sk_buff` structure and forwards the packet to the PPE. The PPE sends the packet to the Huffman coding block, which encodes the payload, updates IDP addresses and returns the packet to the PPE. Eventually, the packet is being sent to the Ethernet interface that creates a frame and sends it to the receiver node. When the packet arrives at the receiver, it is forwarded to the Huffman coding module that decodes the payload data. If everything worked out as expected, the packet should carry its original data again when leaving the module. The user space on the receiver side does not notice an effect of the encoding and decoding that took place between sender and receiver user space.

To generate a realistic letter frequency distribution we chose the book *The Count of Monte Christo* written by Alexandre Dumas from the `Project Gutenberg` [60]. `Project Gutenberg` offers downloads to various books in `UTF-8` format. We wrote a small Python script that opens the book file and scans the text couting different letters. In the end the distribution is printed in sorted order. This alphabet was later used as default alphabet for our Huffman coding modules in software.

Which experiment would be better suited than the encoding and decoding of the book, which was used to generate our alphabet? As a positive side effect, we were able to measure the compression ratio which could be achieved with an optimized code book:

We wrote a user space application that opens the file of *The Count of Monte Christo*, opens an `AF_LANA` socket, creates raw packets with a payload size of `1260` and sends the packets to a given receiver. The packet's payload is filled with data from the book. Those packets are encodeded, sent and decoded again in kernel space of the receiver. On the receiving side runs another application that receives packets from the previously opened `AF_LANA` socket and writes the payload in a file.

We connected an additional counter module to the LANA communication stack on the receiver side seen in Figure 5.1. Then, we ran the experiment twice: Once for a plain protocol stack without Huffman module between socket and vlink, and once for a protocol stack including Huffman coding. From the counter module, we extracted the number of packets and bytes that were received on the receiver side. For the Huffman protocol stack, we compared the received file against the original file with **vimdiff(1)**. The two files were exactly the same which verifies correct functionality of the encoder and decoder. Table 5.1 shows the extracted numbers for compressed and uncompressed transmission.

	Compressed	**Uncompressed**
# Pkts	2551	2551
# MB	1.42	2.5

Table 5.1: *The Count of Monte Christo* transmission

$$\textit{Compression Ratio} = \frac{\textit{Compressed Size}}{\textit{Uncompressed Size}} = \frac{1.42}{2.5} = 0.568$$

Performance Evaluation

In our first benchmark, we wanted to compare the maximal throughput of the Huffman coding module against a simple FB counter in LANA. Therefore, we connected

two machines with each other: A traffic source and a traffic sink. The traffic source generated encoded packets through **trafgen** [61], which is a high-performance zero-copy network traffic generator. The pre-encoded packets that need to be sent were stored in a file. We chose this setup, because it offers a higher throughput and more flexibility than other tools such as **packETH** [62].

On the receiver side, we had two different setups for this test: Our first setup built a LANA protocol stack similar to Figure 5.1 except that there was a FB counter connected to vlink, which counts incoming packets and their size. The second setup was the Huffman setup mentioned before with a FB counter after the Huffman coding block. This way, we were able to measure the maximal throughput of our system. The decoding process is much more complex than the encoding. Therefore, our system is limited by the throughput which can be achieved through decoding.

This setup was tested for the different cases: Huffman codes have variable lengths: We compared longest code lengths, average code lengths and shortest code lengths. It should be mentioned here that the average code length was chosen to be 1 byte. The reason being that likely characters have a length of less than 1 byte, whereas unlikely characters have a length of more than 1 byte. All characters with a code length of 8-bit need the same amount of memory in ASCII or Huffman representation. Before encoding takes place, each character is 1 byte long. Afterwards a character may have a length between 3 and 17 bits, which needs to be taken into account when testing performance. To be fair when compairing the effort and complexity, all following packet sizes refer to the original payload data.

The measurement data was read from the FB counter via the `proc filesystem` (**cat(1)**) from user space. Through **watch(1)** we read the counter value every 10 seconds and noted the value. We collected 6-7 measurements and computed the median from our measurement series, as described in Section 5.2. The results are presented in Figure 5.2.

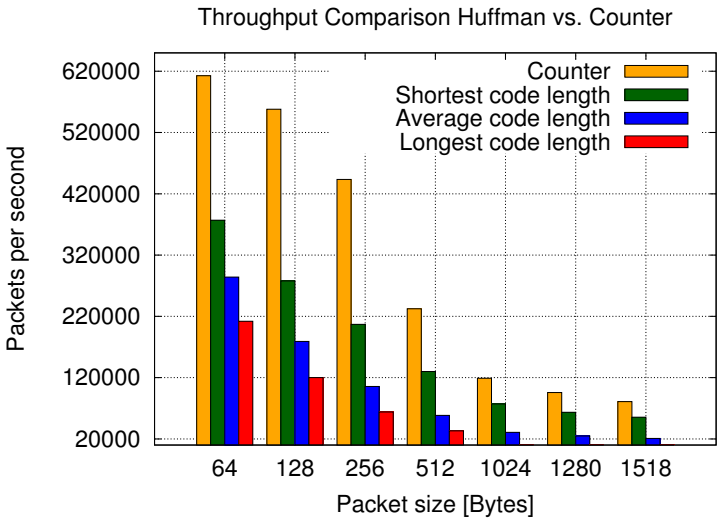
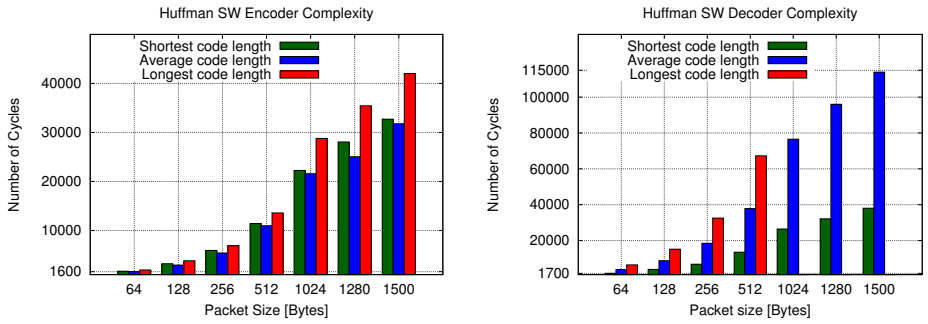


Figure 5.2: Huffman module throughput

The reason why values for packet sizes larger than 512 in the worst case disappear, is that each original character is mapped to a 17-bit code. This means that the encoded frame is more than twice as big, which exceeds the MTU size of 1500 bytes and no fragmentation was implemented. The remaining results are as expected with shortest code lengths causing the shortest tree traversions for decoding.

Our second benchmark compares the complexity between encoder and decoder

module. Therefore, we built the exact same setup as in Figure 5.1. This time we wrote a small user space application that would keep sending frames to the receiver node with the correct header set for Huffman frames. In both of the Huffman coding Linux kernel modules we measured the time for encoding and decoding the frame. To measure the execution time as accurately as possible, we used the **get_cycles** function (provided by the Linux kernel), which returns a processor's cycle counter. The result of those tests are presented in Figure 5.3(a) for the encoding and Figure 5.3(b) for the decoding process.



(a) Encoder complexity

(b) Decoder complexity

Figure 5.3: Huffman complexity

The figures show that decoding is more computationally expensive than encoding. Missing values on the decoder side are caused by packets larger than the MTU. One interesting point in Figure 5.3(a) which needs further clarification is the fact, that the complexity for average code lengths is lower than for shortest code legths. The average code length was chosen to be 1 byte which is easier to append to an internal data buffer of 32-bit. Further, there is no need to split up the code word, because of code words crossing integer borders. This results in a lower complexity to encode words with one byte code length.

5.3.2 Hardware Module

For the hardware function verification and performance evaluation we used the same encoder-decoder setup as for the software part. Verification and evaluation was done in simulation using *ModelSim*. The behaviour of our design built in hardware should be exactly the same as our code running in simulation, when good care of the synthetization process is taken to avoid errors and to control warnings. Verification can be done through simulation with exactly the same setup as in software that is illustrated in Figure 5.1: We need to design and create a flexible testbench that can be resused to verify correct functionality for various test cases. This can be done very efficiently and quick in simulation using scripts and file based testbenches. A big advantage of hardware simulation is accuracy: Execution time can be measured in clock cycles that is the most accurate unit of time for our measurements and we can review the circuit of interest being isolated from other logic that may distort our results otherwise.

Functional Verification

In this section, we present the testbench and setup used to verify correct functionality and operation of the Huffman module in hardware. One major challenge when

writing testbenches is to write an efficient and reusable model that can be used to test and verify all possible states of the module for all possible input vectors. Figure 5.4 shows the testbench architecture:

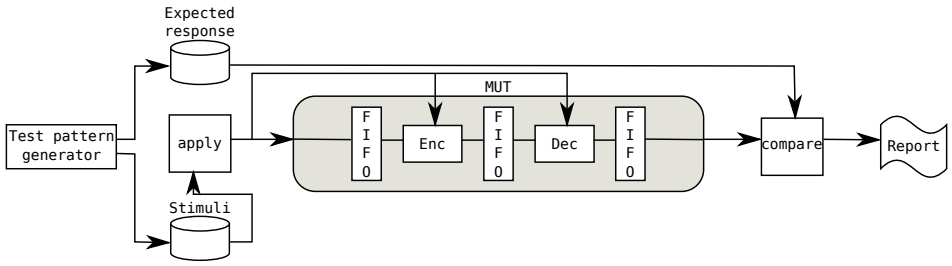


Figure 5.4: Huffman testbench

The core of the testbench was the `Test pattern generator`, which was a *Python* script that creates stimuli and expected response test vectors for Huffman frames. The art of software and hardware testing is an active research topic [63], because even today software and hardware are still shipped with bugs [64]. To quote Dijkstra here: "Testing shows the presence, not the absence of bugs." [65]. At first, we began testing with some real world data such as magazine articles or newspaper reports. Once those, somewhat random, data sets were successfull, we began testing border values. We claim that those cases offer the highest potential for bugs and misbehaviour, namely shortest and longest code words. We also added an average data test case. Finally, we introduced a test case to verify correct operation in case of filled FIFOs. Each FIFO can store up to 16 data units before it marks itself as full. Every time a module wants to write data to a FIFO, it needs to check whether the FIFO is still able to store new data or if it is full. The event of a full FIFO added some complexity and difficulties to our testing setup. In theory, the full event could be reached in every state of the Huffman module. Therefore, we added a separate process, just for testing, that would only read the final FIFO of the model under test (MUT) every 16th cycle. This would cause all other FIFOs to fill up quickly and delay the internal processing of encoder and decoder until new data values have been read from the final FIFO. The *Python* script, that we used contained three constants: The number of packets, packet size and the other one being the char to encode and decode.

This script not only generated stimuli test vectors, but at the same time also the expected response vector, because those two should be exactly the same. Data that passed the encoder and decoder stage should be the same as the original data. Our `Test pattern generator` created two files: A `Stimuli` file and an `Expected response` file. The `Stimuli` file not only contained packet data, but also ReconOS data which was necessary to initialize the register banks in the Huffman encoder and decoder module.

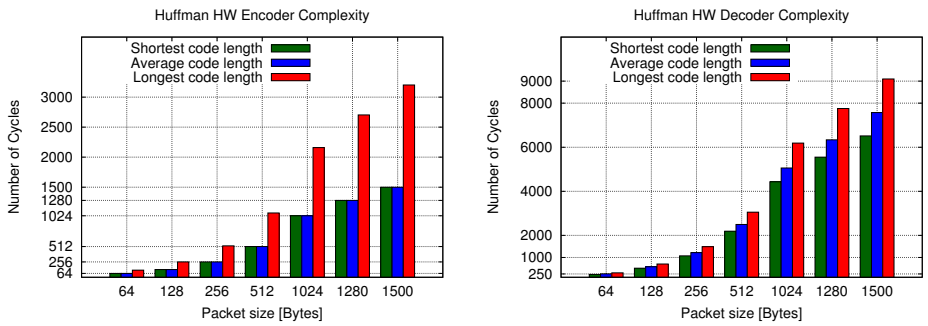
The testbench had a **stim_proc** process, which was responsible to read stimuli from the designated file and apply those to the MUT. Further, there was a **response_proc** process that acquired the response of the MUT and wrote it to another file.

After the simulation had finished, we compared the response file, with the `Expected response` file using **vimdiff(1)**. This procedure offered a quick and flexible way to compare responses. This test was run for the three cases mentioned earlier and for packet sizes described in Section 5.2.

Performance Evaluation

The previously explained testbench setup has been used to evaluate the performance of the Huffman encoder and decoder. Again, we tested for worst, average and best case in terms of code length. It should be noted here, that in order to measure the performance of the hardware implementation, we changed the FIFOs as pictured in Figure 5.4. The `NoC` implementation uses FIFOs based on BRAM modules that offer very small storage sizes with a depth of 16. The problem with FIFOs that small is that they fill up very quickly and once they are full, they block FB from working properly and eventually slow down the whole system. Our Virtex-6 FPGA provides built-in FIFOs that are much bigger (at least 512 bytes). Most FBs will probably change their internal FSM states at some point depending on the header of a received packet. This means that it will buffer the header in a shift register first and process it after the whole header has been received. Hence, the previous FB sending data to the current FB, will have to wait until our current FB finished processing the header and fetches new data. Full FIFOs significantly reduce performance and therefore, we should try to avoid or at least reduce the frequency of this event.

For the following tests we used built-in FIFOs with a depth of 512. Figure 5.5 shows the complexity of encoder and decoder in hardware.



(a) Encoder complexity

(b) Decoder complexity

Figure 5.5: Huffman complexity

Figure 5.5(a) shows the complexity of the Huffman encoder in hardware. **Encoding** for **shortest** and **average code lengths** roughly take **1 cycle** per character, whereas **longest code words** take **2 cycles**. Shortest code words are 3 bits long and average code words 8 bits. This means, we are able to read one 8-bit ASCII character and write one 3-7 bit code word per cycle. Shortest and average code lengths take exactly the same time, because as soon as there is at least one byte to write to the output FIFO, the internal buffer is emptied. This way, we avoid the unconvenient case for splitting up code words. Longest code words are 17-bit long, which means that it takes more than 2 cycles to write those code words to the output FIFO. In contrast to the previous cases, we do have the possibility of overflowing internal buffers. To avoid this, we only read new data if we have at least 17 bits of free storage in our internal buffer.

The complexity characteristic of the hardware decoder module is shown in Figure 5.5(b). Decoding is done in a parallel fashion comparing all code words at once. Why does it make a difference if they are decoded in parallel? Shouldn't the curve look exactly the same for all three cases? The decoding process and writing data to the output FIFO takes the same time for all three cases. A code word between 3 and 17-bit is read and converted into an 8-bit long ASCII character. The complexity difference

is caused by the different code word lengths: To be sure that we can decode a character, we need to have at least 17-bit in our internal buffer. At first, the internal buffer is empty. After reading 3 bytes, we have 24 bits stored in our internal buffer, which is enough to start the decoding process. We could have a match for code words between 3-bit and 17-bit. A match with a very rare code word (length 17) would empty the buffer down to 7-bit. After that, we would need to read new data for 2 more cycles, before starting over with decoding again. A 3-bit code word on the other side, would empty the internal buffer down to 21, which would trigger a new decoding round immediately. **Decoding** roughly takes **4**,**5** and **6 cycles** for **shortest**, **average** and **longest code length**.

Now that performance evaluation of software and hardware have been presented, it is time to compare them against each other. An overview and comparison between Huffman encoder in software and hardware is given in Figure 5.6.

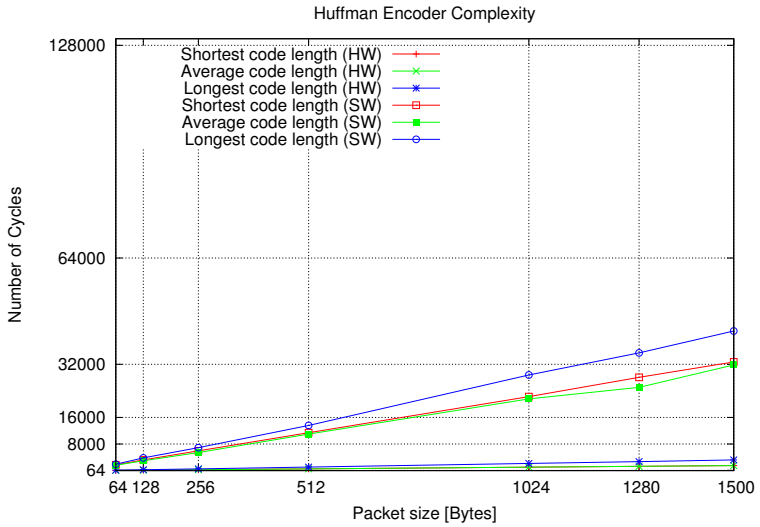


Figure 5.6: Encoder comparison

The speed up from software to hardware implementation depends on the type of data that is sent: We see that the hardware implementation runs approximately **20 times faster** when encoding character with **average** and **short code word lengths**. For **longest code words** the hardware encoder still performs **13 times faster**.

Regarding the decoder comparison, Figure 5.7 presents the measured complexity for both decoder implementations. Measured speed-ups from software to hardware implementation are close to **6**, **15** and **22 times faster** for **shortest**, **average** and **longest code lengths**.

The limiting factor of those numbers is memory access. Encoding and decoding in hardware could be done even faster, if we had a wider data bus that would transfer up to 32-bit. This would increase the performance of the encoder and decoder for the longest code words. First, we could write a 17-bit code word, each cycle and second, we could fill our internal buffer much faster for the decoding process. With a data bus width larger than 17-bit, we could fill our buffer faster than we can empty it. Additional performance gains could be achieved by increasing the clock frequency of our Huffman hardware modules. The maximum achievable frequency for encoder and decoder are presented in Table 5.2. Additional pipeline stages could be used to increase the maximal frequency further, but the main problem at this point is the data transfer between modules which slows down performance. When

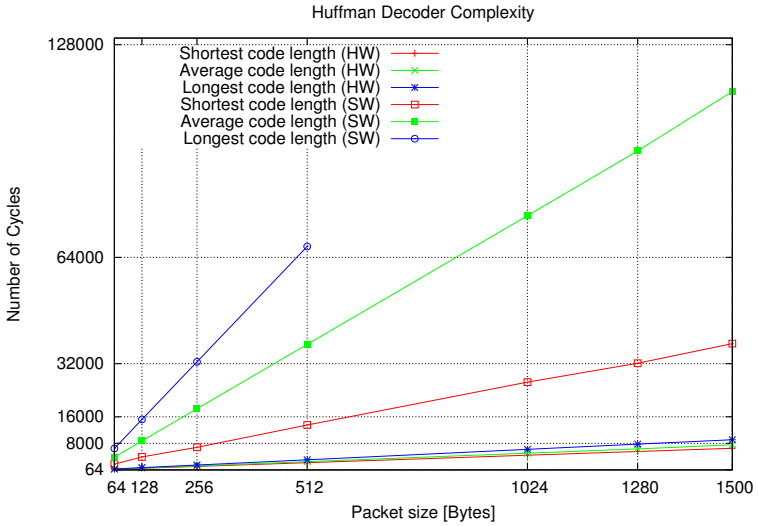


Figure 5.7: Decoder comparison

increasing the frequency we would have to increase read and write clock frequencies of the involved FIFOs as well. A better approach would be to increase the data width and keeping the frequency constant. This approach would only be possible if we had more than 1 Gigabit of bandwidth from the network interface controller.

	Max. Frequency
Encoder	142 MHz
Decoder	206 MHz

Table 5.2: Maximal frequency for encoder and decoder

5.4 Continuous Repeat Request

Within this section, we cover verification and evaluation for Continuous Repeat Request sender and receiver modules. One difference when comparing this setup with the setup of Huffman modules is the fact that we have sender and receiver modules separately. This caused similar problems and challenges as were described in the Huffman section before: Testing sender and receiver as single units is impractical, because involved CRR packets are dynamic and header fields change frequently. Further, there is a tight connection between sender and receiver module. The sender won't continue operation unless it receives ACK frames from a receiver-like device. In general, manual single unit testing is troublesome and inefficient. Therefore, we had to test and debug a complete sender-receiver setup, which increased the difficulty to locate the origin of errors.

5.4.1 Software Module

When thinking of a software implementation of CRR, most people might underestimate the difficulties and challenges involved in this task. The problems described in Section 4.2.1 caused a lot of setbacks and wasted hours trying to figure out the problem in the implementation, especially when the system runs in Linux kernel space. Most of the faced problems were related to the development environment:

Quite some changes, such as a LANA extensions to block user space packets from entering the kernel space or increased Linux socket buffers were necessary to provide a solution for running a CRR protocol in Linux kernel space and the LANA framework.

Functional Verification

Unlike the functional verification of the Huffman module, it was not practical to test and debug CRR sender and receiver in a user space application before porting it to Linux kernel space. The user space implementation would have been much more sophisticated and less reusable than for the Huffman software module. Therefore, we implemented and tested the functionality in kernel space from the start.

Luckily we were able to reuse many existing Linux kernel functions and data structures. This sped up implementation time and reduced the amount of potential bugs in our code. In fact, most of CRR's functionality was built around already existing functions and structures.

Once sender and receiver had been implemented, we could finally start testing the system pictured in Figure 5.8.

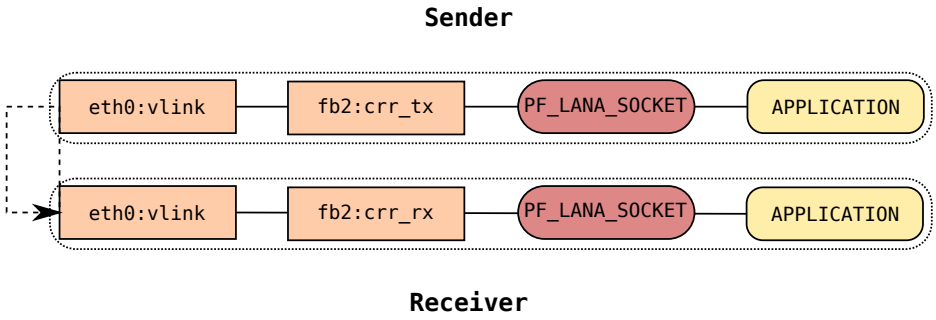


Figure 5.8: CRR setup

The setup consists of a CRR sender and receiver module that are part of a LANA protocol stack. The setup is basically the same as in Section 5.3.1. It should be noted, that connections between FBs are bidirectional. The sender module receives its packet to be sent on the EGRESS path from the `PF_LANA` socket and forwards its CRR data packets on the EGRESS path to `vlink`. CRR ACK packets are received on the INGRESS path from the `vlink`. The receiver, on the other side, receives CRR data packets from the INGRESS path by `vlink`, sends CRR ACK packets to `vlink` through the EGRESS path, and forwards CRR data packets on the INGRESS path to a `PF_LANA` socket.

To test the functional correctness of sender and receiver, we designed a user space application for testing. This application should detect missing and out of order frames. It should also be flexible enough to allow fast testing for different packet numbers and sizes.

The application running on the sender side sends the number of packets with the chosen packet size to the `PF_LANA` socket. The application also sets the header fields of the frame such as MAC addresses and the correct *Ether type*. The first byte of the payload is used as a packet counter that is increment for each packet that was sent. Those packets are sent to the CRR sender module, which builds CRR data frames out of it.

On the receiver side runs another application that extracts the counter value from the payload and compares it with the last value which has been received. The newly received value should be its previously received value incremented by one. If so, the

application continues running. In case of a mismatch, the two packet numbers are displayed using **printf(3)**.

This test printed missing packet numbers, but missing packets would cause the sender to retransmit those packets and stop sending new packets. Finding erroneous or missing frames is tedious and time consuming, when roughly sending 20'000 packets, before the problem happened for the first time. The missing packets in user space were present in kernel space which lead us to the conclusion that the problem was caused by full socket buffers. The user space couldn't keep up reading arriving packets fast enough. The socket buffer got full and newly arriving packets were discarded.

After this problem has been solved we started testing our setup with varying packet numbers, packet sizes, and window sizes of the CRR modules. Those tests verified correct operation of the sender and receiver module. More details of the testing process such as packet lengths can be found in Section 5.2.

Performance Evaluation

To see how our CRR implementation performs in comparison to other reliability protocols, we setup a benchmark to measure TCP's performance. We used **iperf** to measure the maximum bandwidth between a client and server setup. To test this setup for various packet sizes, we had to disable the *Nagle algorithm* presented in RFC896 [66]. The algorithm helps improving the efficiency of TCP/IP networks by combining several small packets into one big packet. This would prevent us from testing the performance of TCP for various packet sizes. The results are presented in Figure 5.9.

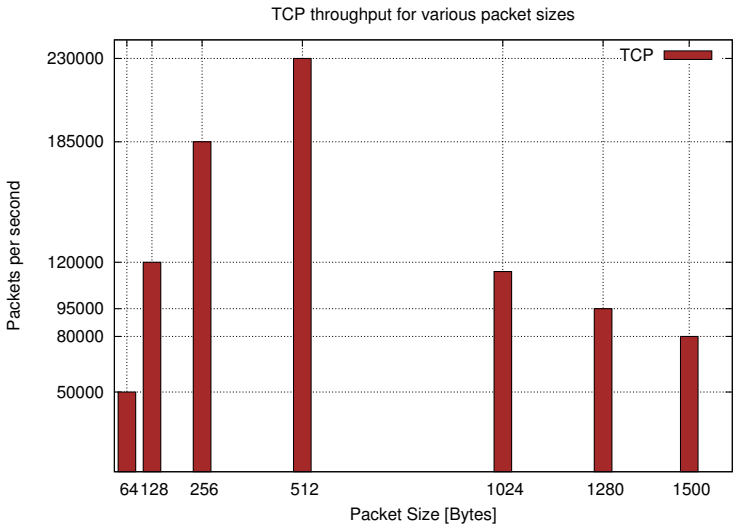


Figure 5.9: TCP throughput

It is interesting to note how TCP's packets per second rate ramps up from packet sizes of 64 until 512 bytes. We would expect to see higher packet rates for small packet sizes. For a packet size of 1500 bytes TCP's performance is very close to the maximal achievable performance with a bandwidth of 960 Mbit. Next, we will compare it with the performance of a plain LANA stack, and a LANA stack including CRR.

We used the setup from Figure 5.8 to run the software benchmarks and tested the throughput for various window and packet sizes. We wanted to test the maximum

performance of our module and tried to avoid any overhead caused by a full transmission queue in the CRR sender. Therefore, we chose a maximum queue length, that would be large enough to handle the transmission and transmitted bursts of data with 32'000 packets. The user space application allows to choose to set the number of packets and their size during run time. The user space application on the receiver side started a timer and counted the number of received packets. It also validated correct reception by checking the packet numbers stored in the payload.

Another reason for this setup was the fact, that high load eventually caused the system to be less responsive and the sender's performance to decrease. The later was probably caused by the system being slow and working to capacity. The origin of this problem still needs further clarifications. The localization is difficult, because it could be the user space to kernel space interface, or a potential bug in the CRR sender module. Further investigations are necessary to discover and fix this problem. The problem does not seem to be related to CRR functionality, because the transmission is still working although with reduced performance. This problem does not manifest itself, when several data bursts of 32'000 packets are sent at once. It seems to be related to data transfers with high load over a longer period of time.

The results for this benchmark are presented in Figure 5.10.

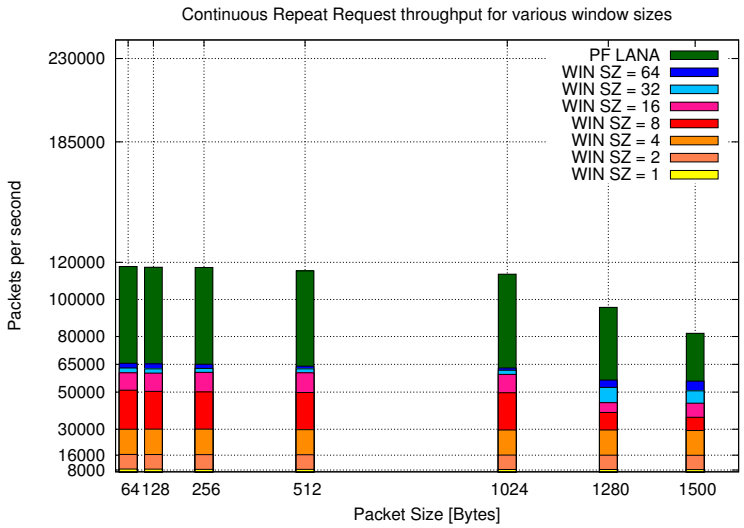


Figure 5.10: CRR throughput

Unlike TCP, the throughput of our CRR module is flat for packet sizes between 64 and 1024 bytes. For larger packets, the additional effort for copying or sending packets can be seen from the characteristic of the graph. We assume that a LANA internal memory allocation strategy is responsible for this behaviour, because the performance of the LANA stack looks exactly the same. This possible memory bottleneck could prevent the protocol stack from reaching higher throughputs for small packet sizes.

Another limiting factor is the transmission delay between sender and receiver and the delay introduced by processing the packet at the receiver side. After sending the last packet of the transmission window, we need to wait for all packets to be acknowledged, before starting a new window. Imagine that all packets have been acknowledged so far. After the last packet is sent, we need to wait a long time before we receive the ACK. During this time, there won't be other outstanding packets. This fact heaviliy decreases the performance of the system, but is a characteristic of CRR.

In a next step, we should implement a CRR version with a larger maximal window size and a sliding window implementation instead of static windows (similar to TCP). This window would start at the position of the oldest oustanding ACK. Each time an ACK is received, the window is shifted by one. If the window is big enough we are able to send data continuously and only have to wait for ACKs in the event of packet loss and retransmissions.

5.4.2 Hardware Module

The same sender-receiver setup from the software verification was used in hardware. Presented results have been acquired from the simulation environment. The same file-based testbench approach has been used as for Huffman coding modules in hardware before. In the following section, we present our testbench for functional verification.

Functional Verification

The testbench architecture of our CRR sender receiver setup is presented in this section. The testbench and involved scripts are very similar to the previously presented approach in Section 5.3.2. In order to avoid redundancy, we will only discuss keypoints that vary. The whole testbench can be seen in Figure 5.11.

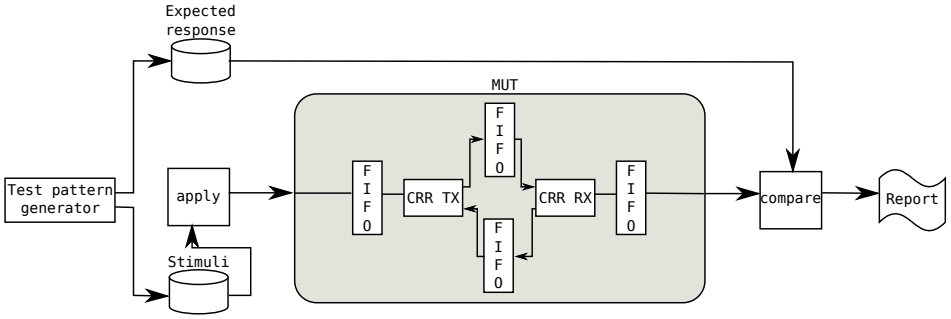


Figure 5.11: CRR testbench

A *Python* script is taking care of creating stimuli file vectors and expected responses. There is no need to initialize internal data structures in the CRR modules before starting the operation. Therefore, CRR, as is, is not connected to the ReconOS interface. Between sender and receiver there are two FIFOs, that are used for carrying CRR data and ACK frames. The script responsible for the `Test pattern generator` was slightly different this time. The sender will alter the header of the frame, before sending it. The effects of the sender could easily be precomputed and a separate test vector for `Expected responses` was generated. It should be noted here, that the hardware module expects the packets to arrive in the correct order, which means that the packet size doesn't need to be changed by the hardware module. Therefore, header fields must be free, or will be overwritten. For instance, the sequence number will be written after the `Ethernet Type`, so this field should not carry any other data.

The *Python* script is configurable and allows the user to set packet sizes, number of packets and payload data. After the test vectors had been generated, the simulation for functional verification started. We sent between 10 and 1000 packets for different packet sizes and window sizes. This setup already takes care of full FIFOs, because of the sheer amount of data which is transfered and the transition states, which would block data from entering the module.

The testbench itself had the same processes as before. After the simulation came to an end, we compared actual responses with expected responses using **vimdiff(1)** again. This test resulted in identical responses for all our tests, which should be sufficient to prove functional correctness. As a side note, we also created special testing cases, that arise when things go wrong. In one case, we created all possible permutations that could possibly show up on the receiver side, to see if packets are forwarded in the correct order. Those permutations were tested for different window sizes as well, which increases the test vector immensely. Additional test cases were time-outs, retransmissions and duplicate frames. All those tests passed successfully.

Performance Evaluation

In this section we used the previous testbench again. Unfortunately, our model does not include transmission delays, so we measured the complexity of sender and receiver modules. We used built in FIFOs again to run the performance benchmark, and to not slow down our hardware blocks, because of full FIFOs during the header processing stage. Figure 5.12 shows the results for the complexity evaluation for sender and receiver modules.

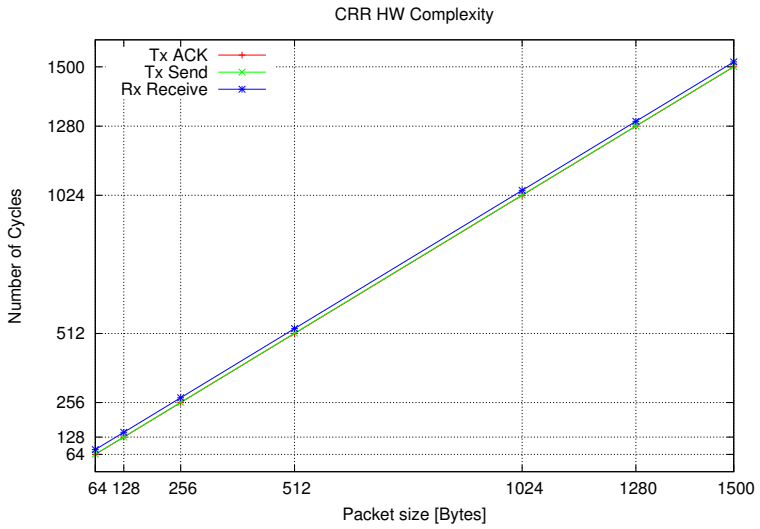


Figure 5.12: CRR complexity

There are three different cases that yield slightly different results: We measured the time it took the **sender module** to **send a packet** and to **receive an ACK**. For the **receiver module** we measured the time it took to **receive a packet** and **send an ACK**. The overall equation for the number of cycles for a packet is:

$$\#Cycles = \#Bytes + Latency$$

We measured a **latency** of **1 cycle** when **sending** a CRR data packet. A latency of 1 means that a packet with a packet size of 60 bytes takes 61 cycles to be processed. The additional cycle was used to initialize some internal data structure. This additional delay is negligible for large packets.

We also measured the **reception** of **ACK** packets by the **sender**. Receiving an ACK takes exactly the same amount of cycles that are necessary to read it. ACKs don't need to be forwarded to following FBs, which means that we only need to look at the packet header to decide, what to do next.

Finally, the **receiver** has a bigger latency than the sender module, because it needs to wait for the whole header to be received, before making a decision on how to continue. This causes a **latency** of **17 cycles** when **forwarding packets**, and **18 cycles** when **sending** the **corresponding ACK** of a packet.

The performance of the CRR hardware modules are close to optimal and suffer from a small latency which is introduced by the header checking process. Possibilities to improve this are to either increase the data width of the FIFO interface, or to change the internal header structure. The data bus width is 8-bit and may run at a frequency of 125 MHz, which is the maximal bandwith of our Gigabit Ethernet controller. Increasing the data bus' width is not an option unless we get to use a faster Ethernet controller.

The packet header is the main source of latency. It takes 14 cycles to receive `MAC addresses` and `Ether type`. A different internal header representation could help to reduce latency. It would be possible to either rearrange header fields, such that the module can extract its data of interest from the header much earlier, or to introduce some specific encapsulation and decapsulation technique for headers. Another option is to strip FB-specific headers before forwarding the packet to the next FB. The achievable frequencies of the CRR modules are presented in Table 5.3.

	Max. Frequency
CRR sender	122 MHz
CRR receiver	123 MHz

Table 5.3: Maximal frequency for CRR sender and receiver

In the current design FBs are clocked at a frequency of 100 MHz. The maximal frequency of CRR sender and receiver module is slightly below the 125 MHz used in the Gigabit Ethernet controller. There are 3 possible approaches to overcome this problem:

- Shorten the longest path by adding pipeline stages which increases the maximum frequency
- A minor code or design modification may be sufficient to reach a maximum frequency of 125 MHz
- Reduce the read and write clock frequency of the FIFOs connected to the FB, but widen the data width. Transfer 32-bit per clock cycle and reduce the clock frequency to 31.25 MHz in order to achieve Gigabit bandwidth.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Within this section we summarize and present the most important insights that we gained during the work on this thesis:

We have seen two different protocols that were implemented as software and hardware modules. Those protocols differ in terms of computational complexity, expense and achievable speed-up for a hardware implementation. Huffman coding provides a high level of parallelism that can be exploited in hardware. We realized a hardware execution speed-up of up to 20 times depending on the code word length. CRR on the other side, depends heavily on memory accesses such as buffering and queueing packets. Its performance is further limited by transmission delays between sender and receiver. Memory accesses are a common bottleneck for both protocols: The Huffman coding hardware modules, for instance, could be further improved with a wider data bus. Encoder and decoder modules cannot fill or empty their internal data buffers fast enough, which forces our coding modules to idle. The hardware implementation of CRR spends the majority of its cycles on data transfers. Increasing the bandwidth of the data bus would be the most efficient improvement in terms of performance. Unfortunately, the bandwidth is limited by the Ethernet network interface controller which achieves Gigabit bandwidth on on our target platform.

The following list summarizes major contributions of this thesis besides the report:

- Design and implementation of Huffman coding modules in software and hardware. Software modules were implemented as LANA FB that run in Linux kernel space. Hardware modules were implemented as EmbedNet FB. Both implementations are flexible in the sense that they may change their Huffman tree and code book during run time. The hardware implementation uses a high level of parallelism and runs up to 20 times faster than its software counterpart.

- Design and implementation of Continuous Repeat Request modules in software and hardware. We created sender and receiver modules for LANA and EmbedNet. Operation parameter such as window size can be easily changed in the source code.

- Design of the state transition mechanism in software and hardware. The proposed mechanism was developed with respect to the needs of our chosen protocols and are reusable for future protocols.

- Implementation of the state transition mechanism for CRR modules in hardware. A specific data block structure has been designed to transfer protocol state between software and hardware. The implementation can be used as an example for other protocols and can partly be reused.

- Functional verification and performance evaluation of our modules in software and hardware. A lot of time was spent on the extensive testing of components when debugging or simulating code.

  Minor contributions include various scripts and user space programs that have been written for testing reasons, building file based testbenches for simulating the VHDL code, debugging our components and testing the LANA framework extensively. We also discovered some bugs and problems in LANA, when testing our components, or contributed to necessary LANA extensions from a design perspective.

6.2 Future Work

The goal for future work is the integration of the various subsystems such as NoC, LANA and EmbedNet to build a single system. Further implementation and testing efforts are necessary to achieve this goal.

In the next step, a test system has to be built including software and hardware components. The presented state transition mechanism could be used to migrate tasks. So far, our software components have been tested and evaluated on a desktop machine, but in the future, we plan to run both modules on the designated development board. First testing experiences showed promising results when running Huffman coding modules on a static protocol stack.

In order to run those various subsystems together at the same time, a common addressing scheme is needed. Using IDPs for packet forwarding in software and hardware is the obvious choice, because it has already been implemented in LANA. Further design decisions and efforts are required to find an effective addressing strategy. At the moment, LANA's PPE is responsible for packet routing, but it is highly inefficient to send packets to the PPE, if the following FB is located in hardware. A PPE clone needs to be implemented in NoC, which forwards packets to hardware FBs. Additionally, it should be clear from the address whether the FB is located in software, or hardware. The PPE in software needs to know when to forward a packet to the PPE in hardware and vice versa. At last, the address of a FB in hardware, should be similar to the address of the same FB residing in software. This would simplify the state transition between software and hardware. Otherwise, the packets that are looped have no chance to reach their corresponding FB, which is about to continue operation. If the address conversion was a simple operation, the PPE could change the address for each packet in the loop, which would enable correct forwarding of packets at a later stage.

Data paths between FB modules in hardware are unidirectional, whereas in software each port offers bidirectional data transmission in both directions. Some protocols such as CRR, for instance, require several input and output ports. The CRR sender module requires one input port for normal data packets and one for ACK packets in order to work properly. Otherwise, a FIFO filled with data packets could prevent ACK packets from entering the module. Therefore, we need to double the amount of `INGRESS` and `EGRESS` paths in the NoC.

Unlike in the OSI model [67], the boundaries of our protocol stack are flexible and may change during run time. We don't have static layers and it is difficult to define layers in a flexible protocol stack. The OSI communication model offers the advantage of a well defined set of layers, that are distinguishable from each other.

This concept gives way to data encapsulation and decapsulation. Such a technique cannot be used in our flexible approach, because we don't know in advance how many other FBs are going to make use of a certain header structure. A well defined common header format could help to avoid this problem and introduce some similar data encapsulation technique.

At last, a possible future protocol could be a LZ77 compression protocol. The DEFLATE algorithm, which has been presented in Section 2.5.2, is used as the underlying compression algorithm for gzip and makes use of a combination of Huffman coding and LZ77. This would result in an efficient compression network protocol. In combination with a reliability protocol such as CRR, it could be used to transfer large amounts of data across the network in an efficient manner.

References

[1] A. Keller, D. Borkmann, and W. Mühlbauer, "Efficient implementation of dynamic protocol stacks," in *Proceedings of the 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*, pp. 83–84, IEEE Computer Society, 2011.

[2] A. Tanenbaum, "Computer networks, 4th edition," pp. 77–81, 2003.

[3] A. Hasler, I. Talzi, J. Beutel, C. Tschudin, and S. Gruber, "Wireless sensor networks in permafrost research-concept, requirements, implementation and challenges," in *Proc. 9th Int. Conf. on Permafrost (NICOP 2008)*, vol. 1, pp. 669–674, 2008.

[4] V. Shnayder, M. Hempstead, B. Chen, G. Allen, and M. Welsh, "Simulating the power consumption of large-scale sensor network applications," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pp. 188–200, ACM, 2004.

[5] J. Rabaey, M. Ammer, J. da Silva Jr, D. Patel, and S. Roundy, "Picoradio supports ad hoc ultra-low power wireless networking," *Computer*, vol. 33, no. 7, pp. 42–48, 2000.

[6] APNIC, "APNIC IPv4 Address Pool Reaches Final /8." `http://www.apnic.net/publications/news/2011/final-8`, 2011. [Online; accessed 12-May-2012].

[7] K. Das, "IPv6 - The History and Timeline." `http://www.ipv6.com/articles/general/timeline-of-ipv6.htm`, 208. [Online; accessed 12-May-2012].

[8] L. Colitti, S. Gunderson, E. Kline, and T. Refice, "Evaluating ipv6 adoption in the internet," in *Passive and Active Measurement*, pp. 141–150, Springer, 2010.

[9] G. Bouabene, C. Jelger, C. Tschudin, S. Schmid, A. Keller, and M. May, "The autonomic network architecture (ana)," *Selected Areas in Communications, IEEE Journal on*, vol. 28, no. 1, pp. 4–14, 2010.

[10] E. Lubbers, M. Platzner, C. Plessl, A. Keller, and B. Plattner, "Towards adaptive networking for embedded devices based on reconfigurable hardware," in *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'10). Las Vegas, NV, USA: CSREA Press*, pp. 225–231, 2010.

[11] A. Keller, B. Plattner, E. Lubbers, M. Platzner, and C. Plessl, "Reconfigurable nodes for future networks," in *GLOBECOM Workshops (GC Wkshps), 2010 IEEE*, pp. 357–361, IEEE, 2010.

[12] E. Lubbers and M. Planner, "Reconos: An rtos supporting hard-and software threads," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pp. 441–446, IEEE, 2007.

[13] E. Lubbers and M. Platzner, "Reconos: Multithreaded programming for reconfigurable computers," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 9, no. 1, p. 8, 2009.

[14] EPiCS, "EPiCS Project." `http://www.epics-project.eu/`, 2012. [Online; accessed 12-May-2012].

[15] hydrogenaudio, "Vorbis Wiki." `http://wiki.hydrogenaudio.org/index.php?title=Vorbis`, 2012. [Online; accessed 1-June-2012].

[16] C. Tschudin and R. Gold, "Network pointers," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 1, pp. 23–28, 2003.

[17] D. Borkmann, "Lightweight Autonomic Network Architecture," Master's thesis, Swiss Federal Institute of Technology, Switzerland, 2011.

[18] Xilinx, "Xilinx ML605." `http://www.xilinx.com/products/boards-and-kits/EK-V6-ML605-G.htm`, 2012. [Online; accessed 15-May-2012].

[19] Xilinx, "Xilinx Microblaze." `http://www.xilinx.com/tools/microblaze.htm`, 2012. [Online; accessed 15-May-2012].

[20] R. Huber, "Hardware Design for EmbedNet," Master's thesis, Swiss Federal Institute of Technology, Switzerland, 2012.

[21] C. Tye and G. Fairhurst, "A review of ip packet compression techniques," in *Proceedings of PostGraduate Networking Conference, Liverpool*, Citeseer, 2003.

[22] M. Degermark, B. Nordgren, and S. Pink, "Ip header compression," tech. rep., RFC 2507, february, 1999.

[23] C. Bormann, C. Burmeister, M. Degermark, H. Fukushima, H. Hannu, L. Jonsson, R. Hakenberg, T. Koren, K. Le, Z. Liu, *et al.*, "Robust header compression (rohc): Framework and four profiles: Rtp, udp, esp, and uncompressed," tech. rep., RFC 3095, July, 2001.

[24] A. Shacham, M. Thomas, R. Pereira, and B. Monsour, "Ip payload compression protocol (ipcomp)," *Consultant*, 2001.

[25] R. Pereira, "Ip payload compression using deflate," 1998.

[26] J. loup Gailly and M. Adler, "zlib." `http://zlib.net/`, 2012. [Online; accessed 15-May-2012].

[27] L. Deutsch, "Deflate compressed data format specification version 1.3," 1996.

[28] D. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[29] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *Information Theory, IEEE Transactions on*, vol. 23, no. 3, pp. 337–343, 1977.

[30] A. Feldspar, "An Explanation of the Deflate Algorithm." `http://www.zlib.net/feldspar.html`, 1997. [Online; accessed 15-May-2012].

[31] R. Friend and R. Monsour, "Ip payload compression using lzs," 1998.

[32] S. Rigler, W. Bishop, and A. Kennings, "Fpga-based lossless data compression using huffman and lz77 algorithms," in *Electrical and Computer Engineering, 2007. CCECE 2007. Canadian Conference on*, pp. 1235–1238, IEEE, 2007.

[33] Z. Aspar, Z. Mohd Yusof, and I. Suleiman, "Parallel huffman decoder with an optimized look up table option on fpga," in *TENCON 2000. Proceedings*, vol. 1, pp. 73–76, IEEE, 2000.

[34] L. Acasandrei and M. Neag, "A fast parallel huffman decoder for fpga implementation," *Acta Tehnica Napocensis Electronics and Telecommunications*.

[35] L. Agostini, I. Silva, and S. Bampi, "Pipelined fast 2d dct architecture for jpeg image compression," in *Integrated Circuits and Systems Design, 2001, 14th Symposium on.*, pp. 226–231, IEEE, 2001.

[36] R. Doss, "OpenCores - Huffman Decoder." `http://opencores.org/project,huffmandecoder`, 2009. [Online; accessed 15-May-2012].

[37] V. Zandy and B. Miller, "Reliable network connections," in *Proceedings of the 8th annual international conference on Mobile computing and networking*, pp. 95–106, ACM, 2002.

[38] J. Mogul, "Tcp offload is a dumb idea whose time has come," in *9th Workshop on Hot Topics in Operating Systems (HotOS IX), Lihue, HI*, 2003.

[39] J. Sax, D. Velten, and R. Hinden, "Reliable data protocol," 1984.

[40] C. Partridge and R. Hinden, "Version 2 of the reliable data protocol (rdp)," *Request for Comments*, vol. 1151, 1990.

[41] C. Wang, K. Sohraby, B. Li, M. Daneshmand, and Y. Hu, "A survey of transport protocols for wireless sensor networks," *Network, IEEE*, vol. 20, no. 3, pp. 34–40, 2006.

[42] F. Stann and J. Heidemann, "Rmst: Reliable data transport in sensor networks," in *Sensor Network Protocols and Applications, 2003. Proceedings of the First IEEE. 2003 IEEE International Workshop on*, pp. 102–112, IEEE, 2003.

[43] A. Tanenbaum, "Computer networks, 4th edition," pp. 246–264, 2003.

[44] L. K. Archives, "T H E /proc F I L E S Y S T E M." `http://www.kernel.org/doc/Documentation/filesystems/proc.txt`, 2009. [Online; accessed 15-May-2012].

[45] Unicode, "The Unicode FAQ." `http://unicode.org/faq/utf_bom.html`, 2012. [Online; accessed 15-May-2012].

[46] R. Benice and A. Frey Jr, "An analysis of retransmission systems," *Communication Technology, IEEE Transactions on*, vol. 12, no. 4, pp. 135–145, 1964.

[47] N. W. Group, "Benchmarking Methodology for Network Interconnect Devices." `http://www6.ietf.org/rfc/rfc2544`, 1999. [Online; accessed 15-May-2012].

[48] J. Reynolds and J. Postel, "Assigned numbers," tech. rep., STD 2, RFC 1700, October, 1994.

[49] B. B. J. Hall, "Beej's Guide to Network Programming." http://beej.us/guide/bgnet/output/html/multipage/htonsman.html, 2009. [Online; accessed 24-May-2012].

[50] J. Corbet, A. Rubini, and G. Kroah-Hartman, "Linux device drivers," pp. 128–129, 2005.

[51] J. Corbet, A. Rubini, and G. Kroah-Hartman, "Linux device drivers," pp. 120–121, 2005.

[52] K. Owens, "x86_64 kernel stack organization." http://www.kernel.org/doc/Documentation/x86/x86_64/kernel-stacks, 2006. [Online; accessed 24-May-2012].

[53] K. Sovani, "Kernel : likely/unlikely macros." http://kerneltrap.org/node/4705, 2005. [Online; accessed 24-May-2012].

[54] M. D. C. ETHZ, "VHDL Naming Conventions." http://www.dz.ee.ethz.ch/de/information/hdl-hilfe/vhdl-namenskonvention.html, 2009. [Online; accessed 24-May-2012].

[55] J. Gaisler, "A structured vhdl design method," *Fault-tolerant microprocessors for space applications*, 2010.

[56] T. Jones, "Boost socket performance on Linux." http://www.ibm.com/developerworks/linux/library/l-hisock/index.html, 2006. [Online; accessed 25-May-2012].

[57] L. M. Page, "ioctl(2) - Linux man page." http://linux.die.net/man/2/ioctl, 2000. [Online; accessed 26-May-2012].

[58] linsyssoft, "KGDB." http://kgdb.linsyssoft.com/index.html, 2011. [Online; accessed 27-May-2012].

[59] L. Torvalds, "Availability of kdb." http://lwn.net/2000/0914/a/lt-debugger.php3, 2000. [Online; accessed 27-May-2012].

[60] P. Gutenberg, "Project Gutenberg." http://www.gutenberg.org/, 2012. [Online; accessed 27-May-2012].

[61] E. R. Daniel Borkmann, "netsniff-ng the packet sniffing beast." http://netsniff-ng.org/, 2012. [Online; accessed 29-May-2012].

[62] packETH, "packETH - Ethernet packet generator." http://packeth.sourceforge.net/, 2011. [Online; accessed 29-May-2012].

[63] G. Myers, C. Sandler, and T. Badgett, *The art of software testing*. Wiley, 2011.

[64] I. Newsroom, "Intel Identifies Chipset Design Error, Implementing Solution." http://newsroom.intel.com/community/intel_newsroom/blog/2011/01/31/intel-identifies-chipset-design-error-implementing-solution, 2011. [Online; accessed 27-May-2012].

[65] J. Meerts, "http://www.testingreferences.com/testinghistory.php." http://www.testingreferences.com/testinghistory.php, 2012. [Online; accessed 27-May-2012].

[66] J. Nagle, "Congestion control in ip/tcp internetworks," 1984.

[67] H. Zimmermann, "Osi reference model–the iso model of architecture for open systems interconnection," *Communications, IEEE Transactions on*, vol. 28, no. 4, pp. 425–432, 1980.

Appendix A

Task Description

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Master Thesis

Network Protocols for Embedded Devices with Dynamic Hardware / Software Mapping (TBD)

Florian Deragisch

Advisor: Ariane Keller, ariane.keller@tik.ee.ethz.ch
Co-Advisor: Dr. Stephan Neuhaus, stephan.neuhaus@tik.ee.ethz.ch
Co-Adviser: Daniel Borkmann, borkmann@iogearbox.net
Professor: Prof. Dr. Bernhard Plattner, plattner@tik.ee.ethz.ch

November 2011 - May 2012

1 Introduction

This master thesis is in the context of the EPiCS project. The goal of the EPiCS project is to lay the foundation for engineering the novel class of proprioceptive computing systems. Proprioceptive computing systems collect and maintain information about their state and progress, which enables self-awareness by reasoning about their behaviour, and self- expression by effectively and autonomously adapting their behaviour to changing conditions. Concepts of self-awareness and self-expression are new to the domains of computing and networking; the successful transfer and development of these concepts will help create future heterogeneous and distributed systems capable of efficiently responding to a multitude of requirements with respect to functionality and flexibility, performance, resource usage and costs, reliability and safety, and security.

In this thesis we focus on the networking aspect of EPiCS which we call *EmbedNet*. EPiCS uses the network architecture developed in the ANA project as a basis. The ANA network architecture is a novel architecture that enables flexible, dynamic, and fully autonomous formation of network nodes. In the EPiCS project we develop the ANA architecture further. On the one hand we will develop mechanisms to adapt the functionality provided by the protocol stack at runtime, on the other hand we will develop mechanisms that map the networking functionality dynamically to either hardware or software.

The objective of this Masters Thesis is to implement several protocols that can be mapped dynamically either to hardware or to software.

2 Assignment

This assignment aims to outline the work to be conducted during this thesis. The assignment may need to be adapted over the course of the project.

2.1 Objectives

The goal of this Master thesis is to develop several networking protocols for EmbedNet. The developed protocols should run in both, hardware and in software. Therefore, a single protocol needs to be implemented in C for execution in software and in VHDL for the execution in hardware. In order to allow for dynamic changes in the mapping of the protocols to either hardware or software a mechanism for transferring protocol state between the two implementations is required.

2.2 Tasks

This section gives a brief overview of the tasks the student is expected to perform towards achieving the objective outlined above. The binding project plan will be derived over the course of the first three weeks depending on the knowledge and skills the student brings into the project.

2.2.1 Familiarization

- Study the available literature on ANA, LANA, EmbedNet and Reconos [1, 2, 3, 4, 5].
- Setup a Linux kernel development environment by following the "Getting started with LANA" tutorial provided in [4].
- Setup an FPGA development environment with the Xilinx tools 12.3 (exact) and ModelSim SE version 6.1f (or higher).
- Familiarize yourself with the Linux kernel coding guidelines.
- Familiarize yourself with the git version control system and with github. Clone the reconos source code repository [6].
- Install the eCos source code and cross compilation toolchain [7].
- Verify your toolchain by running first the sort_demo_thermal and then the pr_demo.
- In collaboration with the advisor, derive a project plan for your master thesis. Allow time for the design, implementation, evaluation, and documentation of your software.

2.2.2 Architecture and hardware design

- Together with your advisors determine a set of protocols that you would like to implement. Choose protocols with different expected characteristics with regards to performance differences between an execution in hardware and in software.
- Determine the exact functionality to be implemented, verify that you can implement it in both hardware and in software.
- Design a unified interface to set and collect internal state for the protocols.
- Design your protocols for the implementation in hardware and software. Your design should follow the ideas of ANA and the implementation constraints given by LANA and ReconOS.

2.2.3 Implementation

- Determine an appropriate version control system. The EPiCS project is hosted at github while LANA is hosted at repo.or.cz. You might want to put your code in the same repositories.
- Implement a dummy protocol in software.
- Implement the system to set and collect state. Verify your system with the dummy protocol.
- Start with the implementation of the protocols in software, continue with the implementation of the protocols in hardware.
- Optional: Develop different hardware implementations for your protocols with different optimization goals (area, speed, power, etc.).

2.2.4 Validation

- Validate the correct operation of your implementation after each implementation step. Use for your validation different packet sizes (short, long, even or odd number of bytes, etc.).
- Check the resilience of the implementation, including its configuration interface, to uneducated users.

2.2.5 Evaluation

- Do a performance evaluation of your protocols for the packet sizes specified in RFC 2544.
- Compare the performance for each protocol when it is run in hardware and in software.
- Optional: Determine the bottlenecks of your implementation.
- Optional: Do a performance comparison between packet forwarding for different combinations of hardware and software networking functional blocks.

2.2.6 Documentation

- Appropriate source code documentation.
- Write a step-by-step how-to that describes the compilation of your code, the loading of the code into the hardware and the execution of your code.
- Write a documentation about the design, implementation, validation and evaluation of your work.

3 Milestones

- Provide a "project plan" which identifies the milestones.
- Two intermediate presentations: Give a presentation of 10 minutes to the professor and the advisors. In this presentation, the student presents major aspects of the ongoing work including results, obstacles, and remaining work.
- Final presentation of 15 minutes in the CSG group meeting, or, alternatively, via teleconference. The presentation should carefully introduce the setting and fundamental assumptions of the project. The main part should focus on the major results and conclusions from the work.
- Any software that is produced in the context of this thesis and its documentation needs to be delivered before conclusion of the thesis. This includes all source code and documentation. The source files for the final report and all data, scripts and tools developed to generate the figures of the report must be included. Preferred format for delivery is a CD-R.
- Final report. The final report must contain a summary, the assignment, the time schedule and the Declaration of Originality. Its structure should include the following sections: Introduction, Background/Related Work, Design/Methodology, Validation/Evaluation, Conclusion, and Future work. Related work must be referenced appropriately.

4 Organization

- Student and advisor hold a weekly meeting to discuss progress of work and next steps. The student should not hesitate to contact the advisor at any time. The common goal of the advisor and the student is to maximize the outcome of the project.
- The student is encouraged to write all reports in English; German is accepted as well.
- The core source code will be published under the GNU general public license.

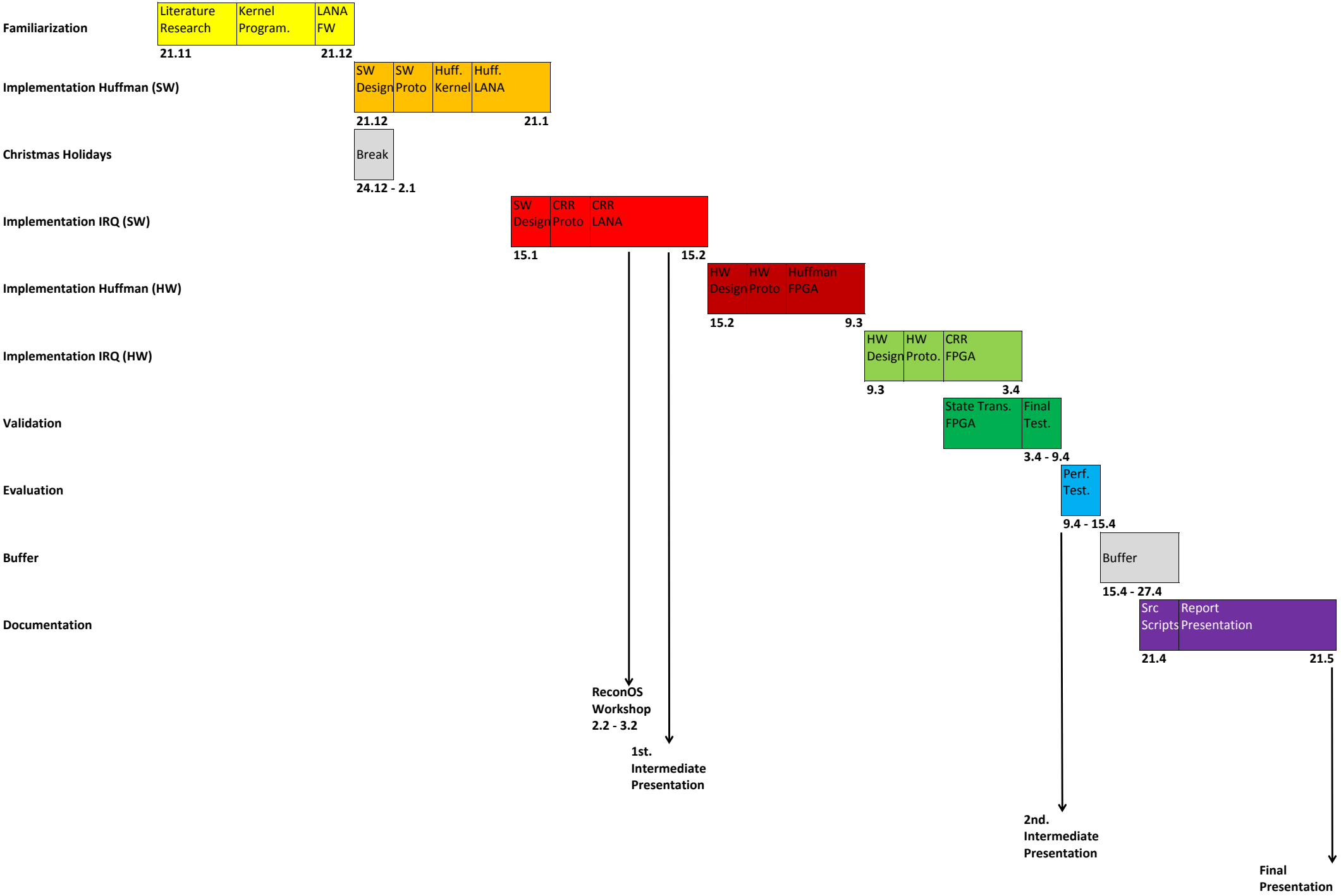
5 References

[1] ReconOS: Multithreaded Programming for Reconfigurable Computers: Description of the hardware/software architecture
[2] Reconifgurable Nodes for Future Networks: Description of how we would like to use the hw/sw architecture to build reconfigurable networks
[3] The Autonomic Network Architecture (ANA): Description of the ideas and sw prototype for configurable networks
[4] Lightweight Autonomic Network Architecture, Master Thesis of Daniel Borkmann
[5] https://github.com/EPiCS/epics-org/blob/master/deliverables/D3-1_Architecture_And_Tool_Flow/D3-1_Architecture_And_Tool_Flow.pdf
[6] https://github.com/EPiCS/reconos/

[7] webpage: ecos.sourceware.org, a version that is compliant with reconos is in the github repository, cross compilation tools: gcc-4.1.2 glibc-2.3.6
Webpages:
http://www.ana-project.org
http://www.epics-project.eu
http://www.reconos.de

Appendix B

Project Plan


Familiarization
Literature Research
Kernel Program.
LANA FW
21.11
21.12
Implementation Huffman (SW)
SW Design
SW Proto
Huff. Kernel
Huff. LANA
21.12
21.1
Christmas Holidays
Break
24.12 - 2.1
Implementation IRQ (SW)
SW Design
CRR Proto
CRR LANA
15.1
15.2
Implementation Huffman (HW)
HW Design
HW Proto
Huffman FPGA
15.2
9.3
Implementation IRQ (HW)
HW Design
HW Proto.
CRR FPGA
9.3
3.4
Validation
State Trans. FPGA
Final Test.
3.4 - 9.4
Evaluation
Perf. Test.
9.4 - 15.4
Buffer
Buffer
15.4 - 27.4
Documentation
Src Scripts
Report Presentation
21.4
21.5
ReconOS Workshop 2.2 - 3.2
1st. Intermediate Presentation
2nd. Intermediate Presentation
Final Presentation

Appendix C

Getting Started

C.1 Loading Functional Blocks in LANA

LANA Environment Setup: Familiarization and installation of the LANA framework according to the *Getting Started with LANA* section of [17]. NOTE: The location of the repository changed to https://github.com/EPiCS/reconos.git where LANA is located under `reconos/linux/lana`.

Loading the LANA Stack: The protocol stack is loaded using scripts such as the one from Listing C.1. The syntax of the script is `scriptname status interface`. Status is either `up` or `down`. The interface is the ethernet interface that should be used.

Example: **./scriptname up eth0**

Open AF_LANA Socket First, we need to run the user space application that opens the socket. The application needs to wait for the socket to be connected at this point. At this point the socket shows up as a functional block. We need to bind the socket functional block with the last functional block in our graph. Now the user space application may continue its execution and is able to send data to the LANA protocol stack. The whole process can be seen in Figure C.1 and Figure C.2.

Send Packets: The easiest way to send packets is by creating raw Ethernet frames. Make sure to respect the defined headers from Section 4, otherwise those packets will be filtered (wrong ETH type), or payload data may be overwritten. Also see the sample applications on the CD.

Unloading the LANA Stack: Make sure that the socket is no longer connected to a functional block, before unloading the LANA stack. After closing all sockets we can unload the protocol stack with the help of our script.

Example: **./scriptname down eth0**

The following points need to be emphasized:

- When compiling the Huffman kernel module, the old LANA version is needed which is located on the CD.
- Make sure to convert data to big endian before passing it to kernel space! (SW/HW)
- Use the defined headers! (SW/HW)

```
#!/bin/sh
# Huffman module initialization

if [ $1 = "up" ]
        then
        echo Huffman PF_LANA init running...
        cd /home/floriade/LANA/lana/src
        insmod lana.ko
        insmod fb_eth.ko
        insmod fb_huff.ko
        insmod fb_counter.ko
        insmod fb_tee.ko
        cd ../usr/
        ./vlink ethernet hook $2
        ./fbctl add fb2 tee
        ./fbctl add fb3 huff
        ./fbctl add fb4 counter
        ./fbctl add fb5 counter
        ./fbctl bind fb2 $2
        ./fbctl bind fb3 fb2
        ./fbctl bind fb4 fb3
        ./fbctl bind fb5 fb2
        echo ...done
elif [ $1 = "down" ]
        then
        echo LANA deinitialization running...
        cd /home/floriade/LANA/lana/usr/
        ./fbctl unbind fb5 fb2
        ./fbctl unbind fb4 fb3
        ./fbctl unbind fb3 fb2
        ./fbctl unbind fb2 $2
        ./vlink ethernet unhook $2
        ./fbctl rm fb5
        ./fbctl rm fb4
        ./fbctl rm fb3
        ./fbctl rm fb2
        rmmod fb_eth.ko
        rmmod fb_counter.ko
        rmmod fb_tee.ko
        rmmod fb_huff.ko
        rmmod lana.ko
        echo ...done
else
        echo Valid parameter are either up or down
fi
```

Listing C.1: Sample script for a Huffman stack

```
root@asusbox:/home/floriade/LANA/lana/usr# ./pflanacrrTX.sh up eth0
Huffman PF_LANA init running...
...done
root@asusbox:/home/floriade/LANA/lana/usr# cat /proc/net/lana/fblocks
eth0 vlink ffff88019b552840 1 4 [2]
fb2 crr_tx ffff88019b5527c0 2 4 [1]
ffff8801b65fed0 vlink ffff88019b552940 3 1 []
root@asusbox:/home/floriade/LANA/lana/usr# ./fbctl bind ffff8801b65fed0 fb2
root@asusbox:/home/floriade/LANA/lana/usr# cat /proc/net/lana/fblocks
eth0 vlink ffff88019b552840 1 4 [2]
fb2 crr_tx ffff88019b5527c0 2 7 [3 1]
ffff8801b65fed0 vlink ffff88019b552940 3 4 [2]
```

Figure C.1: Bind FB to a socket

```
root@asusbox:/home/floriade/LANA/lana/usr# ./test_crr_tx
Hit key!
```

Figure C.2: Waiting user space application

Appendix D

Content of the Attached CD

The following content can be found on the attached CD:

- The **final report** of this master thesis in Portable Document Format and its source
- All intermediate and the final **presentation** and their sources
- All **figures**,**plots** and their **sources** or **scripts** to generate them
- An **old LANA** version to compile the Huffman kernel module
- All of the **Huffman module source code** in C and VHDL
- All of the **CRR module source code** in C and VHDL
- The **state transition mechanism source code** in VHDL
- All **testbenches** and **scripts** used for simulation
- The **Huffman pCores** for the Network on Chip