FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

TCP/IP in hardware using SME

Meznik, Jan
pzj895@alumni.ku.dk
jan@meznik.dk

Jacobi, Mark Jan
dcz738@alumni.ku.dk
mark@jacobi.pm

August 26, 2019

Contents

Abbrevations

ALU Arithmetic Logic Unit. 7

ARP Address Resolution Protocol. 2

ASIC Application Specific Integrated Circuit. 6, 7

CPLD Complex Programmable Logic Device. 6, 7

CPU Central Processing Unit. 1

DHCP Dynamic Host Configuration Protocol. 2

DoS attack denial-of-service attack. 36

EEPROM Electrically Erasable Programmable Read-Only Memory. 7

FDDI Fiber Distributed Data Interface. 3

FIFO First In, First Out. 33

FPGA Field-Programmable Gate Array. i, 6, 7, 31, 32

FTP File Transfer Protocol. 5

HLS High-Level Synthesis. 2, 31

HTTP Hypertext Transfer Protocol. 5

ICMP Internet Control Message Protocol. 2

IEEE Institute of Electrical and Electronics Engineers. 4

IoT Internet of things. 31

IP Internet Protocol. 1, 4, 5

IPv4 Internet Protocol version 4. 2, 5, 24, 36, 37

IPv6 Internet Protocol version 6. 5, 36

NIC Network Interface Controller. 1, 31

RTL Register-transfer level. 2

SME Synchronous Message Exchange. ii, 2, 30, 32, 33, 36

SMEIL SME Implementation Language. 32

SMTP Simple Mail Transfer Protocol. 5

SoC System on a Chip. 6, 7

TCP Transmission Control Protocol. 1, 2, 5, 36

TOE TCP offload engine. 1

ToS Type of Service. 5

TTL Time To Live. 5

UDP User Datagram Protocol. 2, 26, 27, 35

VHDL VHSIC Hardware Description Language. 1, 33

Todo list

Chapter 1

Introduction

This thesis describes the design and implementation of an efficient, high-speed TCP/IP network stack intended to run on custom hardware where performance, responsiveness, and throughput is crucial.

As is the trend with modern automation, computerization, and mechanization, new devices are steadily invented to handle this increasing demand for data and control. With the ever-increasing sophistication of machines generating immense amount of information, the data needs to be transmitted to numerous other machines for further processing, or even simply storage. The most common and the most convenient way of linking multiple devices together is using the internet, and its underlying protocols. However, the networking stack supplied with most major operating systems, while heavily optimised, suffers from considerable penalties due to complexities of a standard computer architecture. For example, heavy network traffic utilizes the computers' internal busses, utilizes the memory, and spend precious Central Processing Unit (CPU) clock-cycles with polling and interrupts. This prevents the machine from using these resources for actual computing tasks.
These issues have been identified and solved by hardware manufacturers by adopting dedicated Network Interface Controller (NIC), which would employ various techniques to offload the processing. One such offloading technique is called the TCP offload engine (TOE), which usually takes care of the essential parts of networking involved – the Internet Protocol (IP) and the Transmission Control Protocol (TCP) [1].
Modern hardware manufacturers can produce NICs boasting network throughput speeds as high as 100 Gigabits [2]. Unfortunately, these cards are highly specialized for certain applications, and even though they provide basic programmability, they are rarely suitable for rapid prototyping of applications and other custom hardware devices. Furthermore, each NIC manufacturer has a diverse set of hardware with varying interfaces, making it hard to combine, swap and test these cards. Licensed software solutions in the form of IP blocks exist as well [3] [4]. Unfortunately, these blocks are usually distributed as black-boxes of VHSIC Hardware Description Language (VHDL) code, which is hard to maintain, and even harder to debug and extend [5]. Additionally, these IP blocks or ”cores” are exceptionaly expensive for smaller design teams, making it nearly impossible to prototype hardware designs with limited funding [5]. Multiple networking IP cores exist gratis for non-commercial use, but use for these is very limited by that nature. Bying a licence for such IP cores is not an easy task either, as the prices of the licenses are rarely listed on the vendor sites, and the sales departments have to be contacted for a custom quotation. It is also common to either require a lawyer to verify and sign the excessive license agreements, or having to sign up for memberships of various IP Licensing deals, such as the SignOnce IP Licensing program by Xilinx [6].

In this thesis, we bridge the gap between the blazingly-fast network offloading devices and their more flexible and malleable software counterparts.

This networking stack is implemented in a fully self-contained fashion so that it is completely independent of any other software running on the machine, while utilizing the performance advantages gained from the lack of overhead in conventional implementations. The use of a high-level programming language in combination with the modern SME model makes the network stack a very versatile implementation with ease of use, debugging, and even extension.

1.1 Related work

Networking is indeed desired in more and more applications and hardware devices. As a consequence, there exist projects that bring the internet protocol suite to the FPGA hardware. One such project is the Xilinx 10Gbps TCP/IP Stack, implementing a full TCP/IP stack inside the FPGA. It supports TCP and User Datagram Protocol (UDP) for data-transmission as well as auxiliary protocols, such as Internet Protocol version 4 (IPv4), Internet Control Message Protocol (ICMP), Address Resolution Protocol (ARP), and Dynamic Host Configuration Protocol (DHCP). This network stack is able to maintain a stable connection of 10Gbps, and higher speeds are easily reachable with better hardware [7].

The network stack has been implemented in C++ using the Vivado High-Level Synthesis compiler, generating the underlying Register-transfer level (RTL) model, greatly increasing the programmer productivity. To achieve performance in the High-Level Synthesis (HLS) C++ language, special loop pipelining and loop unrolling annotiations are used to transform loops from conventional sequentially-executed loops to fully parallel loops [8]. These annotations are not a part of the core C++ language, and they cannot be added generated automatically without the risk of creating race conditions and other bugs. As such, the loop unrolling has to be done by the programmer, and the performance of the finished product relies solely on the ability of the programmer to unroll and pipeline the loops efficiently and without errors.

The architecture of the Xilinx TCP/IP stack, seen on figure 1.1, is very similar to the one described in the implementation chapter.

4.

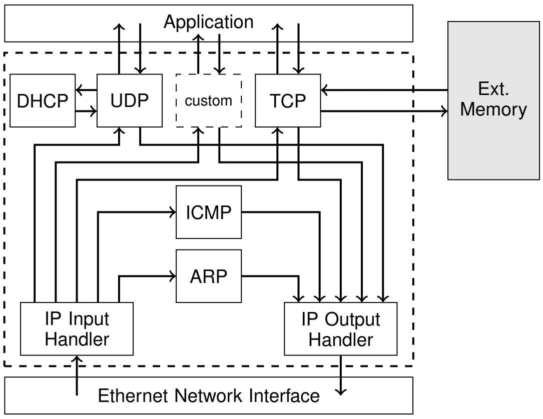


Figure 1.1: The architecture of the Xilinx 10Gbps TCP/IP stack introduced in *Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware, in FCCM'15* [7].

Fernando Luís Herrmann et al. have also created an UDP/IP network stack in FPGA, achieving 1960 Mbps full-duplex throughput through their UDP/IP network on a Spartan-3E XC3S500e-4FG320 FPGA [9].

With a design able to both send and receive 8 bits per clock cycle, this project was able to clock at $122.76MHz$, limited by the IP transmitter, which had the lowest speed. Interestingly, the whole system utilized only 14.19% of the resources on the modest Xilinx Spartan 3E, costing in the ranges from 150 to 500 USD [10].

Chapter 2

Background

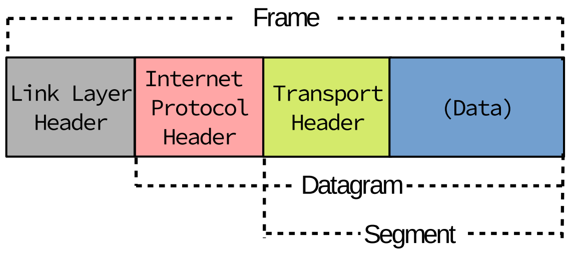


Figure 2.1: Layout of a typical frame sent through the network.

In this chapter, we will introduce the basic concepts of the Internet Protocol Suite, briefly describe its origin, semantics, and some of its protocols. Furthermore, SME and the hardware it will run on will be introduced as a basis for the implementation.

2.1 Internet Protocol Suite (TCP/IP)

Internet Protocol Suite, better known as simply TCP/IP, is a conceptual model providing end-to-end communication between computers. It consists of a collection of protocols specifying the communication between multiple Internet systems [11]. The very early research and development on what would later become the Internet Protocol Suite began in the late 1960s by the Defense Advanced Research Project Agency (DARPA), and was being adopted by DARPA, as well as the public, since 1983 [12]. Although the Internet Protocol Suite predates the newer, arguably more refined Open Systems Interconnection (OSI) model, TCP/IP still remains the popular choice in modern systems. As opposed to OSI 7-layer model [13], the collection of protocols in TCP/IP are organized into 4 abstraction layers, each related to their scope of networking involved.

2.1.1 Link Layer

The link layer is the lowest, bottom-most layer in the Internet Protocol Suite. Link layer addresses methods and protocols operating on the link that the host is physically connected to[1]. The purpose of this layer is to send and receive IP datagrams for the Internet Layer described in the next section.
Contrary to the OSI model, this lowest layer in TCP/IP does not regard the standards and protocols of the physical mediums used (the pin layout, voltages, cable specifications etc.), making TCP/IP hardware-independent. As a result, TCP/IP can in theory be implemented on virtually any hardware configuration, emphasizing the flexibility of the model.
TCP/IP supports many different link layers depending on the underlying hardware used, such as the Fiber Distributed Data Interface (FDDI) using optical fiber cables [14], RS-232 serial lines [15], or perhaps the most well-known link-layer protocol, the Ethernet.

Ethernet (II) and IEEE 802.3

The collection of standards known under the term "Ethernet" are were originally developed by Xerox Corp. between year 1973 and 1974 [16] [17]. Version 2 of the proto-

[1] Wireless connections are also included under this category.

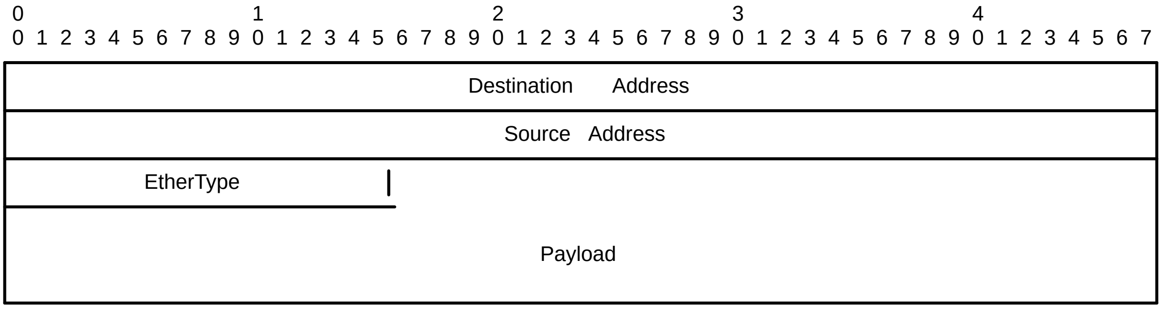


Figure 2.2: Layout of the Ethernet II header. The `EtherType` field is used as `Length` in the IEEE 802.3 specification.

Fix line from ethertype

col was published in 1982 under the name Ethernet II, also known as DIX Ethernet, named after 3 companies participating in its design: DEC, Intel and Xerox [18].
Seeing wide adoption of this standard, it has been formally standardized as by the Institute of Electrical and Electronics Engineers (IEEE) under the publication of IEEE 802.3 in 1983 [19]. Unfortunately, the original specification of Ethernet frame format deviate slightly from that of IEEE 802.3 [17]. Figure 2.2 shows the layout of a standard Ethernet header. The `EtherType` field is used to indicate the type of the payload type, whereas it is used as `Length` in the IEEE 802.3. Fortunately, none of the valid `EtherType` values in conventional Ethernet are valid as `Length` in the 802.3 standard, and so the headers are easily distinguishable.
The divergent way of encapsulating IP datagrams is defined in RFC 894 for true Ethernet [20] and RFC 1042 for IEEE 802 networks [21].

2.1.2 Internet Layer

The internet layer mainly concerns itself with sending data from the source network to the destination network. This seemingly simple task requires multiple functions from the layer:

- Addressing and identification
- Packet routing
- *Basic* transmit diagnostic information

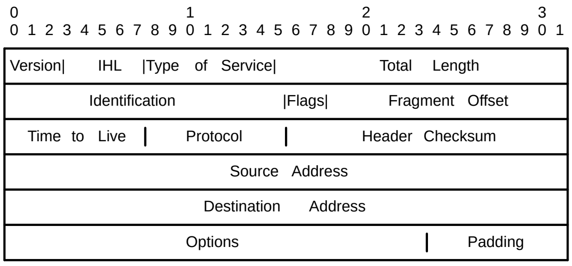

0 1 2 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
Version| IHL |Type of Service| Total Length
Identification |Flags| Fragment Offset
Time to Live Protocol Header Checksum
Source Address
Destination Address
Options Padding

Figure 2.3: Layout of the IPv4 packet header

Fix line from TTL, Protocol, etc.

- Carrying data for various upper layer protocols

Internet Protocol (IP)

The core protocol of the Internet Layer is the aptly named Internet Protocol (IP), which is the workhorse of the TCP/IP protocol suite. IP is a connectionless datagram delivery service based on the end-to-end principle, where the application-specific computation is located on the end nodes. It is often described as an ”unreliable” protocol, as there are no guarantees that an IP datagram successfully arrives at the intended destination. By being connectionless, IP does not maintain any information about the current state about the outgoing datagrams. As a consequence, IP is a best effort service, in which the protocol does its best to deliver a packet without additional knowledge of the next steps in the network [17].
The first major version, and the currently

most used version of the IP is version 4, usually referred to as IPv4. However, it is steadily being replaced by the newer Internet Protocol version 6 (IPv6), with an adoption rate of over 29% total users connecting to `www.google.com` using an IPv6 address by Jul 30, 2019 [22]. Despite the improvements that the new version provides, it has been proven to be a hard challenge to replace, update, and modify all the existing network devices to support this new protocol, extending the life current version 4. In this thesis, only IPv4 will be considered.

IPv4 provides two basic functions: addressing and fragmentation. By using 32-bit addresses located as one of the last fields in the header on figure 2.3, the nodes use the destination address to transmit internet datagrams towards their destination. The process of selection a path on the network is called routing [23].

Fragmentation on the other hand is used to split large datagrams and transmit these through nodes supporting smaller packets. The `Fragment Offset` in the packet header defines the offset of the data in the datagram with the `Identification` number.

IPv4 uses additional mechanisms to achieve its service:

- **Type of Service (ToS)**
  ToS is used to indicate the desired quality of the service provided. It provides options to minimize delay or monetary cost, or options to maximize throughput or reliability [17] [23].

- **Time To Live (TTL)**
  TTL is an upper limit of how many routers a datagram is allowed to pass. For every node visited, the TTL is decremented, and if the counter reaches 0, the datagram is destroyed. This ensures a reasonable routing by eliminating datagrams stuck in loops [23].

- **Options**
  Options can be added to the end of the header as a variable-list of optional information. It can specify additional functionality during the transmission, such as timestamping, routing options, security restrictions etc.. However, since it is not required to support Options in the IPv4 header, these are rarely supported and used [17].

- **Header Checksum**
  To ensure the correct routing and handling of the IP datagram, the checksum in the header assures the header-part of the packet is intact. Note that the checksum is only calculated for the header, and not the data, as this is done by most transport protocols.

2.1.3 Transport Layer

The transport layer establishes end-to-end data transfer between hosts. Protocols in the transport layer can provide additional services to the user, such as reliability, ordering, error- and flow-control, application addressing (port numbers), error-checking, and so on.

While it is possible to bypass the protocols in this layer on most modern network stacks, the protocols in the transport layer provide such essential and useful services that it hardly ever makes sense to implement in the application layer.

Transmission Control Protocol

While there are numerous protocols defined in the Transport Layer, perhaps the most well-known protocol in the stack is the TCP. Being one of the most used transport protocol for its reliability and congestion control systems, it is rightly justified to refer to the whole Internet Protocol Suite as simply "TCP/IP".

2.1.4 Application Layer

The application layer protocols are used by applications and services to exchange information over the network. A few of the well-known application layer protocols are the Hypertext Transfer Protocol (HTTP) [24], File Transfer Protocol (FTP) [25], and Simple Mail Transfer Protocol (SMTP) [26].

This layer is usually implemented by the user-space applications themselves, and therefore are not strictly required to actually run a TCP/IP network.

2.2 Hardware

The networking stack is intended to be flexible enough to run on just about any configuration of hardware and software. However, this also means that it cannot depend on any major external components, such as an existing memory, a processor, or any form of operating system. Fundamentally, not only the software-part of the networking stack has to be implemented, but the hardware needs to be defined as well. This hardware should be self-contained enough to work well in combination with any additional system, which the user incorporate for networking.
A wide variety of hardware types exist for such independent system, such as Application Specific Integrated Circuit (ASIC), Complex Programmable Logic Device (CPLD), System on a Chip (SoC), and FPGA. Each of these integrated circuits have their advantages and disadvantages; some of them are re-programmable, some are cheap and disposable, and some are excellent for general-purpose applications.

2.2.1 Field Programmable Gate Array (FPGA)

Field Programmable Gate Arrays, or FPGA for short, are devices containing integrated circuits (ICs) consisting of arrays of logic blocks. These logic blocks can be programmed to form arbitrary logic circuit by simply synthesizing a design and then loading it onto the board. This process alone can save the manufacturer months by not having to fabricate a whole new IC.
FPGAs can be used for any computational tasks without the need of any additional hardware. Usually, these devices are used for smaller, domain-specific tasks, where the control over the hardware yields significant performance increases. FPGAs are indeed very universal, and can be used in product-design, prototyping, as well as in final products. Products like car driver assistance systems [27], audio decoders [28], or even internet search engines [29] all utilize FPGAs to increase the performance, lower the electrical bill, and boost the development potential.

Technical specifications

Field Programmable Gate Arrays consists of a vast number of ICs, which can be reprogrammed at any time for a desired application or functionality [30], making the devices very flexible and extensible, even after manufacturing.
These ICs are practically totally independent, and their logic within can be programmed and combined in virtually any way with other ICs. This, however, poses a problem, as signals do not propagate through circuitry, immediately, but rather, they have a slight delay. Sometimes, two events precede each other, while other times, events of distinct timings must occur simultaneously. Since the order of events is critical for correct and expected execution in digital circuits, a digital clock is used to ensure everything runs in sync. A clock in this context emits a series of pulses in a pre-determined and very precise interval. These pulses are used to control the execution of various elements in the circuitry.
When synthesizing to a FPGA, the compiler finds the longest code-path, it finds the required circuitry to perform the calculation, and then it determines the minimal required time for the signals to propagate through the path. In this manner, the fastest possible clock can be found for that particular circuit.[2] With innovations and steady improvements in modern FPGAs, the circuitries within the devices can be clocked at higher than 500 MHz [31].

[2] Although many modern FPGAs consist of multiple regions which can have individual clock-rates. While it is a demanding task to propagate signals across these boundaries, a performance increases can be gained.

2.2.2 Application-specific integrated circuit (ASIC)

ASIC is an integrated circuit consisting of components such as transistors, capacitors, and resistors. During the manufacturing of an ASIC, the electrical design with the intended functionality is hardwired onto the board. This enables the devices to be very compact, efficient, fast, and low-power. Unfortunately, with this permanent nature of the board, the functionality cannot be changed after the fabrication, and the board has to be replaced in order to introduce new functionality [32].

The immutability of ASIC boards make them very inefficient for prototyping, but the bulk production once the final design is found can drive the per-device price down significantly. However, it is to be noted that some businesses still chose FPGA in finalized products in order to be able to patch and update the devices after the release, avoiding costly replacements if problems are found after the fact. [32]

2.2.3 Complex Programmable Logic Devices (CPLD)

CPLD are consists of a fully programmable AND/OR gate array and a set of macrocells. Macrocells are units prefabricated with higher-level functionality, such as Arithmetic Logic Unit (ALU)s, registers, and flip-flops. Since these macrocells are pre-made by the manufacturer, they can be heavily optimised in regards to both performance, but also power efficiency. [33]

CPLD devices are re-programmabla, but unlike FPGAs, CPLD are non-volatile since their programming is stored on Electrically Erasable Programmable Read-Only Memory (EEPROM). This means that the programming is kept on power down, and the functionality is active as soon as the device is turned on [33].

Unfortunately, with the introduction of the pre-made functionality in the macrocells, a bit of flexibility and extensibility is lost compared to FPGA. For instance, while it is easy to integrate FPGA with external memory chips, CPLD can only utilize built-in non-volatile memory.

2.2.4 Choosing FPGA for prototyping

In this thesis, only FPGAs will be taken into consideration for its re-programmability, its fairly low-cost, availability, and the compatibility with SME code-generators. During the implementation, the TUL PYNQ™-Z2 FPGA board will be used for prototyping. This board is based on Xilinx Zynq SoC, and is an excellent device for smaller projects given its wide assortment of documentation to get started prototyping. [34]

Programming an FPGA

Unlike conventional processors with a very sequential nature, the logic blocks in FPGAs are truly parallel in nature. Given the right programming, an FPGA can allocate dedicated sections of the chip for each independent subtask, enabling the circuitry to perform numerous independent calculations at once [30]. Unfortunately, this universality of FPGAs comes at a cost to their performance. Whereas conventional processors are heavily optimise based on the predetermined circuitry, FPGAs programmers must ensure to utilize the parallel nature of the device in order to secure best possible performance. Even worse, the FPGA must be programmed in such a way that all paths in the electrical wiring can be in any time-frame.

Due to this parallel nature of FPGAs, conventional programming languages are next to impossible to use. To define the behavior of an FPGA, Hardware Description Languages (HDL) are used. These programming languages are not easy to learn without a good grasp of electrical engineering. Even with prior programming knowledge, the unusual approach to concurrency in these languages can be hard to understand for average developers.

To simplify the development process, most manufacturers offer predefined circuits along their FPGAs. These predefined circuits are more commonly known as Intellectual Property (IP) cores, and can provide the hardware designers with pre-made circuitry for a wide variety of functionality. While most IPs provide the functionality of processors for testing on an FPGA, mp3 audio decoding or PCI bus interconnect can be obtained as well [32].

2.3 Synchronous Message Exchange

The Synchronous Message Exchange model (SME) is a messaging framework created in order to help model hardware descriptions [35]. It was conceived once the flaws of using Communicating Sequential Processes (CSP) was identified during the modelling of a vector processor with CSP using PyCSP [36]. It turned out that there is a major discrepancy between the way data is propagated in hardware opposed to that of the CSP model. While CSP does not pose any requirements on the communication between processes, in digital hardware, all communication has to be synchronized, driven by a clock. To combat this in the CSP model, a global clock process needed to be implemented, which was connected to all other processes. Additionally, latches had to be introduced in order to not overwrite values during a cycle. This caused an explosion of both channels and latches in the final design, making CSP a much less viable framework for hardware modelling [35].

2.3.1 The model

The SME model consists of only a few fundamental concepts. Each SME model is a *network* consisting of one or more *processes*. These processes do not share any memory or storage, but are interconnected with *busses*. These busses are perhaps the most interesting units in SME model, as they not only propagate information between processes using the underlying *channels*, but also introduce an implicit clock between the processes.

2.3.2 Process execution flow

The execution flow of a process is fairly simple, and relates very closely to that of the actual hardware. At the beginning of a clock-cycle, the input-ports are read into the busses they are connected to. Then, the process executes its ”compute” stage, and the results, if any, are written to the output-port, which will be read by the following bus. A visualization of the execution flow can be seen on figure 2.4. It is important to note that although certain channels might be written earlier than others in a process clock, the subsequent processes connected to said bus will first see the values change in the beginning of the next clock cycle.

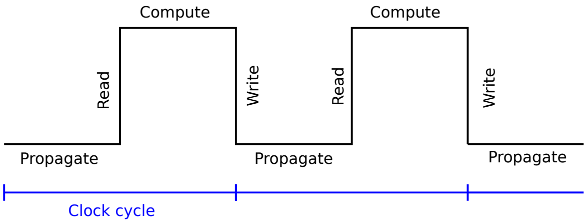


Figure 2.4: An illustration of a typical SME clock-cycle

2.3.3 Using SME

SME has undergone multiple iterations, reworks, and extensions. While it is still under very active testing and development, its core functionalities and features are well-established and stable [37].
SME has concurrent implementations in the C# and Python languages, with promising efforts to unify these under a common intermediate domain-specific language SMEIL [38]. The C# version has exhibited various advantages over the Python counterpart, such as the more error-prone strong typing system, which better reflects the functionality of the hardware, as well as making the code more readable to the programmer. At the time of writing, the C# implementation currently enjoys the most recent features of the SME model, as it is being the most actively developed version.

Chapter 3

Design

3.1 Overview

The networking stack introduced in this thesis is implemented in the C# programming language with SME. The aim of its design is to capacitate performance, flexibility, and ease of use. In this chapter, the design principles are described, the architecture of the solution is outlined, and the components are outlined.

3.1.1 Design principles

As briefly mentioned in the introduction, the proposed network stack is to provide an alternative to the existing proprietary network offloading engines. While the main goal of this thesis is to research and study the suitability of SME for implementing a TCP/IP stack on an FPGA, there are many other aspects of the system to be studied.
The extensibility of the network stack are to be tested by studying the effects of introducing new protocols to the stack. While the network stack should be able to be refined with new and custom protocols, it is to be studied which implications it has for the system. Mainly, it is to be seen how the addition of new protocols affect the performance, scalability, and viability of the system.
In the same vein, the design should be as FPGA-agnostic as possible. While this is mainly guaranteed by the SME framework used to develop the system, the underlying systems, operations, and features should be easily portable across FPGA manufacturers.
Lastly, the design of the networking stack should be interoperable with other systems on the FPGA, or even FPGAs. It is to be seen how easy it us to modify and extend the versatility of the system without any major modifications or even extensible knowledge of the system. As an example, the networking stack can be expanded with a firewall, developed alongside this project.

3.1.2 Initial requirements

Following our design principles, initial requirements and goals for the networking stack are set so that these can be tested and improved upon.

- **Essential protocols only**
  Considering that the SME project is still fairly early in its development, and considering the sheer number of protocols in the internet protocol suite, the networking stack in this thesis is to support only the absolutely essential protocols required to provide the users with a meaningful interface to the internet. These protocols should be picked such that the system can provide the end-user with a network data-stream, which can transport information to and from a remote computer.
  The initial protocols chosen may be implemented and supported partially, but they must not deviate from the standard specifications.

- **Support an interface for the end-user**
  The system must be controlled by an end-user on the FPGA. Such an interface is very unique in its own way, compared to standard software interfaces,

like the ones defined in the POSIX collection of specifications. By supporting such an external interface gains insight in the way such a networking stack will be used, and which measures must be taken in order to provide the best possible integration and performance considerations.

- **Independent of underlying physical hardware**
  By using SME, the underlying hardware description language code can be abstracted away from the actual implementation. This will later provide developers to easily modify and tweak the networking stack without having to consider the target hardware.
  Likewise, the networking stack may not rely on using a certain physical layer hardware, and must be designed to be independent of the underlying hardware used for the physical connections. This will ensure that the target hardware can easily swap between physical connectors, such as going from ethernet cables to wireless, or even another FPGA.

3.2 Initial design

The initial architecture had a very simplistic approach to its design in order to aid with early identification of potential issues and problems. The basic idea of the initial design is to minimize the number of memory operations carried out. Under the assumption that the Ethernet interface used in the network stack will likely have limited memory, everything needs to be copied directly to the stack. Figure 3.1 shows the initial design, where the leftmost module Ethernet connects to the memory for direct access, and each parsing layers listens to this connection and performs accordingly. Each parsing layer also connects to the subsequent layer in order to flag when the subsequent layer should start listening on the data-bus (that is, where the current packet header stops and the next header begins).
An advantage that this design provided was the cetralized memory, which is much easier to up-scale in terms of capacity and bandwidth. This global memory would also be able to modify packets in-place, removing duplicate data, minimizing the need to copy data around, and making it easier to keep track of the memory fragmentation.

3.2.1 The issues

As anticipated, this initial design brought fourth the main issue fairly quickly in the implementation phase, where most of these stem from the differences in programming hardware as opposed to the software network stack, from which the inspiration was drawn.

Internal parsing buffer or memory is largely unavoidable

Although listening to the global databus and processing on the bytes currently therein seems like an efficient way of minimizing the data-transfer required across processes, it has shown to yield some unavoidable challenges.
Parsing fields in a packet-header is much more cross-dependent than initially anticipated; each field might have numerious implications on the way preceeding and subsequent fields are read and interpreted. As an example, in the Internet Control Message Protocol (ICMP), a redirect message type has an IPv4 address field in the header, whereas in the timestamp message type, this field is interpreted as both an identifier and a sequence number. This sort of interdependency is hard to parse without the ability of caching or buffering the header locally in the parsing process.

Overutilized memory module

While the ethernet is the main writer to the memory module, the parsing layers need to access and write to the module as well. At the very least, the memory module would have 6 connections in the network, not counting any additional components, such as user interface, firewall, etc. Although numerous memory implementations exist on the FPGA landscape, Block RAM (BRAM) seems to be the most suitable in

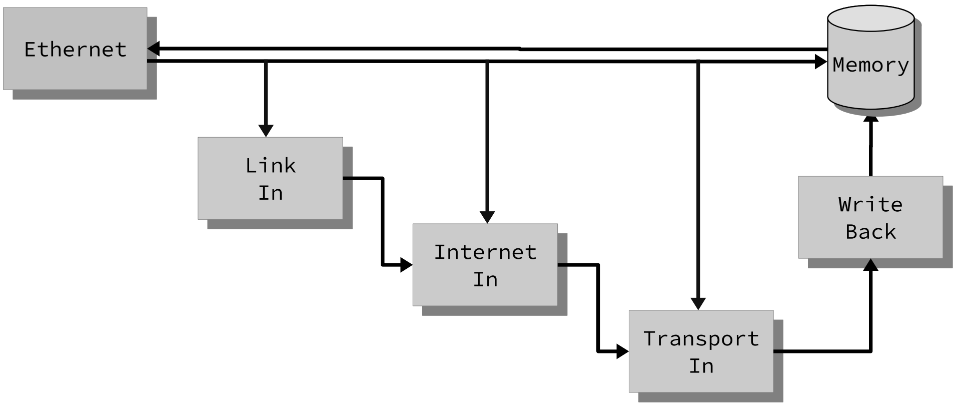


Figure 3.1: The initial design

this situation for its speed and latency. Unfortunately, many widely used block RAMs only have 2 simultaneous connections (or ”ports”) at the same time. Additionally, the block rams are frequently limited to only operate a few bytes of data at a time. Although some block RAMs, such as the ones found in Xilinx FPGAs can be cascaded [39] to lessen the impact of these limitations, this hardly provides a good basis for a scalable design.

Data fragmentation and memory management

Another problem a unified address space in the global memory is how costly basic memory operations, such as moving or copying, become. The initial assumption that packets stored in memory could be reused by modifying them in-place and send them turned out to be misguided, since the layout, size, and the number of the outgoing packets very rarely resemble the in-going packets.
Furthermore, the general purposeness of the memory makes it very hard to structure. Without very complex memory management, the memory can get very fragmented and slow.

3.3 Revised design

The initial architecture focused heavily on the input from the link interface, minimizing hardware memory requirements, and to minimize the latency from the source data-stream to its respective layer handler. Unfortunately, the opposite revealed to be true, as the overly-utilized memory unveiled plentiful issues to the performance.

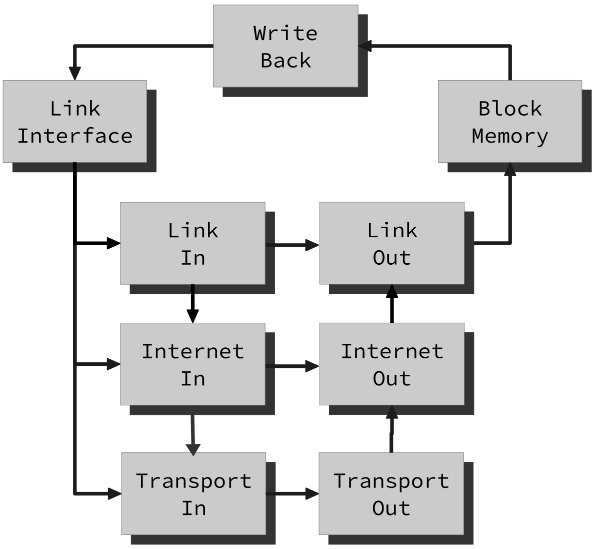


Figure 3.2: Second iteration of the design

To avoid the problem with global memory, the data needs to ”flow” through the components that need the data at that time. Therefore, as opposed to the initial design, where the link interface writes to memory directly, in the revised design, this connection is completely removed. The idea is to let all the data pass pass through each state, letting each stage parse the required information, and passing the rest to

the next components.
As seen on figure 3.2, the link interface, which provides the raw byte-stream from the network, is connected to all of the input parsing layers. The layers are connected in the order in which a network frame is parsed; link- to internet- to transport-layer. This approach aims to utilize the fact that the layers can act immediatelly upon the packets received directly from the source, avoid having to buffer the whole packet in each stage, as well as easing the logic required to buffer the data across the layers.
This design starts by the Link Interface sending one byte at a time through its bus. The `Link In` will parse the first header, and signal the next layer upon completion. `Internet In` will then start to listen on the `Link Interface` bus and, using the information from `Link In`, parse the internet header accordingly. The same procedure would be applied to the connection between `Internet In` and `Transport In`.
When data is to be sent to the internet, the network frame would be built bottom up from the transport layer through internet to the link layer.

3.3.1 The issues

The issues quickly surfaced during the implementation of this revised design. Although the interconnect from the `Link Interface` to all the subsequent layers in parallel promised negligible latency, it came with a great cost to the solution.

Process under-utilization

Since each "in" process has to wait for the previous layer to signal when to start listening on the data-bus, the layers would in average only be active a third of the time. Since each layer has very little information about the states of the other layers, it would become a challenge to get any other work done during these phases.
For example, it would be an immense challenge to coordinate an ICMP reply on a faulty packet in the `Internet In`.

Redundant Link layer

While the Link layer is an essential part of the Internet Protocol Suite, it did not fit well with the functionality of the rest of the stack. Most network interfaces are equipped with buffers, on which integrated circuits perform operations such as error check using cyclic redundancy check, denoising, timeslot management, etc. Likewise, the Pmod NIC100 Ethernet interface has built-in controller with internal memory suited for buffering the incoming packets [40]. This memory, apart from the cyclic redundancy check, can be used as the initial step for parsing the packet, and only send the datagram to the stack.

IPv4 fragmentation and out of order TCP packets

The chaotic nature of internet routing might cause packets to come out of order, or even get fragmented along the way. Since each layer parses the packet immediatelly as it is written to the bus, it became a challenge for the layers to figure out what to do. On IPv4 fragmentation, if the second half of a dataframe arrived first, the Transport header would not be available to the Transport layer. Although IPv4 fragmentation is an increasingly rare phenomenon, the network design is not able to handle the situation well.

TCP connection state sharing

With a clear separation between the "in" layers and the "out" layers, the Transport block had to be split as well. Unfortunately, unlike the other stateless layers, the transportl layer actually needs to keep track of the connections and their states. On every segment received, the appropriate connection needs to be updated accordingly.
In the TCP protocol, the connection state changes on both receiving and sending. In this case, the `Transport In` and `Transport Out` have to agree on a shared state. As these states can be quite large, and the should support multiple connections at once, one large bus containing all the information is not feasible. To solve

this, a negotiation protocol may be introduced, however, as pointed out in item 3.3.1, the processes are very limited in their execution time. A negotiation would be very hard to achieve in such circumstances.

Problematic order of building and sending outgoing packets

Outgoing packets are built in the reverse order of which they are parsed – the inner layers are built first, and then the packet grows outwards by adding new layers on top of the existing one.
That in itself is not a problem, as the packet can be easily passed through the network backwards from the last byte first. That is, the inner-layer of a packet is passed on to the next layer first, then the header packet header is built, and lastly, the header is passed on backwards.
However, this assumes that the next layer to process the packet is available at all times in order to process the packet. This is certainly not the case, as layers such as `Link Out` and `Internet Out` might be in the process of sending out their own protocol-specific control messages (such as an ARP annoucement in the Link Layer or ICMP reply in the Internet Layer).
A negotiation protocol can be implemented between the outgoing processes in to postpone the outgoing packet, but these structural hazards, as known from conventional processor pipelines, add a lot of complexity to the system.

3.4 Pipelined design

While it would be possible to work around these identified issues with the revised design in the code, the added complexity would have additional ramifications on the project as a whole. Upon further analysis of analysis, it is clear that the source of the issues is the parallel arrangement of the process blocks.

The next iteration of the design utilizes a fairly standard approach to pipelining, albeit with an unusual transfer of data between the stages.

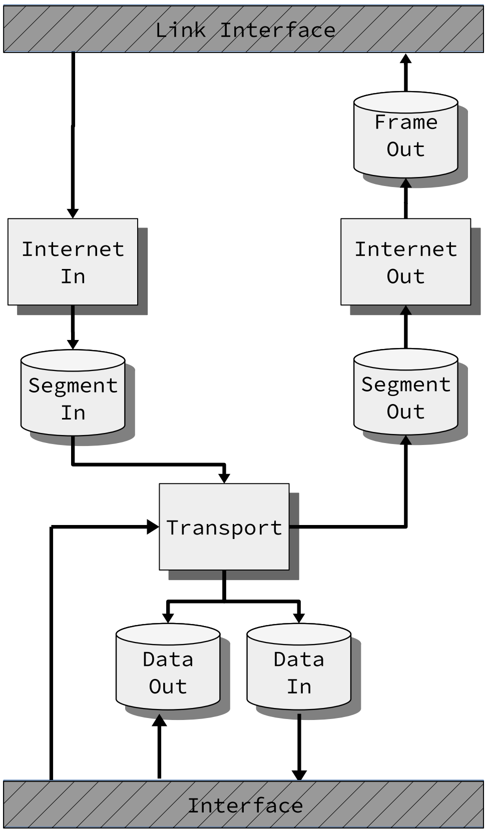


Figure 3.3: The final design. The rectangles represent ”compute” processes, while the cylinders represent the buffers.

Fix Link Interface to be the same as Interface

The idea with the pipeline is to enable the processes to receive, compute, and forward data at their own pace, without any major limitation from the other parts of the system.

3.4.1 Internet layer processes

The processes performing computation and processing on the actual internet packets, called "layer processes" for brevity, are by large kept intact from the previous design. The fairly simple, but highly sequential nature of packet header parsing turned out to be very complicated to optimise with the additional computing power of the hardware, without introducing too much complication.
Missing from the updated figure 3.3 are the `Link In` and `Link Out` processes, which, for now, are handled by the ethernet interface, which can easily parse and strip the first frame headers.

3.4.2 Busses

The busses in the revised pipelined design are devised such that communication is only limited to the immediate neighbours in the logical network. This design is put in place so that the synchronization and the order of execution is much easier to keep track of, so that fewer race-conditions occur, and so that blocks in the network are easier to replace, remove, or modify, without having a large cascade effect on the whole network, as opposed to only the neighbours.
Whereas in the previous design where the busses would simply write new data on every new clock-cycles regardless of the reading processes, now there must be some logic to actually ensure that the reading process is ready to receive new data. While the Data Buffers, as introduced in the next section 3.4.3, solve the issue of blocks reading and writing data at their own pace, the busses must support an interface for sharing the state of both processes. Thus, while the busses are depicted as directional with arrows in figure 3.3, there is naturally a need for a control signal in the opposite direction to control the data-flow. This communication protocol will be discussed further in the implementation section 4.3.

3.4.3 Data buffers

Illustrated as cylinders on figure 3.3, First-In, First-Out (FIFO) buffers are introduced between each parsing process in order to control the data-flow between the layers. Apart from maintaining a fairly large memory bank through the block-RAM, these buffers also contain logic to store the incoming data intelligently in order to offload the following processes. For example, the `Segment In` buffer ensures that fragmented IPv4 packets are defragmented before leaving the buffer. However, introducing a new "type" of a process — the buffers — poses a new challenge. While the buffers can be read from at any time, the layer-parsing processes do not have this luxury, as they do not have any significant internal buffer. Here again it becomes obvious that a handshake protocol needs to be used in the bus-communication between the buffers and the processes.

Order of outgoing packets

In the previous design, it was obvious how sending packets might become a challenge given that all processes involved must be ready to promptly operate on the outgoing packet. With the introduction of data buffers, the processing of an outgoing packet can be delayed. However, there lies another small problem with the way data is passed around. The `data`-section has to be read by Transport in first-in, first-out order, solely for the reason that Transport does not know beforehand how much data to send, nor whether more urgent tasks appear during the transmission. Figure 3.4 shows the possible ways of creating and passing a packet along the network stack. On the right side of figure 3.4, the data is sent from the last byte first, indicated by the red arrow from bottom up. Then, the transport-, internet-, and lastly, the frame-header is built and sent with it in the same order, from the last byte of the header till

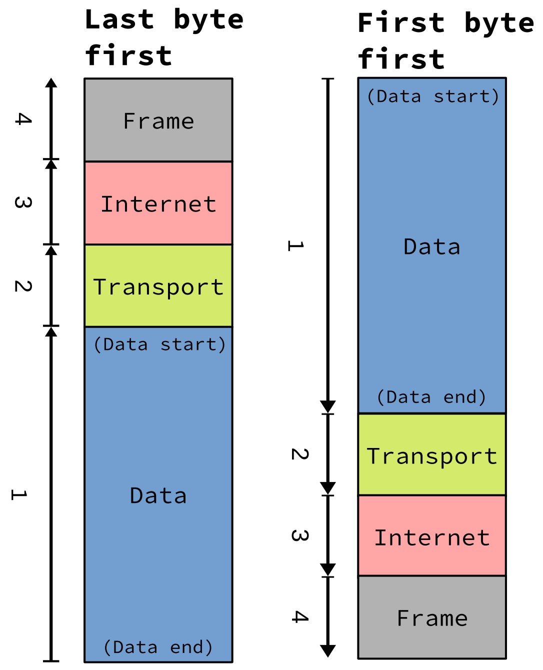


Figure 3.4: Possible orders of ways of building an outgoing packet

the first. While this method is fairly simple, it has two problems that are hard to work around – firstly, the user might be in the process of writing outgoing bytes to the `Data Out` buffer, in which case, Transport cannot beforehand start sending the last byte. The second problem is that, although Transport wants to send as much data at once as possible to lessen the overhead from the rest of the package, more urgent tasks might come up during the transmission, yet the operation has to finish reading the last (first) byte.
The left side of figure 3.4 shows another approach to sending a packet. Here, the data portion of the packet is read as is in the first-in, first-out byte order. The headers are written afterwards, also in the FIFO order. It is important to notice that even though the package looks structurally right when reading backwards (the frame header is in the beginning, then Internet header and so on), the ordering of the bytes is not correct! For example, if reading the right-hand on figure 3.4 packet in reverse order, the data section would begin with the very last byte!
Here, the intermediate buffers once again offload this problem, as one of the brilliant features is that they enable the system to pass bytes out of order. In that case, the sending process can begin in the middle of a package, and then append to the beginning of it at the very last.
Figure 3.5 shows the building of an outgoing packet. Between each process, the colored block represents the state of the packet being passed along the processes. The red arrow indicated the first stage of the connection between processes, where data is passed from a previous layer. The blue line indicates the data that the process itself appends to the stream.
The implementation of this out-of-order forwarding will be described in-depth in the implementation chapter 4.

3.4.4 Interface

Lastly, the Interface is designed so that the end users and system can utilize and use the network stack.
The networking stack is controlled with 3 connections (consisting of bus-pairs): the control-, the read-, and the write-connection. While the last two connections are incredible simple and only transfer data from the `Data` buffers, the last connection controls the whole network-stack.

Read/Write interface

In conventional programming languages, the user would usually supply the a function call with an array or a list on which the function can operate. However, in hardware, this is generally not the case, as arrays are costly to transmit at once.
Therefore, the two read and write interfaces would simply stream one byte at a time, and it is up to the user to be prepared to read or write the data.

Interface Control Bus

The Interface Control Bus controls all the ”business” logic of the network – maintaining the active connections, starting and closing connections, and various other protocol-specific control on each connection.

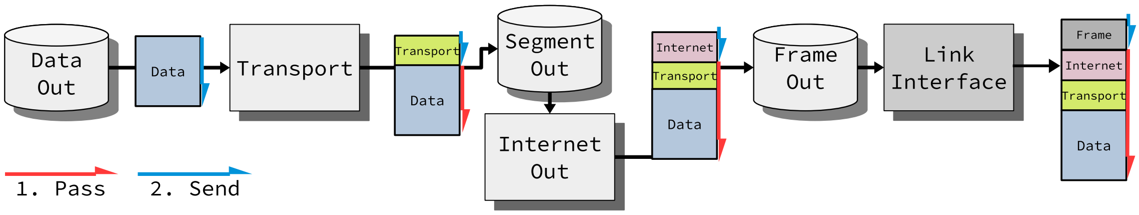


Figure 3.5: The order in which an outgoing packet is built and passed through the network pipeline. The colored boxes represent the state of the outgoing packet, the red arrow indicates the first stage of forwarding, and the blue line indicates the last stage.

Currently, all this can be handled by the `Transport` block which has all the necessary information to handle the interface requests.
The design of the Interface is based on the widely adopted Berkeley Sockets library, which saw the first implementation in 4.2BSD, and has been defacto a standard component in the POSIX specification [41]. There are multiple reasons for this decision:

- The Interface will feel more familiar to users accustomed to the Berkely Sockets API commonly found in mainstream systems such as Linux, OSX, and BSD variants.

- The inner workings of the stack will be more transparent, and the API exposes fairly fine-tuned control over the whole network stack.

- Even with the relatively few functions exposed, the API has thrived in the most used systems as of now. It would be an understatement to say that the Berkeley Sockets API have stood the test of time, and therefore, it is a good basis for the interface used in the network stack.

The first version of the stack should support the following functions:

`listen(protocol, port)`
Finds and initializes a free socket with the given protocol and port. This socket is immediatelly put into listen mode. Returns error if protocol is not supported, if port taken, or if no free sockets are available.

`connect(protocol, ip, r_port , l_port)`
Connects to a remote endpoint on `ip:remote_port` using `protocol`. This call is used mainly by connection-based protocols that need to establish a connection before exchangnging data, although it can also be used by datagram-based protocols as a way of setting the default destination to send subsequent data to. Returns error if protocol is not supported, if no free sockets are available, or if the optional local port is taken.

`accept(socket)`
Accepts the pending connection and sets up the underlying socket state accordingly. Note that unlike the POSIX implementation of `accept`, which returns a new socket with the connection, this implementation changes the state of the current, listening, socket. Returns error if no pending connection to accept, or if invalid socket supplied.

`send(socket, byte, [ip], [port])`
Queues a byte for sending through the socket. An optional IP address and port can be specified in certain connectionless protocols. Guaranteed to succeed, given that the transport-bus can be written to. More on this discussed later.

`recv(socket)`
Reads (or ”receives”) a single byte from the socket. The appropriate error code is set if no byte available on that particular socket.

`close(socket)`

> Closes the connection on `socket` and frees the socket for further usage. Calling on an already closed on non-existent socket has no effect. Guaranteed to succeed.

While the arguably most essential functions have been defined, there are some functions from the Berkeley Sockets API that have been omitted for purely technical and practical reasons.
The function `socket()` is mainly used to allocate and create new sockets in an environment, but given that the hardware network stack has static allocation of the sockets, it is not needed.
Additionaly, the `bind()` function is also missing for the sole reason that in the current implementation of the network stack does not have any valid reason not to bind a socket immediatelly.

Chapter 4

Implementation

In this chapter, the implementation of the network stack using the pipelined design from chapter 3 is outlined and described, the application of SME detailed and evaluated, and lastly, the viability of the system on an FPGA is discussed.
The network stack is implemented in C# using the C# version of SME, which is, at the moment of writing, more mature and feature-rich. The current version of the implementation supports most of the absolutely vital parts of the IPv4 protocol, as well as the UDP protocol, as specified by RFC 1122 [11]. Although work has been carried out in order to ensure that additional protocols can be implemented without obstructions, no additional protocols are supported at the moment.
The solution is fairly well-divided into 3 different types of components, relating closely to those of SME: processes, buffers, and busses. The most interesting parts of these components will be described in further detail in the following sections.

4.1 Processes

The processes are arguably the most vital part of the system, as they provide the computation and "processing" on the in- and out-going packets. It is important to note that although there are many other types of "processes", in the network, such as the buffers, we will mainly refer to the modules doing actual business-logic as "processes" [1].

The essential processes in the network are represented as light-grey boxes in the figure 3.3. These processes are `Internet_In`, `Internet_Out`, and `Transport`.

4.1.1 State-machines

Network communication can consists of countless different packets, formats, protocols, combinations of flags and settings, and even errors and corrupted bits. The processes in the network have to take on a manifold of jobs in order to handle all these scenarios, which sadly cannot be handled with a simple combinational logic circuit. To operate under these various conditions, these processes are modelled as finite state machines, maintaining a single state at all times.
The processes have a lot of similar states, such as `Idle`, `Receive`, `Pass`, or `Send`, but these can work very differently, as shown in the following sections. Before moving on to describing the state-machines of the 3 processes, it is crucial to understand how these can be modelled in SME.

SME process execution flow

To implement a process in SME, the C# class has to inherit from either the `Process` abstract class, or the more simple `SimpleProcess` class. The latter class is, as its name states, a simpler version of the former. This class implements an `OnTick()` method, which is invoked once for every clock-cycle.
The more advanced, but also more capable `Process` class provides an abstract method

[1] These processes are not to be confused with SME processes, which are used for the implementation of both the buffers and processes.

Run which is to be overriden and filled with the code desired to be run in the process. The interesting feature about this method is that it is asynchronous, meaning that the code can execute other tasks while waiting for resources, such as functions, to return. In this case, this asynchronous feature is used to give the programmer ability to split the function into multiple segments, separated by the clock signal.
Figure 4.1 compares these two approaches for the same finite state-machine with 3 consequtive states.

The "synchronous" approach using a SimpleProcess in subfigure 4.1b has to implement a variable tracking the current state of the process. On each new clock, this state has to be analysed and the inteded function to be called based on the value. This approach requires a lot of approach and boilerplate code, especialy if there are several states.
The asynchronous approach on subfigure ?? on the other hand can do with only single Run() method split into three parts – A, B, and C. After each code-segment, the process waits for the clock signal, and continues with the execution of the next segment. This functionality gives the programmer a very granular control of the way a process works, how it is split into multiple steps on the hardware, while maintaining simplicity, as seen on the statemachine diagram on subfigure 4.1c.

Internet Out state machine

This way of modelling a process in SME first the Internet Out process very well, as it only has one responsibility, which is reading outgoing segments and wrapping them in an Internet header. The figure 4.2 shows the pseudo-code and state-machine for the Internet Out process. This process was easy to model and implement, because it only has one input and one output, and the state-changes are simple and intuitive.

Internet In and Transport state machines

Unfortunately, The state-machine of Internet In is probably the most simple of all the state-machines, as it can effectively only read new packets from the Link-layer, and pass it along the pipeline. Although it might be desireable for the Internet layer to send control packets out to the network, this is not supported in the current build.

Internet Out state machine

4.1.2 Internal memory

4.2 Buffers

Problems such as packet fragmentation and out of order insertion are solved in the memory buffers.
All of the buffers needs to handle input and output of memory at the same time. If not, the system would slow down by a factor of two, since the input would need to wait for the output to finish submitting, and vice versa. This requires the underlying memory to handle read and write at the same time.
Each of the memory buffers have slightly different variations. In general, there are several problems to solve:

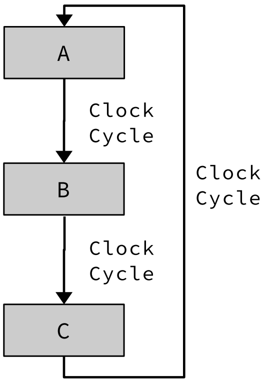
- **Fragmentation**
  In Segment In and Data In there are segmentation. In Segment In IP fragments may arrive out of order, and is therefore only sent through the system when all segments are received. Data In is essentially the same, just with other protocols such as TCP.
  In both cases we want the order to be concurrent. Packets that are not fragmented is in FIFO order. Fragmented packets in Segment In are held back, until all segments are received. Then all data segments are sent as a single block to Transport. In Data In, the segments are handled as a stream, where the last valid segment ID is known. The buffer only sends data when the last valid segment is set.

```
public class SomeProcess : StateProcess
{
  private override async Task OnTickAsync()
  {
    // A
    await ClockAsync();
    // B
    await ClockAsync();
    // C
    await ClockAsync();
  }
}
```

(a) Example using inheriting from `Process`, using the `Run()` asynchronous method.

```
public class SomeProcess : SimpleProcess
{
// Initial state
state = A;

protected override void OnTick()
{
  switch(state) {
    case A:
      a();
      state = B;
    case B:
      b();
      state = C;
    case C:
      c();
      state = A;
  }

}
```

(b) Example pseudocode of using a ”synchronous” `SimpleProcess`

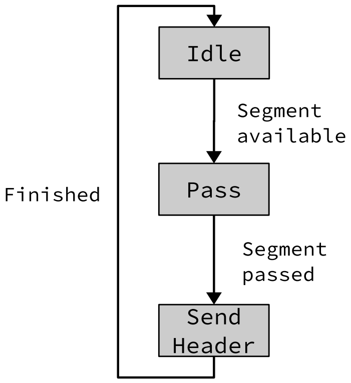


(c) The statemachine resulting from both code-examples

Figure 4.1: A simple state-machine implemented in the asynchronous (left) and asynchronous (right) approach in SME using C#

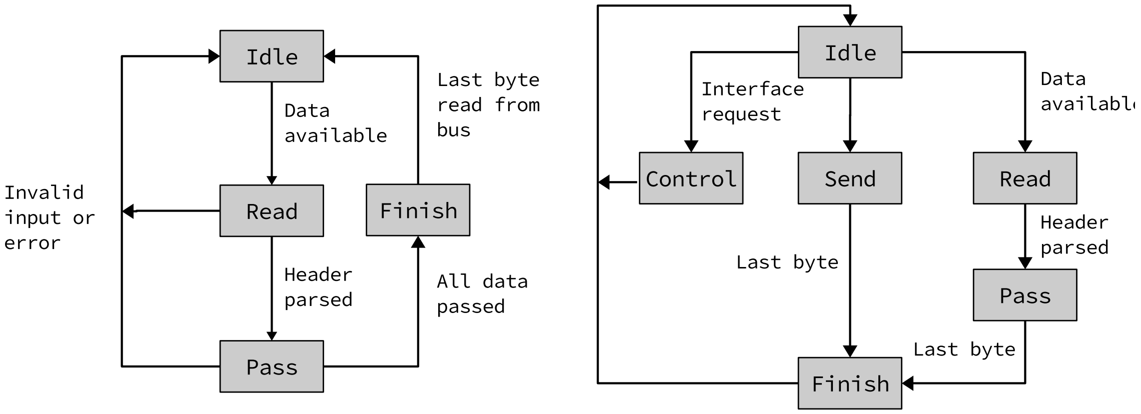
```
public partial class InternetOut: StateProcess
{
public override async Task OnTickAsync()
{
  while segment_available() {
    pass_segment();
    await ClockAsync();
  }

  while header_available() {
    pass_header();
    await ClockAsync();
  }
}
}
```

(a) Pseudo-code for the `InternetOut` process



(b) The statemachine for the `InternetOut` process

Figure 4.2: The implementation of the `InternetOut` process



(a) The `Internet In` state machine

(b) The `Transport` state machine

Figure 4.3: Statemachines for `Internet In` and `Transport`

This way, the buffer can contain segments with an higher ID than the valid, and know not to send it.

- **Unknown size**
  Some of the buffers do not get information about how much data has to be allocated. This happens in `Segment Out` and `Data Out`. In `Data Out` we do not know how much data the user sends into the system. (see: section 3.4.4). In `Segment Out` we do not know some of the packet sizes, since `Transport` may not know how big the packet is going to be, for example when constructing an response packet.

- **Out-of-order submission**
  In Some cases the header of an packet can only be created after the data has been received. An example of this is the calculation of the checksum.
  This feature is required on `Frame Out` and `Frame Out`. This only applies to buffers where data is sent out and where calculations are done.

- **Data ready**
  When The buffer indicates that data is ready to be read, the next clock should also be ready and contain data. If not, the consumer would request data, wait at least 2 clocks for the data, and then request new data, slowing down the data transfer process significantly.

An oversight of this can be seen in Table 4.1.

	Fragmentation	Unknown size	Out-of-order submission	Data ready
`Frame Out`			✓	✓
`Segment In`	✓			✓
`Segment Out`		✓	✓	✓
`Data In`	✓			✓
`Data Out`		✓		✓

Table 4.1: The requirements for the buffers

4.2.1 Components

This section describes the components briefly to give a better overview of the general structures of the buffers. The specific components are described in detail in the following chapters.
To solve fragmentation the system uses ”segments” in the memory. A segment is an abstract structure consisting of metadata and two memory addresses pointing to the start and end of the actual data.
To handle these segments, two interfaces are created, one to handle fixed size allocations, and one to handle dynamic allocations, where the size is unknown. Out-of-order submissions are solved simply by having an address sent beside the data, which

are added to the beginning of that respective segment. See subsection 4.2.2 for a in depth explanation.
To keep order of the segments, a simple directory of keys and linked lists are needed. The linked list are ordered at insertion time, so the fist element always is the smallest. This makes lookup to the next element constant time, and insertions of new at most $O(n)$ time. See subsection 4.2.3.
To have the data ready, a small internal buffer is needed. This small buffer starts filling as soon as a segment is ready. See subsection 4.2.4.

4.2.2 Memory segments

The memory segment structures consists of two types width slightly different implementations. These are called Multi memory segments and Single memory segments. Their differences are explained in the end of this section.
Both types consists of an lookup table, where each table entry contains the meta data and head and tail pointers. A Illustration of this can be seen in Figure 4.4
In the implementation, the memory is not actually read, but an address is returned. This makes it possible to use any memory type, since the latency issues are in the submission and retrieval of the data itself, but not the calculations for the address.
The lookup table consist of multiple segments references. A segment reference contains meta data(Not illustrated in the figure) and a start and stop pointer.
In the illustration segments 0,1 and 4 are used, and 2 and 3 are free. Note that segment 4 wraps around. If address 2 is requested from segment 4, memory address 15 is returned. If address 3 is requested, memory address 0 is returned.

Multi memory segments

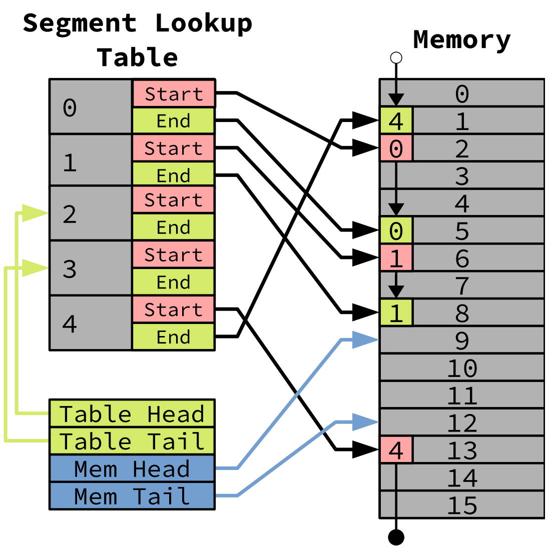


Figure 4.4: The general structure of the memory segments in use.

Each segment has three states, indicated by two boolean values; `Done` and `Full`. When none are true, the segment is currently being filled with data. When it is filled, and ready to be read, filled is marked as true, and the data can now be read from. When the reading has ended, both `Done` and `Full` are set to true, and the segment is now ready to be reused in the lookup table. See Table 4.2. To describe the operation of the memory module, all the interface functions are listed, and their operation explained. The memory segment interface consists of the following functions:

	Full	Done
Filling		
Reading	✓	
Done	✓	✓

Table 4.2: The memory states of ”Multi memory segments”

`int AllocateSegment(int size)`
When a segment is allocated, the new data is saved in the segment table head by looking at the `Table Head` pointer. The start and end pointers are calculated based on the `Mem Head` pointer. If the range between `Mem Head` and `Mem Tail` is smaller than `int size`, -1 is returned, indicating error. If not, the segment table id is returned(`int seg_ID`).

`int FocusSegment()`

A segment is classified as a focus segment if the data is ready to be sent (`Reading` state). It will automatically find the next segment, by incrementing the pointer by one, and if needed wrapping back to 0. The `Table Tail` is also moved by this action, to always point on the first instance of an active segment.

`int SaveData(int seg_ID, int offset)`
Returns the memory address for that specific segment with an offset. This is calculated by finding the start address in the segment table, and adding the offset. If the segment is not in "Saving" state, an error is returned of -1.

`int LoadData(int seg_ID, int offset)`
The same as `int SaveData(...)`, with the exception that an error is returned if we are not in the loading state.

`int DelaySegment(int seg_ID)`
This function delays a focus segment by copying it to the table head, and freeing the current `int seg_ID`. When this happens, the chronological order of the segments gets mixed.
It is now impossible to tell if the next segment in the segment table contains the actual continuation of the memory. The next segment may be out of order, and point to memory anywhere. To get around this, each block gets an incremental ID at creation time. Since the creation of blocks always uses memory from the range `Mem Head` to `Mem Tail` the memory must be consumed in order.

`void SaveMetaData(int seg_ID, MetaData meta)`
Saves the `MetaData` into the current segment.

`MetaData LoadMetaData(int seg_ID)`
Loads the `MetaData` from the current segment.

`void SegmentFull(int seg_ID)`
Returns if the segment is `Full`.

`void SegmentDone(int seg_ID)`
Returns if the segment is `Full`(See Table 4.2).

`int AllocateSegment(int size)`
Allocates a segment by getting the first available from `Table Head`. If none are available, return

Single memory segments

The single memory segments are a bit different compared to the multi segments. The single segments does not return an `int seg_ID` at any point, because the focused segments are controlled internally. This is done to limit the complexity for of the model, and to insure that segments are not accessed out of order. Knowing and keeping the order makes it possible to allocate segments without knowing the size beforehand.
The internal segment control consists of three pointers. A pointer for the segment that is currently being saved to, a pointer for the segment that is currently being read from, and a pointer to the next segment ready for allocation.
This model is used as the internal buffers(see subsection 4.2.4). This requires that the metadata can be saved before saving the data.
Since segments in this implementation have undefined length before being marked as `Full`, only one segment can be filled at a time. To save metadata, the "allocation" method from "Multi memory segments" are reused. Instead of giving the size of the segment, the metadata is given. This marks the segment as `Active`. An `Active` segment is ready to receive data, but only when the currently receiving segment is done filling. See Table 4.3 for the different states.

Finish!

4.2.3 Dictionary

The dictionary are used to keep track of out of order memory segments.

Finish!

	Full	Done	Active
Inactive			
Filling			✓
Reading	✓		✓
Done	✓	✓	✓

Table 4.3: The memory states of ”Single memory segments”

4.2.4 Memory types

Block ram on FPGA chips from companies such as Xilinx have a low latency, high throughput but low capacity. [39] The capacity limitation is a problem. Worst case the system would have to hold multiple packets of the maximum size for that specific protocol. For example, an IPv4 packet may have an max size of 65,535 bytes. [23] If a lot of packets accumulate in the system, or the user rarely empties the `Data In` buffer, we may run out of memory.

An additional solution would be the to use external memory with high latency, low(or high) throughput and high capacity. The throughput would have to be equal or bigger than the stream from the system itself, to not be an bottle neck.

In both cases, the memory takes at least one clock to get results back from memory. To guarantee that the data is avaliable as soon as possible (see: section 4.3) one must use perfecting of the memory.

This is solvable by using small internal buffers that use the fast registers. The size of these small buffers would be based on the latency between the request and response from memory. This would make it easier to replace the underlying memory, by simply increasing the buffer size to that of the memory latency.

Talk about how metadata is needed before the data itself

4.3 Interface Signal protocols

With the introduction of buffers between each parsing processes, a clear pattern emerged. The layer-handling processes are responsible for numerous real-time tasks (parsing, sending, protocol-specific tasks, etc), while also limited by their fixed internal buffers. These processes are not always ready to receive input from preceding processes, while they at the same time must be able to write their output to following processes immediately.

The buffers are a stark opposite, as their large internal block memories enable them to buffer huge chunks of memory, while also being able to wait for the succeeding process to start reading.

With these two established scenarios, protocols for each can be proposed – the Buffer-Producer protocol, and the Compute-Producer protocol.

4.3.1 Buffer-Producer

The Buffer-Producer (BP) is the interface signal protocol where the producer of the data is a buffer process (such as `Data Out` or `Segment Out`).

The Buffer-Producer is heavily inspired by the Transfer signalling protocol in the AXI4-Stream standard, which ensures a two-way flow-control mechanism for both the producer and the consumer[2] [42].

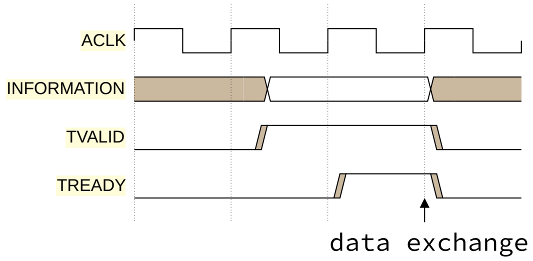


Figure 4.5: The AXI4 handshake process, adapted from “AMBA 4 AXI4 Stream-Protocol specification” by ARM, 2010, p. 19.

The AXI4 protocol uses two signals (also called ”flags”), the `TVALID` on master, and `TREADY` on slave. Every time both `TVALID` and `TREADY` are asserted during a clock-cycle, a data-exchange happens. Figure 4.5 shows a data exchange, where the information (the bytes) are placed on the bus

[2] The producer and consumer are called master and slave respectively in the AXI4 specification.

and the TVALID is raised. When this signal propagates to the slave, it asserts TREADY. When this signal propagates back to the master, it knows that the information was read, and that it can proceed with the next byte, or in this case, de-assert the TVALID to indicate no more bytes available [42].

The information transferred in the AXI4 protocol can be defined by the user, but it usually is a payload consisting of multiple element, such as a byte location or the type of the data.

The Buffer-Producer protocol draws heavy inspiration from this model, as it provides a simple flow-protocol with only a few flags. In the BP protocol, the producer has a `valid` flag, while the consumer has the `ready` flag. However, without any modifications, the AXI4 protocol sees the data exchanged as being a single data-stream. In the case of the Buffer-Producer however, we want to differentiate between the end of a packet (or simply just a segment of data), and the beginning of a new one. In the very first iterations, this was done by distinguishing the parsed data by finding a known delimiter, such as the ID of the IPv4 packet, or the sequence number of the TCP header. Unfortunately, this solution is very dependent on the scenario where different delimiters might appear, or not be present at all. To generalize this issue, an additional signal `bytes_left` is added to the control bus of the protocol, which indicates the end of a current data-block by setting `bytes_left = 0`. The final Buffer-Producer protocol can be summed up in these following rules:

- A data transfer only occurs if both `valid` and `ready` are raised at the same clock-cycle.
- When the producer has data available, it is immediately put in the bus and the `valid` flag is raised.
- Once the `valid` flag is raised, it cannot be reset until a data-transfer occurs.
- The consumer is allowed to wait until the `valid` flag is raised before raising the `ready` flag.
- If a consumer raises the `ready` flag, it is allowed to reset it before `valid` is raised.

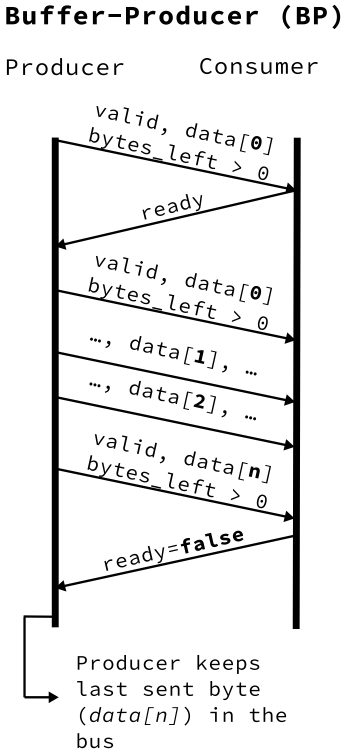


Figure 4.6: The usual data-transfer between a buffer (Producer) and a compute-process (Consumer).

The conventional data-exchange using the BP protocol in the network stack is perhaps better visualized by a sequence diagram on figure 4.6.

4.3.2 Compute-Producer

The Compute-Producer (CP) protocol is the interface signal protocol from a compute-process to a buffer. The requirement for this protocol is that compute-processes do not usually have the luxury of being able to wait with the data transfer, which usually happens if the compute-process is building a packet header or passing information along from another buffer.

The concept for the Compute-Producer model is fairly simple; since the producer (compute-process) does not have the luxury to wait, it always sends the data on the bus, regardless if the consumer is ready. It is up to the producer to mark the end of an ongoing data-stream.

Thus, the rules for the Compute-Producer protocol are as such:

- If the producer puts data on the bus, the `valid` flag must be raised.
- If `bytes_left` is greater than 0, the data in the next clock will be valid.
- If `bytes_left` is 0, the current byte ends the current sequence of bytes.
- If the consumer deasserts `ready`, it *may* not read the data in the bus.
- The producer may act upon the knowledge that the consumer is either reading (`ready = true`) or ignoring (`ready = false`) the data.

Such a scenario is visualized on figure 4.7, where the consumer becomes unavailable during the transaction. The producer has the opportunity to drop the transaction, but it might also continue till the end.

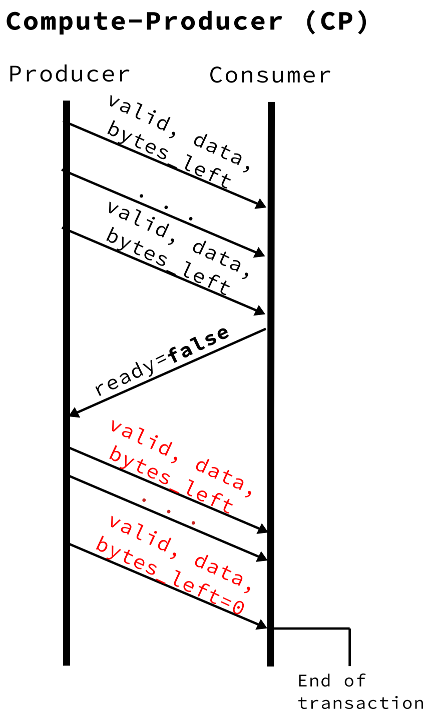


Figure 4.7: The usual data-transfer between a compute-process (Producer) and a buffer (Consumer). Note that the consumer becomes unavailable halfway through the transaction.

4.4 Interface Control

The `Interface` is a collection of 3 busses provided to the user. Two of the busses are direct connections to the data-buffer, used to transfer the actual data to and from the network. As seen on 3.3, the connecting going to `Data In` follows the Compute-Producer interface signal protocol, while the connection from `Data Out` follows the Buffer-Producer protocol.
The connection going from the interface into the `Transport` process is a bit more interesting. It is simply called the `InterfaceBus`, and it is used to control the whole networking stack. As per usual, the connection actually consists of a `InterfaceBus` controlled by the user, and the `InterfaceControlBus`, used by the networking stack itself to respond to the user requests.
Unlike the connection between buffers and processes where chunks of data are transmitted over multiple clock-cycles, the interface connection is more of a request-response model with only 1 clock-cycle required to submit the request or send the response. However, after submitting the request, the user *should* keep the data in the bus until a response is received, because the `Transport` process might be busy at the moment handling in- or out-going packets, and not have time to process the request.
Figure 4.8b shows the definitions of the busses. Note the `interface_function` byte containing the type of "function" to call, defined in the `InterfaceFunction` `enum` on figure 4.8a.

4.4.1 Usage

The usage of this interface is very basic and primitive. The user sets the appropriate values in the `InterfaceBus`, and raising the `valid` flag. For example, to start listening on port 81 using the UDP protocol, the user would put these values in the bus:

```
enum InterfaceFunction : byte
{
    INVALID = 0,
//    BIND = 1,
    LISTEN = 2,
    CONNECT = 3,
    ACCEPT = 4,
    CLOSE = 7,
    // ...
    OPEN = 255,
}

struct InterfaceData
{
    public int socket;
    public uint ip;
    public byte protocol;
    public ushort port;
}
```

(a) Definitions of the structures used in the interface busses

```
interface InterfaceBus : IBus
{
    bool valid;
    byte interface_function;
    InterfaceData request;
}

interface InterfaceControlBus : IBus
{
    bool valid;

    byte exit_status;
    byte interface_function;
    InterfaceData request;
    InterfaceData response;
}
```

(b) The definitions of the interface busses.

Figure 4.8: Pseudocode of the definitions used for the interface connection.

```
InterfaceBus {
  .interface_function = CONNECT
  .request {
    .ip = 10.0.0.2
    .protocol = UDP
    .port = 81
  }
  .valid = True
}
```

If the port is available under the UDP protocol and sockets are available for allocation in the stack, the user should eventually receive a response. Here, the user gets an `OK` response with the `socket = 2`:

```
InterfaceControlBus {
  .valid = True
  .exit_status = OK
  .interface_function = CONNECT
  .request {
    .ip = 10.0.0.2
    .protocol = UDP
    .port = 81
  }
  .response {
    .socket = 2
  }
}
```

Once the user has received a valid socket, they can use the it at heart's content for sending and receiving. For example, to send a byte:

```
DataOut.WriteBus {
  .socket = 2
  .byte = "A"
}
DataOut.ComputeProducerBus {
  .valid = True
  .bytes_left = 0
}
```

The user is not required to wait for any response, as the Compute-Producer interface signal protocol is used.

4.4.2 Limitations

As briefly mentioned, a big limitation of the interface control is that only one request can be proposed at a time, and the user has to wait an arbitrary number of clocks before the response arrives. The issue is that the `Transport` might be occupied processing in-going and out-going packets.

To circumvent this, experimental features of creating a queue of requests in the `Transport`-process was made. However, this only added complexity to the code, increased the resource-consumption of the process by a large margin just in order to maintain the queue data-structure, and apart from convenience, it added no improved performance. For these reasons, the initial approach was kept.

Chapter 5

Evaluation

Remember to describe that we could not get the network stack down on the FPGA

5.1 Setup

5.2 Test

5.3 Verification

Chapter 6

Discussion

The intent of this project was not only to implement an efficient, high-speed, and responsive TCP/IP stack, but also to test the viability of SME for this types of projects. In this chapter, the results from the tests are investigated, the viability of the networking stack is discussed, and the usage of SME for this project is examined.

6.1 Performance

In the test chapter 5, we ran the simulation for multiple hours. In about 10 million clock cycles, the networking stack was able to handle 118400 packets in total, where 1280 of the packets were valid connections intended for the user of the stack. During the test, the valid data-stream was also sent out in identical 1280 packets.
If the network stack was clocked at a very modest clock-rate of 10 MHz of the maximum 866 MHz on the Xilinx Zynq-7000 series [43], the network is theoretically able to run at:

$$10000000\,\mathbf{bytes} = 10000\,\mathbf{KB} = 10\,\mathbf{MB}$$

$$\frac{10\,\mathbf{MB}}{10\,\mathbf{MHz}} = 8\,\mathbf{Mbps}$$

These speeds are by no means exceptional, in fact, they are very slow compare to even common ethernet network adapters found in consumer server hardware, such as the Intel Ethernet I210 series [44].

However, the network stack never reached the actual hardware, so the theoretical speed is questionable. Furthermore, since the most resource-heavy processes cannot be investigated, optimization of the stack would be premature as of now.

6.1.1 Improving performance

Even though many optimizations can be done on the network stack codebase itself, there are other ways of achieving better performance from the project.

During the design-phase, it was important to ensure the active connections arrived on the same stack, since the IP fragments have to be collected on the same buffer, and the TCP state has to be synchronized.

The `Transport` is especially burdensome, as it has to handle both ingoing- and outgoing-packets, as well as the user interface calls and protocol-specific handshakes and other operations. Finding a way of getting around this congestion, it should be possible to parallelize the system for better performance.

By replicating the networking stack and assign a unique IP address to each, it should be possible to create a ”Load Distributor”, distributing the packets across multiple networking stacks. Figure 6.1 shows a prototype design, wherethe inner part of the networking stack is duplicated on the FPGA itself. The `Load Distributor` conveys the packets to the appropriate network-stack.

On the other side of the same figure, the `Load Collector` ensures that the replicated network stacks work as one, and that they are abstracted away in the `User Interface`, so that the user does not have to make up for the separation. Even though the `Transport` processes do not share any information across the stacks, they can have hardcoded `socket` numbers, assigning a block of sockets to each stack, so that overlap does not happen.

6.2 Usability

Perhaps the most important aspects of any software project is its usability, versatility, and its application. While it is shown in chapter 5 that the networking stack performs arguably well in a reasonable simulated scenario, it is to be seen whether the project can bring any value to the user.

6.2.1 Intended usage

The intended usage of the networking stack is to be integrated with existing FPGA hardware in order add networking capability to a system. These systems range from simple embedded Internet of things (IoT) devices to large NIC cards.
Although it is possible to connect an SME project with other VHDL projects, it is much more straight-forward to add the networking code directly into an SME project. For instance, other SME projects, such as the "High Throughput Image Processing in X-ray Imaging" developed by Troels Skjøttgaard Ynddal [45], which does realtime image processing on x-ray images, could benefit from a network stack by sending the processed images to a server for further analysis and backup.

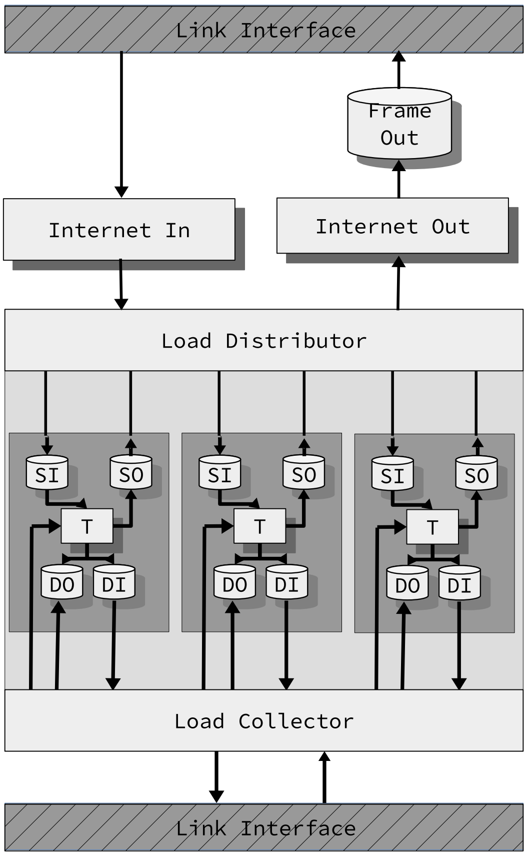


Figure 6.1: Replicating the "inner" part of the design in order to multiply the performance.

6.2.2 Existing solutions

Sadly, the developed networking stack could not be brought onto an FPGA, making the comparison to existing solutions difficult. In theory, if the networking stack worked on an FPGA, it would bring little to no runtime advantages over existing FPGA TCP/IP stacks, such as the Xilinx 10Gbps TCP/IP Stack [46]. However, the networking stack is easily extensible and modular. The design choices made during the development have proven to make the stack very flexible, and the programmer can easily add or remove protocols. The use of the C# programming language makes it more accessible for software engineers to modify the code without prior knowledge to the hardware itself, or special HLS tools and languages, albeit without the dynamic constructs of the C# language.

6.2.3 Integration with existing hardware

As an extension to this project, the code for the Digilent Pmod NIC100 [47] has been developed by Carl-Johannes Johnsen to act as the `Link Interface` [48], so that the networking stack could be tested on real hardware. While the stack never reached bare-metal, the testing suite simulates this connection from the Pmod100 into this `Link Interface`. Although testing on real hardware has to be carried out for definitive results, the simulation suggests that this connection with networking hardware can indeed work.

6.3 Using C# with SME

Not only had the use of C# with SME a great impact on the design, but the whole project as a whole.

6.3.1 Concurrency

As was the intent of the SME project, the messaging framework was indispensable during the development of the networking stack. Besides not having to concerns ourselves with the synchronization across processes, the "Shared Nothing" property of SME was a great help during the design, and forced the design to be neatly isolated in the appropriate layers.

The isolation of each `Process` object gave a great overview of what each "thread" of the system was doing. Likewise, the information exchange was very clear, since it was nicely collected in one place – the bus.

Sadly, the parsing of network packets is excessively sequential, and the project does not utilize the full potential of SME in that regard.

6.3.2 Cumbersome initialization and alternation

The way projects are written in SME with C#, the programmer has to choose a design hardware architecture beforehand, since adding new SME processes after the fact is cumbersome, and requires a lot of manual modifications. For instance, injecting a new process in between two existing processes requires potentially adding new busses to the system and set these up correctly in the code. Although this is a trivial task, the process itself is prone to typos and oversights from the programmer.

6.3.3 Using the C# language

In the beginning, it was pleasant to use a familiar language for the implementation of the networking stack. Yet, as the code-base grew in size, the flaws started to surface.

The C# programming language is a very actively developed language with many modern features, it is widely used with a very active community, and it has a lot of useful packages and frameworks.

Unfortunately, almost none of these features are available with SME, as most of these require dynamic constructs, such as instantiation of new classes or use of dynamic collections. Although this restriction is caused by the hardware and not SME itself, the C# language started to be much less intuitive. Instead of applying best practices and exploring new features in a well-known and established language, we had to limit ourselves to the viable subset of the working features, and create the design around it. While it is understandable why a proven language is used to test out a new message passing framework, the C# language did not always feel right for hardware development, and the upcoming SME Implementation Language (SMEIL) might prove itself to be a welcome addition to the SME project. [49].

Pre-written components and modules

The SME framework already contains premade objects for use, such as the `TrueDualPortMemory`, which is a process that helps the programmer write code interfacing with the built-in block memory on the FPGA.

As of yet, only a few memory components

are included in the SME framework.

During development, there was a pattern of prevailing components and systems reused in multiple processes:

- **Generic buffer process**
  As documented in the design section, the buffers make up a big part of functionality in the whole system. However, these First In, First Out (FIFO) constructs are very commonly used during FPGA development [50]. A generic FIFO buffer component would be a very useful feature during the development of the networking stack.

- **AXI4 communication process**
  In the processes, the programmer is free to implement any bus interface signal protocol. Although a custom protocol was implemented for the system, it has been cumbersome and challenging to implement two processes with a shared signal protocol. Pre-written processes with support for established signal protocols could be a valuable addition to the SME ecosystem.

6.3.4 State modelling

6.3.5 Bugs and other lesser issues

TheSME framework has been surprisingly stable, albeit with a few minor bugs. Although most of these issues are being fixed at the time of writing, they were still a noticeable hindrance during development:

- **Reserved VHDL keywords**
  Certain words in the C# code were reserved inVHDL. While this is a non-issue if the project is not being compiled to VHDL code, this needed to be taken into consideration during development. One such reserved word was the "Transport", used to denote the `Transport` process class.

- **C# structs**
  A lot of information has to be passed between the buffers and processes. For the sake of readability and maintainability of the codebase, this data is best encapsulated in namespaces for a hierarchical organization. Luckily, `structs` became available shortly after the discovery of this requirement, and are used in the project. For instance, the `InterfaceBus` uses two of the `InterfaceData` struct.

- **Slow simulation**

Chapter 7

Conclusion

Chapter 8

Future work

8.1 Improvements

Although the current implementation of the networking stack has enough functionality to be useful in certain applications, there are a lot of improvements that can be carried out in order to better the project.

8.1.1 Wider data-channels between processes and buffers

One of the main bottlenecks for the throughput is the raw amount of data the networking stack is working with at any given moment. Currently, the channels in the SME busses carrying data are only a byte wide, that is, at any given clock, a process will only have at most 8 bits to work with. This 1-byte width has been used because it is the most intuitive to work with, as packet-header fields usually span a factor of 8 bits.
Extending the width of the data-channel in the busses might improve the throughput manyfold, but it might also introduce more logic to the bit-shifting and masking to parse the headers. Additionally, this change does not only affect the parsing and the headers, but also the logic in the addressing for storage in the buffers, as well as, for instance, the checksum calculation.

8.1.2 Add external verification tool

The tests performed and described in chapter 5 on the current implementation have shown no major nor breaking bugs or errors for the supported subset of protocols in the internet protocol suite. Since the existence of undiscovered bugs cannot be ruled out with the custom-made testing suite, it is better to use an external verification tool in order to not transfer any wrong implementation-details over to the test-tools.
There are a multitude of ways to TCP/IP implementation with both regards to bandwidth and throughput, stability, loss, etc. One such tool is the `iperf3`, which is a ”TCP, UDP, and SCTP network bandwidth measurement tool” [51]. It is a highly efficient tool for both performance test, but also verify the correctness of a networking stack. By introducing iperf3 as a means to test the implementation of the networking stack, a much better insight of the correctess of the stack can be gained.

8.1.3 Unsupported Transport protocol pass-through to Application layer

Write this

8.2 New features

As discussed in chapter 3, the networking stack is designed with extensibility in mind, with a separation for the link-, the internet-, and the transport-layer. Given a protocol following the classic internet protocol suite 4-layer model, it should be possible to implement this protocol in the networking stack without major changes to the underlying structure of the whole codebase.
The implementation only supports UDP in the transport layer, which is still a great streaming protocol for a multitude of viable scenarios. However, situations where packet loss or out-of-order data-stream is

not acceptable, TCP is the ideal choice of protocol to implement in the stack. With the ability to share state across sockets in the `Transport` module, the parser for the TCP header can be implemented, and the out-of-order nature of the incoming data is already handled in the `Data In` buffer.
Likewise, the newer version of the Internet Protocol, the IPv6, is slowly, but surely, seeing adoption on the internet. Fortunately, the protocol itself is actually slightly easier to parse compared to its predecessor, the IPv4. For instance, multiple flags from the older IPv4 header have been moved into the optional section of the IPv6. Furthermore, fragmentation of IPv6 packets have been restricted somewhat, and not all hosts are required to support it [52].

8.3 Adding a firewall

In tandem with this project, a FPGA firewall has been developed by Patrick Dyhrberg Sørensen and Emil Sander Bak. Just like the networking stack, the firewall is implemented in SME, with the intent to be easily integrated with networking stacks, such as this one [53].
The firewall consists of whitelist rules, connection states, ”deciding” processes, and busses for communication and synchronization [53].
The processes making the actual decisions are split into 3 state processe:

- **Incoming IPv4**
  The first point where the firewall can do any decisions is when the internet protocol header is parsed. This header provides information such as the source and destination address, and the firewall can do basic checks for blacklisted or whitelisted addresses, as well as detecting some suspiciously formatted packets.

- **Incoming Transport**
  The transport-part of the firewall is perhaps a bit more interesting. Here, the firewall can not only check on the source and destination ports of a packet, but also verify that there are no malicious intents with regards to the protocol specification itself. For example, the firewall can detect and stop a SYN-flood attack, which is a form of denial-of-service attack (DoS attack) where the attacker sends a large number of TCP SYN-request to a host in order to consume all the host computational resources.

- **Outgoing**
  The last point that the firewall monitors is the outgoing traffic. This is in place for situations where the networking stack tries to communicate with malicious or forbidden hosts.

8.3.1 Proposal for the design of incorporating the firewall

Although both projects have been designed to be combined with each other, there have not yet been any attempts at uniting them. That being said, both projects can run self-sufficiently, and all the essential information can be easily shared across the processes with SME busses. The proposed design for incorporation can be seen on figure 8.1, where the parsing processes send information to the firewall, and the firewall in turn shares its decision with the consequent buffer. This way, the parsing process has to add all the parsed fields in the bus, and the firewall can mark the current packet in the bus if it needs to be removed.
The main tasks to implement firewall into the networking stack are:

- **The bus connection and protocol**
  The first task is to create the busses for the communication, and to agree on a signal protocol. Since only one data-transaction with the header info is needed for each packet, this should not pose a too big of a challenge.

- **Connection and packet identification**
  The networking stack relies on multiple methods of identifying packets. Once a packet reaches the transport layer, the networking stack can identify which connection (if any)

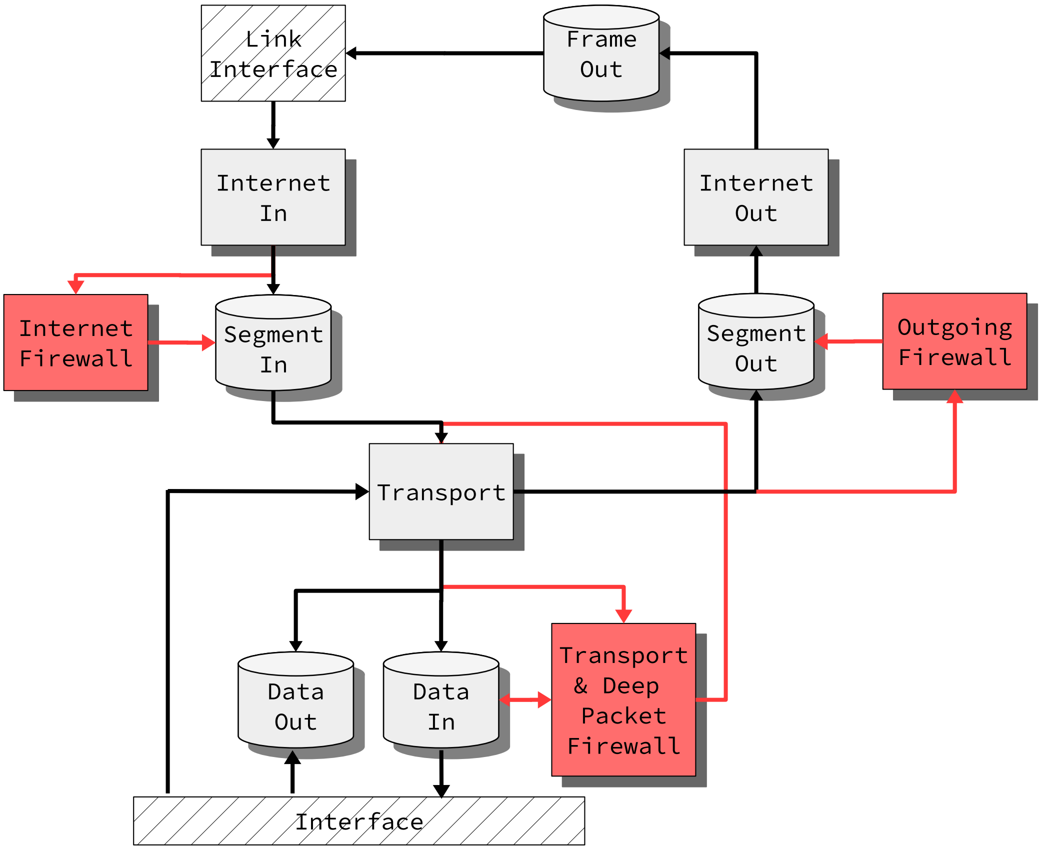


Figure 8.1: The proposed design for integrating the network stack with the accompanying firewall

the packet belongs to. To delimit network-frames, a frame-number is used, which is supplied by the interface. An IPv4 datagram is identified with its `Identification` header field, and transport connections are identified by their socket number. Depending on the implementation of the firewall, there needs to be made an agreement on the identifiers used to distinguish connections across the projects.

- **Buffer support for packet removal**
  Apart from the packet itself, the buffer holds additional meta-data used both by the next parsing layer, but also by the buffer itself to do defragmentation, segmentation, and so on. If a packet is detected to be malicious by the firewall, a flag should be added to the entry in the buffer to indicate the removal of it.

- **User interface support to control the firewall**
  It is reasonable to think that the network user would want to control the firewall live by adding new rules to the white-list, or block certain port numbers. To simplify this process, new function-calls can be added to the `Interface`, and this interface can be connected to the global state-table in the firewall.

Appendices

Bibliography

[1] J. C. Mogul, "Tcp offload is a dumb idea whose time has come," USENIX Association, 05 2003.

[2] X. Inc., "100g nic with pp integration." https://www.xilinx.com/applications/data-center/network-appliances/100g-nic-pp-integration.html. [Online; accessed 2019-05-13, Archived by WebCite ®at http://www.webcitation.org/78L0it9n0].

[3] Microtronix, "Ip core and fpga products." https://www.microtronix.com/products/IP-CORE-AND-FPGA-PRODUCTS-c32613313, 2019.

[4] AVNET, "Ip cores." https://www.avnet.com/shop/us/c/programmable-logic/ip-cores/, 2019.

[5] OpenCores, "Opencores: Mission." https://opencores.org/about/mission, 2019.

[6] X. Inc., "Signonce ip licensing." https://www.xilinx.com/alliance/signonce.html, 2019.

[7] D. Sidler, G. Alonso, M. Blott, K. Karras, *et al.*, "Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware," in *FCCM'15.*

[8] X. Inc., "Loop pipelining and loop unrolling." https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/sdsoc_doc/topics/calling-coding-guidelines/concept_pipelining_loop_unrolling.html, 2019.

[9] F. L. Herrmann, G. Perin, J. P. J. de Freitas, R. Bertagnolli, João, and B. dos Santos Martins, "An udp / ip network stack in fpga," 2009.

[10] D. Inc., "Digilent: Spartan-3e starter board (limited time)." https://store.digilentinc.com/spartan-3e-starter-board-limited-time/. [Online; accessed 2019-08-22, Archived by Internet Archive Wayback Machine®at https://web.archive.org/web/20190417082818/https://store.digilentinc.com/spartan-3e-starter-board-limited-time/].

[11] R. Braden, "Requirements for internet hosts - communication layers," STD 3, RFC Editor, October 1989. http://www.rfc-editor.org/rfc/rfc1122.txt.

[12] M. Waldrop, "Darpa and the internet revolution." https://www.darpa.mil/attachments/(2015)%20Global%20Nav%20-%20About%20Us%20-%20History%20-%20Resources%20-%2050th%20-%20Internet%20(Approved).pdf. [Online; accessed 2019-04-29, Archived by WebCite ®at http://www.webcitation.org/77zkNxkL8].

[13] T. S. S. O. ITU, "Open systems interconnection - basic reference model: The basic model," std, ITU, November 1994. [Online; accessed 2019-04-29, Archived by WebCite ®at http://www.webcitation.org/77zmViPac].

[14] D. Katz, “Proposed standard for the transmission of ip datagrams over fddi networks,” RFC 1103, RFC Editor, June 1989.

[15] E. D. Electronic Industries Associationm, *Interface between data terminal equiptment and data communication equipment employing serial binary data interchange*. Electronic Industries Association, 1969.

[16] N. M. of American History, “Ethernet prototype circuit board.” https://americanhistory.si.edu/collections/search/object/nmah_687626, 2019.

[17] W. Stevens, *TCP/IP illustrated, The Protocols*. Reading, Mass: Addison-Wesley Pub. Co, 1994.

[18] D. Heywood, *Drew Heywood’s Windows 2000 Network services*. Indianapolis, IN: SAMS, 2001.

[19] I. S. Association, “Ieee 802.3 ’standard for ethernet’ marks 30 years of innovation and global market growth.” https://standards.ieee.org/news/2013/802.3_30anniv.html. [Online; accessed 2019-08-8, Archived by Internet Archive Wayback Machine®at https://web.archive.org/web/20140112041706/http://standards.ieee.org/news/2013/802.3_30anniv.html].

[20] C. Hornig, “A standard for the transmission of ip datagrams over ethernet networks,” STD 41, RFC Editor, April 1984.

[21] J. Postel and J. Reynolds, “Standard for the transmission of ip datagrams over ieee 802 networks,” STD 43, RFC Editor, February 1988. http://www.rfc-editor.org/rfc/rfc1042.txt.

[22] G. Inc., “Google ipv6 statistics, ipv6 adoption.” https://www.google.com/intl/en/ipv6/statistics.html, 2019. [Online; accessed 2019-08-08, Archived by Internet Archive Wayback Machine®at https://web.archive.org/web/20190731110637/https://www.google.com/intl/en/ipv6/statistics.html].

[23] J. Postel, “Internet protocol,” STD 5, RFC Editor, September 1981. http://www.rfc-editor.org/rfc/rfc791.txt.

[24] T. Berners-Lee, R. T. Fielding, and H. F. Nielsen, “Hypertext transfer protocol – http/1.0,” RFC 1945, RFC Editor, May 1996. http://www.rfc-editor.org/rfc/rfc1945.txt.

[25] A. Bhushan, “File transfer protocol,” RFC 114, RFC Editor, April 1971.

[26] J. Postel, “Simple mail transfer protocol,” RFC 788, RFC Editor, November 1981.

[27] X. Inc., “Powering next-generation automotive solutions.” https://www.xilinx.com/applications/automotive.html, 2019.

[28] X. Inc., “Powering next-generation automotive solutions.” https://www.xilinx.com/applications/audio.html, 2019.

[29] Bing, “Bing launches more intelligent search features.” https://blogs.bing.com/search/march-2018/Bing-Launches-More-Intelligent-Search-Features, 2018.

[30] N. Instruments, “Fpga fundamentals.” http://www.ni.com/da-dk/innovations/white-papers/08/fpga-fundamentals.html, March 2019. [Online; accessed 2019-05-12, Archived by WebCite ®at http://www.webcitation.org/78K4kDEjB].

[31] X. Inc., "What is an fpga?." https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html, 2019. [Online; accessed 2019-05-12, Archived by WebCite ®at http://www.webcitation.org/78K4ocp7U].

[32] R. W. Andrew Moore, *FPGA for dummies.* 111 River St., Hoboken, NJ 07030-5774: John Wiley & Sons, Inc., 2017.

[33] X. Inc., "Cpld." https://www.xilinx.com/products/silicon-devices/cpld/cpld.html, 2019. [Online; accessed 2019-08-19, Archived by Internet Archive Wayback Machine®at https://web.archive.org/web/20190123013148/https://www.xilinx.com/products/silicon-devices/cpld/cpld.html].

[34] T. Inc., "Product - fpga." http://www.tul.com.tw/ProductsPYNQ-Z2.html, 2019. [Online; accessed 2019-08-19, Archived by Internet Archive Wayback Machine®at https://web.archive.org/web/20190328015350/http://www.tul.com.tw/ProductsPYNQ-Z2.html].

[35] B. Vinter and K. Skovhede, "Synchronous message exchange for hardware designs," 08 2014.

[36] J. Markus Bjørndalen, B. Vinter, and O. J. Anshus, "Pycsp - communicating sequential processes for python.," vol. 65, pp. 229–248, 01 2007.

[37] B. Vinter and K. Skovhede, "Bus centric synchronous message exchange for hardware designs," 08 2015.

[38] T. Asheim, "A domain specific language for synchronous message exchange networks," 2018.

[39] X. Inc., "7 series fpgasmemory resources – user guide," data sheet, 2019.

[40] M. T. Inc, "Enc424j600/624j600 data sheet, stand-alone 10/100 ethernet controller with spi or parallel interface," data sheet, Microchip Technology Inc., 2009.

[41] G. WRIGHT, *TCP/IP ILLUSTRATED, VOLUME 2 (PAPERBACK) : the implementation.* Place of publication not identified: ADDISON-WESLEY, 2017.

[42] A. Limited, "Amba 4 axi4-stream protocol, version 1.0, specification," data sheet, 2010.

[43] X. Inc., "Zynq-7000 soc." https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html#productTable, 2019.

[44] I. Inc., "1 gbe intel®ethernet network adapters." https://www.intel.com/content/www/us/en/products/network-io/ethernet/gigabit-adapters.html, 2019.

[45] T. S. Ynddal, "High throughput image processing in x-ray imaging," 5 2019.

[46] D. Sidler, Z. Istvan, and G. Alonso, "Low-Latency TCP/IP Stack for Data Center Applications," in *FPL'16.*

[47] D. Inc., "Digilent: Pmod nic100: Network interface controller." https://store.digilentinc.com/pmod-nic100-network-interface-controller/, 2019. [Online; accessed 2019-08-22, Archived by Internet Archive Wayback Machine®at https://web.archive.org/web/20190418052504/https://store.digilentinc.com/pmod-nic100-network-interface-controller/].

[48] C.-J. Johnsen, "Pmod-nic100." https://github.com/carljohnsen/Pmod-NIC100, 4 2019.

[49] K. Skovhede, “Sme implementation language work.” https://github.com/kenkendk/SMEIL, 2019.

[50] www.nandland.com, “What is a fifo in an fpga.” https://www.nandland.com/articles/what-is-a-fifo-fpga.html, 2019. [Online; accessed 2015-12-07, Archived by Internet Archive Wayback Machine®at https://web.archive.org/web/20151207092605/https://www.nandland.com/articles/what-is-a-fifo-fpga.html].

[51] ESnet, “iperf3.” https://software.es.net/iperf/, 2019. [Online; accessed 2018-04-22, Archived by Internet Archive Wayback Machine®at https://web.archive.org/web/20190422055959/https://software.es.net/iperf/. Source-code available at https://github.com/esnet/iperf].

[52] T. H. of Domain Names, “Ipv6 proposed.” http://www.historyofdomainnames.com/ipv6/, 2019. [Online; accessed 2018-06-12, Archived by Internet Archive Wayback Machine®at https://web.archive.org/web/20180612211153/http://www.historyofdomainnames.com/ipv6/].

[53] P. D. Sørensen and E. S. Bak, “Open source fpga firewall,” 09 2019. Unpublished.