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

August 2, 2019

Contents

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 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 Controllers (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 and the Transmission Control Protocol [17].
Modern hardware manufacturers can produce NICs boasting network throughput speeds as high as 100 Gigabits [8]. 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. Unfortunately, these blocks are usually distributed as black-boxes of VHDL code, which is hard to maintain, and even harder to debug and extend.

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 Synchronous Message Exchange model makes the network stack a very versatile implementation with ease of use, debugging, and even extension.
MORE TO COME!!

Chapter 2

Background

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 [6]. 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 [21]. 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 [14], 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]. 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.

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
- Carrying data for various upper layer 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

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

that it hardly ever makes sense to implement in the application layer.
While there are numerous protocols defined in the Transport Layer, perhaps the most well-known protocol in the stack is the Transmission Control Protocol (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) [3], File Transfer Protocol (FTP) [4], and Simple Mail Transfer Protocol (SMTP) [18].
This layer is usually implemented by the userspace 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), Socket on a Chip (SoC), and Field-Programmable Gate Array (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.
In this thesis, only FPGAs will be taken into consideration for its re-programmability, its fairly low-cost, and the compatibility with SME code-generators.

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 [10], audio decoders [11], or even internet search engines [5] 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 [13], 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 [12].

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 [13]. 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 [1].

[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.

2.3 Synchronous Message Exchange

The Synchronous Message Exchange model (SME) is a messaging framework created in order to help model hardware descriptions [19]. 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 [16]. 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 [19].

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 2.1. 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.

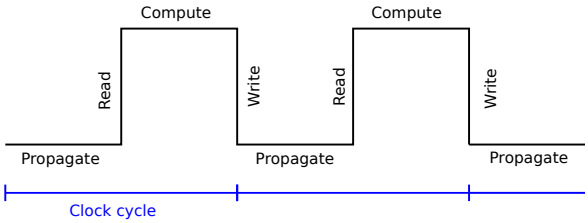


Figure 2.1: An illustration of a typical SME clock-cycle

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 [20].
SME has concurrent implementations in the C# and Python languages, with promising efforts to unify these under a common intermediate domain-specific language SMEIL [2]. 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 data-bus 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

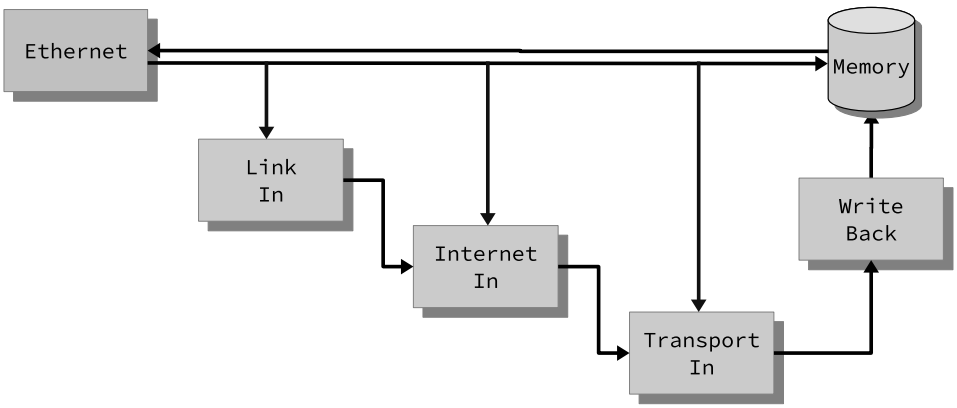


Figure 3.1: The initial design

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 [9] 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.

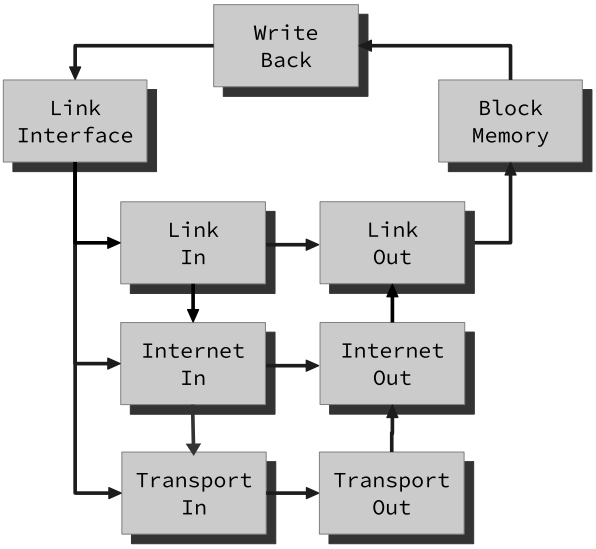


Figure 3.2: Second iteration of the design

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 [7]. 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.

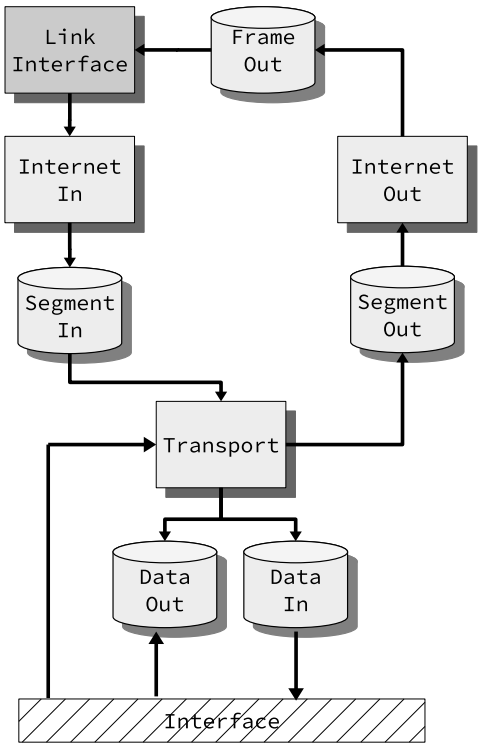


Figure 3.3: The final design. The rectangles represent "compute" processes, while the cylinders represent the buffers.

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.

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 4.4.

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

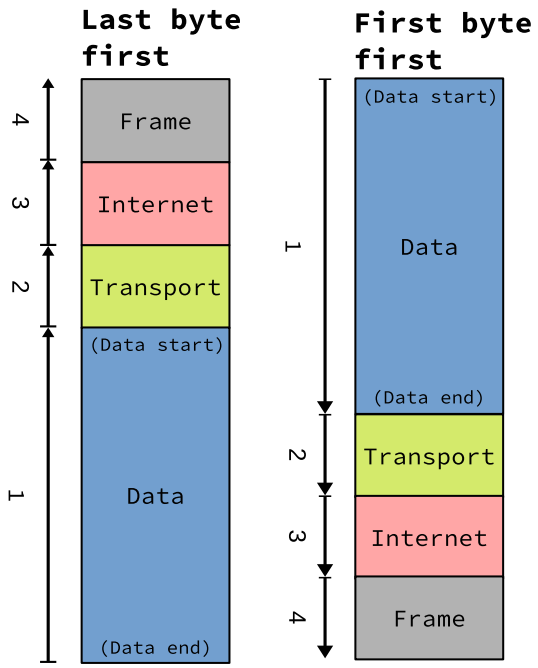


Figure 3.4: Possible orders of ways of building an outgoing packet

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.

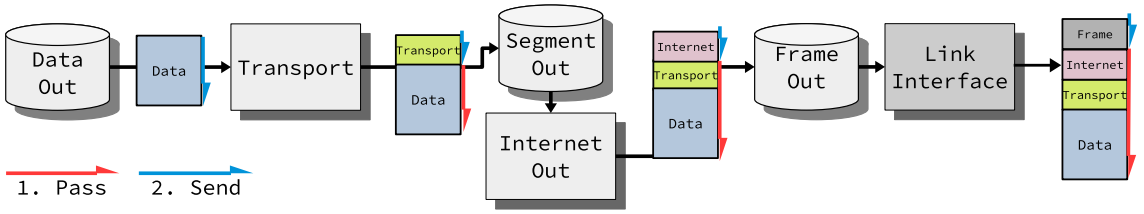


Figure 3.5: The order in which an outgoing packet is built and passed through the network pipeline. The colored boxes represent the state of the outgoing packet, the red arrow indicates the first stage of forwarding, and the blue line indicates the last stage.

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 [22]. 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 exchangning 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 [6]. 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].

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

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

`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

4.2.1 Internal memory

4.2.2 IPv4 de-fragmentation and segment unification

4.2.3 Allocation

4.3 Busses

4.4 Interface Signal protocols

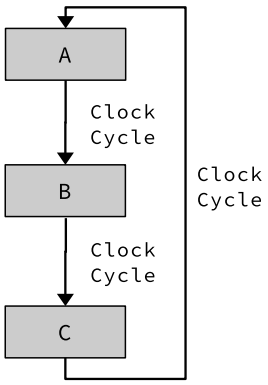
With the introduction of buffers between each parsing processes, a clear pattern emerged. The layer-handling processes are responsible for numerous real-time tasks (parsing, sending, protocol-specific tasks, etc), while also limited by their fixed internal buffers. These processes are not always ready 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.

```
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`

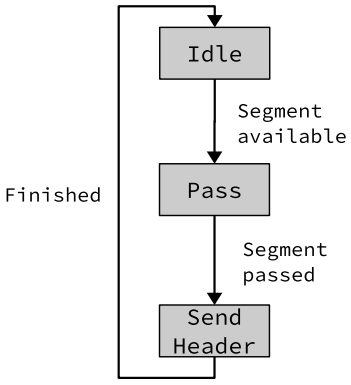


(c) The statemachine resulting from both code-examples

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

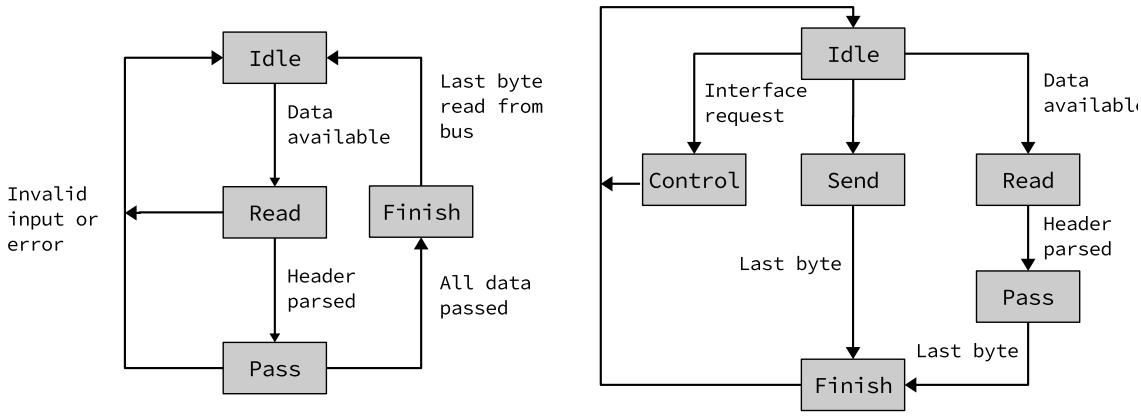
```
public partial class InternetOut:
    StateProcess
{
public override async Task
    OnTickAsync()
{
  while segment_available() {
    pass_segment();
    await ClockAsync();
  }

  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`

4.4.1 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] [15].

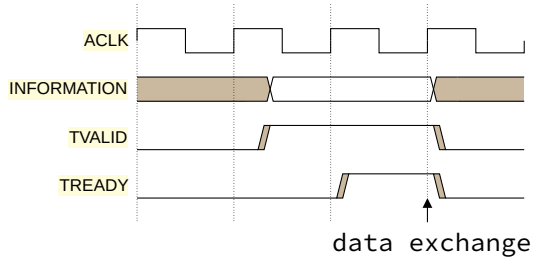


Figure 4.4: The AXI4 handshake process, adapted from "AMBA 4 AXI4 Stream-Protocol specification" by ARM, 2010, p. 19.

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 4.4 shows a data exchange, where the information (the bytes) are placed on the bus 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 [15].

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-

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

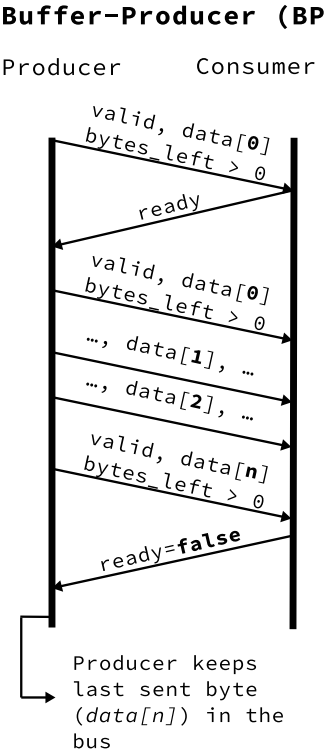


Figure 4.5: The usual data-transfer between a buffer (Producer) and a compute-process (Consumer).

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.

The conventional data-exchange using the BP protocol in the network stack is perhaps better visualized by a sequence diagram on figure 4.5.

4.4.2 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 4.6, 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.

4.5 Interface Control

4.5.1 Usage

4.5.2 Limitations

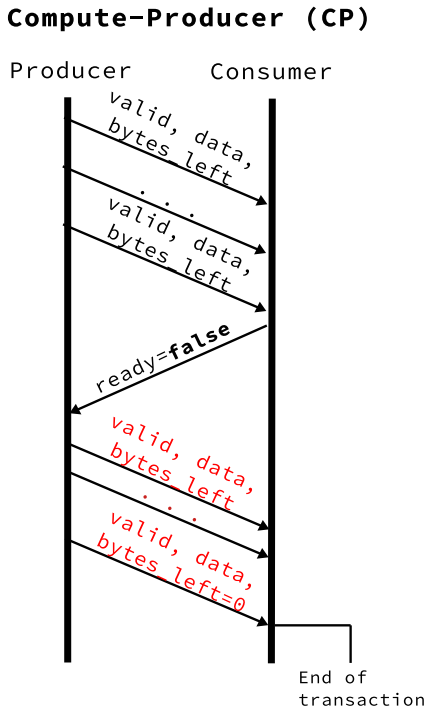


Figure 4.6: The usual data-transfer between a compute-process (Producer) and a buffer (Consumer). Note that the consumer becomes unavailable halfway through the transaction.

Chapter 5

Simulation

Chapter 6

Verification and evaluation

Chapter 7

Discussion

Chapter 8

Conclusion

Chapter 9

Future work

Appendices

Bibliography

[1] Ron Wilson Andrew Moore. *FPGA for dummies*. John Wiley & Sons, Inc., 111 River St., Hoboken, NJ 07030-5774, 2017.

[2] Truls Asheim. A domain specific language for synchronous message exchange networks. 2018.

[3] Tim Berners-Lee, Roy T. Fielding, and Henrik Frystyk Nielsen. Hypertext transfer protocol – http/1.0. RFC 1945, RFC Editor, May 1996. `http://www.rfc-editor.org/rfc/rfc1945.txt`.

[4] A.K. Bhushan. File transfer protocol. RFC 114, RFC Editor, April 1971.

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

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

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

[8] Xilinx 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`].

[9] Xilinx Inc. 7 series fpgasmemory resources – user guide. Data sheet, 2019.

[10] Xilinx Inc. Powering next-generation automotive solutions. `https://www.xilinx.com/applications/automotive.html`, 2019.

[11] Xilinx Inc. Powering next-generation automotive solutions. `https://www.xilinx.com/applications/audio.html`, 2019.

[12] Xilinx 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`].

[13] National 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`].

[14] TELECOMMUNICATION STANDARDIZATION SECTOR OF 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`].

[15] Arm Limited. Amba 4 axi4-stream protocol, version 1.0, specification. Data sheet, 2010.

[16] John Markus Bjørndalen, Brian Vinter, and Otto J. Anshus. Pycsp - communicating sequential processes for python. volume 65, pages 229–248, 01 2007.

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

[18] J. Postel. Simple mail transfer protocol. RFC 788, RFC Editor, November 1981.

[19] Brian Vinter and Kenneth Skovhede. Synchronous message exchange for hardware designs. 08 2014.

[20] Brian Vinter and Kenneth Skovhede. Bus centric synchronous message exchange for hardware designs. 08 2015.

[21] Mitch 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`].

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