Designing a scheduler for cloud-based FPGAs

Simon Jonsson
Master of Science Thesis in Electrical Engineering

**Designing a scheduler for cloud-based FPGAs**

Simon Jonsson
LiTH-ISY-EX--18/5162--SE

**Supervisor:**
Anders Hallberg
Ericsson AB,
**Michael Lundkvist**
Ericsson AB

**Examiner:**
**Kent Palmkvist**
ISY, Linköpings universitet

---

**ISY**
Department of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden

Copyright © 2018 Simon Jonsson
Acknowledgements

Big thanks to Ericsson in Linköping for sponsoring this thesis work with everything that has been necessary. A special thanks to my supervisors Anders Hallberg and Michael Lundkvist for all the support. Also thanks to Kent Palmkvist for being my examiner.
Abstract

English abstract

The primary focus of this thesis has been to design a network packet scheduler for the 5G (fifth generation) network at Ericsson in Linköping, Sweden. Network packet scheduler manages in which sequences the packages in a network will be transmitted, and will put them in a queue accordingly. Depending on the requirement for the system different packet schedulers will work in different ways. The scheduler that is designed in this thesis has a timing wheel as its core. The packages will be placed in the timing wheel depending on its final transmission time and will be outputted accordingly. The algorithm will be implemented on an FPGA (Field Programmable gate arrays). The FPGA itself is located in a cloud environment. The platform in which the FPGA is located on is called "Amazon EC2 F1", this platform can be rented with a Linux instance which comes with everything that is necessary to develop a synthesized file for the FPGA. Part of the thesis will discuss the design of the algorithm and how it was customized for a hardware implementation and part of the thesis will describe using the instance environment for development.
Svensk abstract

## Contents

Acknowledgements iii

Abstract v

List of Figures ix

List of abbreviations xi

1 Introduction 1
   1.1 Motivation ................................. 1
   1.2 Problem statements .......................... 2
   1.3 Research limitations .......................... 2
   1.4 Thesis outline .............................. 2

2 Background 3
   2.1 5G transmission ................................ 3
   2.2 Packet scheduler .............................. 3
       2.2.1 Structure .................................. 4
       2.2.2 Issues with this design ...................... 4
   2.3 Amazon EC2 F1 ................................ 5
       2.3.1 Hardware specification ....................... 5
       2.3.2 FPGAs in the cloud ......................... 5
       2.3.3 Marketplace ................................ 6
       2.3.4 Amazon machine image ...................... 6
       2.3.5 Amazon FPGA image .......................... 6
       2.3.6 F1 instance ................................ 7

3 Theory 9
   3.1 Carousel ....................................... 9
       3.1.1 Basic concept ............................. 9
       3.1.2 Timestamper ............................... 10
       3.1.3 Timing wheel .............................. 10
       3.1.4 Back pressure ............................ 12
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Basic traffic scheduler</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>AWS F1 instance</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>Structure of the AWS F1 platform</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>Carousel scheduler</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>Time now = 1</td>
<td>11</td>
</tr>
<tr>
<td>3.3</td>
<td>Time now = 2</td>
<td>11</td>
</tr>
<tr>
<td>3.4</td>
<td>Time now = 7</td>
<td>12</td>
</tr>
<tr>
<td>3.5</td>
<td>Top level of DDR4</td>
<td>13</td>
</tr>
<tr>
<td>3.6</td>
<td>Memory model</td>
<td>14</td>
</tr>
<tr>
<td>4.1</td>
<td>AXI4 read</td>
<td>19</td>
</tr>
<tr>
<td>4.2</td>
<td>AXI4 write</td>
<td>19</td>
</tr>
<tr>
<td>4.3</td>
<td>UltraScale Architecture-Based FPGAs Memory Interface Solution Core Architecture</td>
<td>22</td>
</tr>
<tr>
<td>4.4</td>
<td>DDR4 Memory</td>
<td>23</td>
</tr>
<tr>
<td>4.5</td>
<td>Overview of the system</td>
<td>24</td>
</tr>
<tr>
<td>4.6</td>
<td>Shell to custom logic communication</td>
<td>25</td>
</tr>
<tr>
<td>4.7</td>
<td>FSM of the address tracker</td>
<td>26</td>
</tr>
<tr>
<td>4.8</td>
<td>Structure of the testbench system</td>
<td>30</td>
</tr>
<tr>
<td>5.1</td>
<td>Speed drop over time</td>
<td>36</td>
</tr>
</tbody>
</table>
**List of abbreviations**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOC</td>
<td>System On Chip</td>
</tr>
<tr>
<td>5G</td>
<td>5 Generation (network)</td>
</tr>
<tr>
<td>4G</td>
<td>4 Generation (network)</td>
</tr>
<tr>
<td>RTL</td>
<td>Register-Transfer Level</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>NIC</td>
<td>Network Interface Card</td>
</tr>
<tr>
<td>FIFO</td>
<td>First in first out</td>
</tr>
<tr>
<td>LTS</td>
<td>Last Timestamp</td>
</tr>
<tr>
<td>NTS</td>
<td>New TimeStamp</td>
</tr>
<tr>
<td>SOP</td>
<td>Size Of Packet</td>
</tr>
<tr>
<td>CR</td>
<td>Configuration Rate</td>
</tr>
<tr>
<td>vCPU</td>
<td>Virtual Central Processing Unit</td>
</tr>
<tr>
<td>SSD</td>
<td>Solid State Drive</td>
</tr>
<tr>
<td>OpenCL</td>
<td>Open Computing Language</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DDR</td>
<td>Double Data Rate</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This thesis will discuss the subject of a hardware implemented scheduler. The reason for using hardware acceleration on an algorithm is to make it run more efficiently than by only using the software. The algorithm will be described using RTL code, which is a design method that can represent synchronous digital circuits by setting registers. These registers will then decide the flow of data between logical operations and registers[24]. The hardware and the instance with the software, licenses and everything a hardware developer will need to program are rented, and the hardware itself is placed inside a cloud. The focus of the thesis will be designing for the cloud-based hardware "Amazon EC2 F1" and investigate if this hardware algorithm with the FPGA is fast enough to schedule 5G data.

1.1 Motivation

One important concept in today’s society is mobile networks. Each day more and more devices are being connected to servers all over the world. More devices imply even higher data usage than ever before. For the network to be able to handle this massive workload Ericsson are developing products to support this task at hand. With the new fifth generations of the mobile system being develop new solutions to problems has to be found [5].

Cloud development is also something which has great potential, instead of multiple users that have access to a personal platform, all these users can instead rent the FPGA when it is needed. This keeps the cost down for a high purchase price. It is also more environmentally friendly since less FPGAs has to be manufactured, almost like carpooling.
1.2 Problem statements

The questions that this thesis aims to answer is developed from the viewpoint of 5G data scheduling, and also how a design can be limited by using an FPGA board design that cannot be modified by the user.

1. Is it possible to design the Carousel algorithm on the FPGA platform "Amazon EC2 F1" by using a streamlined kernel structure?

2. What are the limitations of the design and the FPGA?

1.3 Research limitations

This thesis will only consider the result of this specific FPGA design, and it cannot guarantee that the result will be the same on every FPGA design. The FPGA used in this study will be the "Amazon EC2 F1" with mostly opensource software. This thesis will not focus on optimizing the energy consumption, but only the implementation. The result will also be measured inside a testbench from a simulation which will have the same pin placement and the same external modules as the FPGA.

1.4 Thesis outline

The first part of the thesis will describe the theory behind hardware acceleration, and why it is sometimes suitable to use specific hardware to accelerate software algorithms. The theory chapter will also bring up the subject of schedulers. When the theory is established it will go through an implementation process of how the algorithm was designed for the FPGA, and also any changes that were made to the algorithm, also how the Amazon EC2 F1s specifications and some rules that must be followed when designing for the board. Finally the result and discussion of the system and what could have been done differently.
2 Background

2.1 5G Transmission

5G is the latest generation of mobile network, compared to the current generation, 4G which can transmit data wireless around a few hundred Mbps it is expected that 5G can send a few Gbps which is a massive improvement, higher speed also comes with higher bandwidth. While the IoT is growing so will the number of connected devices to the 5G network. When more devices get connected to the network more and more devices will naturally communicate with each other, this means a significant increase in traffic. Therefore it is important to schedule the system fast, efficiently and correctly to avoid packet drop and increasing traffic rate with traffic bursts.[8] [20]

2.2 Packet Scheduler

Packet scheduling is a technique used to achieve certain specification of data sent in a network. Traffic scheduling considers multiples factors to determine how important a certain packet is, and then arrange the packets in an order depending on different factors. Traffic scheduling can regulate factors such as the rate, bandwidth, throughput, changing the wait time and latency. Different scheduler aims at different goals; some scheduler main goal is to minimize the response time, while others can work for maximum throughput or both. It all depends on the need for their specific requirement. A lot of schedulers have certain design constraints, which makes it possible to optimize the design for specific needs, an example of this is what happens when there is a risk for overshoot, should the packet be dropped, or should the packet be bunched together to increase the rate. All of this depends on what kind of requirement the designer face if there is a hard limit on the rate then the packet should be dropped. While other times it
might be possible to implement a larger memory, for making it possible to buffer low priority packets for a later time.\[9\] \[12\] \[7\]

### 2.2.1 Structure

Traditionally and in a lot of cases the structure of a packet scheduler will have the components shown in figure 2.1

1. Classifier, which will configure the rate of the packets and divide them into different classes. The classifier will have multiple input sources will consist of data traffic.

2. Class queue, the queues are used for placing all the traffic with the same rate classification in the same queue.

3. Scheduler, the scheduler is used for scheduling the packets in the queues and send them for example, to a network interface card (NIC) or directly to a network.

The figure below describes the concept of a traffic scheduler. How the and scheduler works individually can differ a lot depending on the structure, however; the function of them will most likely be the same. [10]

![Figure 2.1: Basic traffic scheduler](image)

### 2.2.2 Issues with this design

This structure works well when it comes to smaller networks when there are not too many different traffic sources and not a lot of different rates which needs to be classified.

For every new traffic source, the classifier would need to be able to take care of one more input, and sometimes creating an additional rate. This will lead to that the rate queues would need to be stored in a data set that would be very large, which can lead to storage problems.
2.3 Amazon EC2 F1

The Amazon EC1 F1 is the cloud-based hardware that will be used in this thesis. The product is designed for software and hardware developer to accelerate and create hardware solutions for their software algorithms.[21]

2.3.1 Hardware specification

There are two different sizes of the F1 available. The f1.2xlarge and f1.16xlarge. In this thesis, the system will be designed for the f1.2xlarge, which consists of

1. One FPGA card
2. Eight vCPUs
3. 122 GiB of instance memory
4. 470 GB SSD storage

The FPGA is the "16nm Xilinx UltraScale Plus FPGA". The FPGAs connects to 64GiB DDR4 externally connected memory, and the CPU are connected with a 16xPCIe connection. Almost 7000 Digital Signal Processing (DSP) engines and around 2.5 million logic elements.

2.3.2 FPGAs in the cloud

FPGAs can be used instead of building a fully customized ASIC; it is possible to create custom hardware directly into the registers on the FPGA. For example, FPGAs were put into Bing search system, with a small percentage power increase, it was almost a doubled boost in conduct[11]. FPGAs are generally a bit less power efficient than an ASIC[15], but since power consumption is not a problem in this project, this will not be an issue. Cloud-based hardware is nothing new, servers have been used to store information in the cloud for a very long time. But the possibilities for not just storing data in the cloud is immense. One reason for using cloud computing is that there is no need to buy expensive hardware when it is as smooth and also cheap to rent one remotely. Cloud computing can be recognized as a compilation of recourses that can be split between users. FPGAs in the cloud can also be useful since it connects multiple people to the same platform which will have the same design rules. This can lead to a community with designers who all follow the same design rules.
2.3.3 Marketplace

The marketplace makes the construction of IP block convenient. This is something that AWS have had in mind when designing the platform. There is also a community for this already, on the AWS marketplace, where both hardware designers and software designers can put up their design in an encrypted environment, where the purchaser only have access to the function of the block, and not the actual code itself.

2.3.4 Amazon machine image

The Amazon machine image or AMI is the server where the instance is launched. This is where it is possible to SSH into. It is possible to turn off and on this instance when needed so that you only rent the instance when it will be used. To the AMI it is also possible to rent extra space so that the application can run and handle information. In this project, the FPGA developer AMI will be used.

2.3.5 Amazon FPGA image

The Amazon FPGA image (AFI) is what the completed design is defined as. This is what is deployed on the F1 instance or even the marketplace. The AFI can be reused as many time as needed.
2.3.6 F1 instance

Figure 2.3 shows how the user sees the AWS F1 on its platform. The operations will be placed in the application box inside the CPU. The application has access to make calls to the FPGA through accessing the API. The drivers of the x86 will then communicate with the DDR4 memory through the DMA. DDR memories are often used with FPGAs since they are both fast and big. When the Kernel in the FPGA is done with the calculations, it will then save the data in the memory and send it back to the CPU or another system connected to the PCIe.
The theory chapter will go over what the Amazon EC2 is and what it will be used for in this thesis. How hardware acceleration can be done for a software algorithm. The carousel algorithm, which will be redesigned and implemented in this study will also be described. It will also introduce the subject of how traffic scheduling works in its core structure.

3.1 Carousel

Carousel is a rate limiter which scales a significant number of sources. It relies on a single queue shaper which is based on a timing wheel. [6]

3.1.1 Basic concept

Carousel is an algorithm which the main idea is to have an individual queue for each output. If we take networking for example, if the task at hand is to schedule mobile communication and radio communication there should be two individual shapers and one shaper maximum on each CPU core. It is possible for the carousel to work with many CPUs, but there should always be maximum once shaper per CPU core. The scheduler consists of three different parts.

1. Timestamper
2. Timing wheel
3. Back pressure

The timestamp is used to determine the rate of the packets so that the packages can be scheduled at the correct location. When the rate has been specified it can be placed in the timing wheel; the timing wheel is a logic right-shifting matrix
with a definite size. To make sure that there is no overflow in the array, the back pressure signal will send a handshake to the data source to tell the system that the system can receive another input.

![Carousel scheduler](image)

*Figure 3.1: Carousel scheduler*

Assume that each aggregate on the timing wheel above is 1s.

### 3.1.2 Timestamper

The timestamper consist of multiple steps to calculate the actual timestamp. In this case, the first timestamp will be calculated from the UDP stack. This stamp will then go through more timestampers to aggregate the actual timestamp that will be used for insertion in the timing wheel. The timestamper is the first module in the Carousel. The function of the timestamper is to calculate the time of when the packet should reach the end of the timing wheel. To calculate this, the timestamper uses three different parameters. The size of the packet (SOP), configuration rate (CR) and the last time stamp (LTS) to calculate the new time stamp (NTS). The calculation is as described below.

\[
NTS = LTS + \frac{SOP}{CR}
\]  

(3.1)

### 3.1.3 Timing wheel

The Timing wheel is based on the calendar queue but uses time as an index. It can be seen as a circular array with buckets in every slot. The Calendar queue has the packet sorted within each queue slot, while the timing wheel has FIFOs. The timing wheel is the central part of the carousel, the size of the timing wheel can be described as
3.1 Carousel

\[ \text{SlotsInMatrix} = \frac{\text{Horizion}}{\text{G}_{\text{min}}} \]  \hspace{1cm} (3.2)

The time horizon, which is the maximum time for which a packet can be scheduled to can be calculated as:

\[ \text{Horizon} = \frac{l_{\text{max}}}{r_{\text{min}}} \]  \hspace{1cm} (3.3)

The height of the timing wheel is decided by the Lmax variable, which tells the system how many packets that it can fit onto each aggregate inside the wheel. The timing wheel works as a time-scaled calendar queue. Every single time slot in the line can be seen as a day in a calendar. Where each day is as long as the time Gmin. The calendar is a year long, which can be resembled as the length horizon. In a calendar, it is possible to do multiple tasks, but there is a limit on how many jobs it is possible to do each day, this is described as with the variable Lmax. In the example below, the timing wheels time horizon is 8s, and the Gmin is one second. This will make the wheel have eight slots, and every slot is 1s.

![Figure 3.2: Time now = 1](image1)

The pointer is moving clockwise and will point at a new bucket each time it spins. In this example, it will first output V1, and when it spins, it will output V2. Vn is a list which can consist of multiple packets up to the number of packet Lmax.

![Figure 3.3: Time now = 2](image2)
When the pointer has pointed to a time it indicates that this time now has become older then the time now, that specific time slot will be updated to represent the time \( \text{now} + \text{horizion} \), on the next spin.

\[ \text{Figure 3.4: Time now = 7} \]

The wheel will grant access to pull packets every Gmin. The length of the whole matrix is the time horizon. The main difference between the timing wheel and the calendar queue is that the timing wheel uses time index. [26],[11]

### 3.1.4 Back preasure

If the algorithm were to be implemented without any feedback, it would face the risk of overflow inside of the matrix, which would cause a significant drop of packets in some scenarios. This is the main argument for having the feedback after the timing wheel output instead of having it directly after the timestamp. If the implementation had been done by having feedback directly after the timestamp, it would face a risk of "head of line blocking."[22]

### 3.2 Memories

In this section the memory of the platform will be shortly introduced.

#### 3.2.1 Block RAM

Block ram(BRAM) is an on-chip memory on the FPGA, the memory itself it quite big compared to for example bunched together D-Flip-flops or distributed RAMs, but they are in general smaller than external memory, like a DDR. They are however a bit slower than a distributed RAM. However, it is highly unlikely that the speed of a block ram will be a big bottleneck of the system. Blocks rams are generally easy to use, and on this FPGA they can be controlled with the help of AXI4 system. BRAM is often used for lookup tables, temporary store data or a FIFO, these are just some of the areas where they can be used. They are placed in the discrete part of the FPGA, and there is an always a limited space of discrete area. Therefore there is always limited amounts of BRAM on
the FPGA itself. On the FPGA that is placed on the F1 platform, there are 2160 BRAM available.[14]

### 3.2.2 DDR4

The DDR4 is a off-chip memory, they are typically quite big, on this platform each memory is 16GiB. The memory is organized into four different parts, Row, Column, Bank and Bank groups.

![Diagram of DDR4 structure]

**Figure 3.5: Top level of DDR4**

In figure 3.5 it is possible to see the top level of a typical DDR4 memory structure with four bank groups, four banks and \( n \) columns and rows.[2] To do a read operation an logical address needs to be provided, and for a write operation it will need data and an logical address. For the memory to handle this address it will need to translate this logical address to a physical address. Inside the memory, there are column decoders, row decoders, buffers and sense amplifiers. The memory itself is a transistor based memory, which will hold the bit value inside with the help of a capacitor.[25][13]
3.2.3 Memory model

This is the memory model that is used for communication between the device and the host.

Host Memory

The host memory is a memory connected to the host only, in this case, the x86 processor, the host has full access to both read and write transactions to this memory. If data would be needed by a kernel, it must be first read from the host memory, then written to the global memory by the host.

Global Memory

Both the device and the host have read and write access to this memory, the host is however in control of the memory and should handle the allocation and deallocation to this memory. If a kernel needs access to this memory, the host loses access to it until the kernel is done with the transactions.

Constant Memory

The host has access to both read and write transactions to the constant memory, but the device also has access to this memory, but only for read transactions. This memory is usually memory chips that are connected to the FPGA.

Local Memory

The local memory is a memory that is used for the device only, for the host to get access to this memory it must be read from the local memory by the device and
sent to the glocal memory where the host can read it. It is typically a block RAM inside the FPGA

**Private Memory**

This memory is only used by the processing elements; it is typically a low latency memory

### 3.3 Measurement and Testing

In any design, testing and verification are always essential to guarantee that the system is running according to the specification [18]. In this design the examination will be done inside the shell, so the shell will act both as the NIC and also the testbench. Since this project will not be implemented on an FPGA, it will just be simulated with models attach to it to mimic the real hardware as much as possible; testing is of the crucial essence to make sure that the design should work on a physical platform as well. The measurement is a very time-consuming practice when it comes to hardware design since the simulation time can become very long, especially when the design has attached models to it.

### 3.4 DPDK-Mbuff

The incoming payload data contains a lot of different information, not only data that is used by the end user. But also data that can be used in the back end program, such as length or timestamp. The incoming data to this system is DPDK-Mbuff which is stored in a NIC before being transferred to the FPGA. However; this will be simulated.[1]
In this chapter the method for the thesis will be presented, what will be discussed is how the rate regulator was implemented and also how the testbench for the system was built.

4.1 Implementing the timing wheel

This part will go over the design aspect of the timing wheel, which is the core of the scheduler.

4.1.1 Circular buffer

The circular buffer is what keeps track on what time index each FIFO has. The length of the circular buffer is the time horizon, the time range for each slot is determined by the variable Gmin, there is also the variable Lmax which determine the maximum height of the wheel. The maximum time horizon will be estimated to be approximately around 150 ms. Since the queue is not dynamic, once something has been placed in the line, it will not be possible to change the transmission time of it. The number of slots in the circular buffer will be around 100000 according to (3.2); this will take 17 bits to describe in signed numbers. The Amazon EC2 F1 does have quite a lot of memory, but it is still necessary to keep this in mind when scheduling. If the circular buffer were only needed to hold a few bits in each slot the best idea would be to use a block ram for the memory since they are both fast and easy to use. But since the packets will be between 1KB and 2KB, the DDR4 will be used. The idea is to divide the DDR4 memory into different sections, where each section will represent a time where each section will have all the data for a time Gmin.
4.1.2 Buffer

The timing wheel consists of a circular queue where every slot in the queue is a linked list. A linked list consists of a head pointer, tail pointer, idle address and a next pointer[19]. To do this, the hardware code would need pointers. Which is lacking good support for in FPGA development.[17] There is also the task at hand in which memory should you save the pointers and where should the actual data be saved, timing will also, of course, be a great issue. It is by no means impossible, but not really necessary either. A linked list also have some function which will not be necessary for this design, since all data in each slot will be outputted in FIFO order. Instead, a FIFO will work as replacement for the linked lists.

At first the idea was to use the block RAMS on the FPGA and have them behave as FIFO, which is easily done with already pre-built protocols, however, they lacked vastly in size. The DDR4 memories have a data rate of 2400 and 72 read/write lines, making the maximum throughput to 19.2 Gbps this throughput can be achieved when there is a constant flow of data coming in the same direction. On the cloud FPGA, there is 4 16GiB DDR4, making it possible to store at maximum 64GiB of payload data.

The DDR4 memories are controlled by an IP interface block provided by Xilinx. The DDR4 memory and the IP block is described in section (4.1.5)

4.1.3 AXI4

The AXI4 is a bus system used widely in Xilinx IP blocks. AXI4 is a memory based bus system, which implies that the bus system is a using a master and slave interface. Where the master device will ask the slave for either a read or a write operation. The master also needs to tell the slave which address the process will occur at. There are different modes which can be used, for the DDR4 memory controller, it is possible to choose a burst mode, where the master will tell the slave how many read or write which will occur. This means that when the burst length is "x," the slave will handle "x" number of read or write operations. The slave will accumulate the starting address, and the last address for the request will be starting address + "x." The block ram used in this implementation is also using AXI4 interface. However, burst mode is not available for these.
In figure 4.1 the read operation for the buss system is described. The master first needs to decide what address the read operation will start from, the length of the burst and also some control signal. When this is done, the slave interface will respond with data and a control signal.

In figure 4.2 the write operation for the AXI4 is shown, just as with the read operation the master will send an address and some other signals. After that, the master can start sending data. There is also an additional channel for a response signal. This is used so that the master knows that the slave has received data.

### 4.1.4 Block ram control block

The Block ram uses an AXI4 bus system. But to simplify the process of reading and writing to the BRAM an AXI4 translator block was built. This block is divided into two different section. One read section and one write part. The write
section takes the data, the address and a active signal. With the help of these signals it will communicate with a AXI4 interface which is connected to the BRAM. It will also output valid signals so that the rest of the system knows when the data can be used. The read section works in the same way, except it outputs data instead of using it as an input. Both of these system will also take a length as an input, and can then increment over the address if necessary. When the operations are completed, they blocks will sent a done signal to the system that has used the block.
4.1.5 DDR4

The DDR4 memories in this thesis are built up in structures of columns, rows, and pages. The memory is the DDR4-2133. In table 4.1.5 the specifications of the DDR4 connected to this FPGA is shown.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>16GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module rank address</td>
<td>1 CS0n</td>
</tr>
<tr>
<td>Device configuration</td>
<td>8Gb (2 Gig x 4, 16 banks</td>
</tr>
<tr>
<td>Device bank group address</td>
<td>BG[1:0]</td>
</tr>
<tr>
<td>Device bank address per group</td>
<td>BA[1:0]</td>
</tr>
<tr>
<td>Row addressing</td>
<td>128K(A[16:0])</td>
</tr>
<tr>
<td>Column addressing</td>
<td>1K (A[9:0])</td>
</tr>
<tr>
<td>Page size</td>
<td>1KB</td>
</tr>
</tbody>
</table>

The DDR4 are externally connected to the FPGA and it is important to note that they are not part of the actual FPGA.

4.1.6 DDR4 control block

To the DDR4 controller, a custom block has been implemented in this project; this block is placed inside the custom logic of the design, this block was made to control the DDR4 controller provided by Vivado. The custom block has two major parts; one send part and one receiving section. The write routine will have the burst length, the data and an active signal as inputs, not counting for clock and reset signals which are inside all the synchronous blocks. This block will work as a controller for the controller; this block will translate the inputs to the AXI4 bus system so that the controller can handle the signals when it receives the wlast signal from the AXI4 that is connected to the DDR4 controller it will deactivate the active signal.

The read routine will work in the same manner, except it will not receive a rlast signal, it will instead generate this itself and turn off the active read signal when it is done. This signal is depending on the burst length.
4.1.7 Memory interface of Vivados DDR4 controller

Controller
The task of the controller is to handle the data; it can receive bursts of data and time the data to and from the CL to reduce the number of dead cycles and minimize the utilization loss and therefore reducing the risk of lower throughput. The controller is connected to the external DDR4s. It is controlled by an AXI4 connection.

Calibration
The calibration block is both for calibration and memory initialization, the task of this block is to set timers and provide initialization routines.

Physical Layer
The physical layer is used to give the DRR4 SDRAM an interface; this is needed to provide the block with the optimal timing to ensure a fast system.

4.1.8 Design constraints
In this thesis we will assume that each payload packet will have a maximum size of 2KB (average size is estimated to be 1350B), so each payload packet will be stored inside two pages. The number of slots will be 112950 as mentioned in (4.1.1) which is possible to describe with 17 bits. Since there are multiple DRR4 memories on the board, in total there are 4 of them there are a few different ways that it is possible to design this, one way would be to use multiple DDR4 memories, and split the data over all 4 to increase the bandwidth, another method
would be to use one DDR4 and use the burst function as much as possible. In this design, the later one is used. As previously mentioned it is assumed that the number of sources will be approximately 150000. But the amount of sources is not what matters in this design, since all the sources will still be gathered in the NIC to the same PICe port. Therefore the data rate is what will matter. To keep the system fast, but as burst free as possible for the NIC the queue height will be kept small, at 255x512bits maximum.

![Figure 4.4: DDR4 Memory](image)

Since there is a lot of memory on this board, everything will most likely not be used. Therefore the memory usage will not be optimized. To represent the number of slots in the timing wheel almost the whole row addressing for one group will be used ([16:0]=131071). This means that the Lmax variable will be [7:0] = 255. Where each payload will be saved inside two 1 KB pages. A new FIFO will be outputted to the network each 16 micro seconds. In the table below a specification of the timing, wheel is shown

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>150 ms</td>
</tr>
<tr>
<td>Number of slots</td>
<td>112950</td>
</tr>
<tr>
<td>Height for each slot</td>
<td>255</td>
</tr>
<tr>
<td>Time period for each slot</td>
<td>16 micro seconds</td>
</tr>
</tbody>
</table>

### 4.2 Timestamping

The sources will come from a User Datagram Protocol(UDP) tunnel, the UDP has a lot of functions but one of the functions inside it a timestamping function. Therefore the first timestamp will already be set. Since the main focus of this thesis is hardware design of the algorithm, the thesis will rely heavily on this
timestamp. The packets that are sent from the tunnel is called a Mbuff packet as described earlier.

4.3 System overview

This is an overview of the whole system. The block called Downlink Data & Scheduled Data is inside the shell, and all the blocks below them are placed inside the custom logic part.

![Diagram of the system]

**Figure 4.5: Overview of the system**

4.3.1 Input buffer

The data that is sent from the testbench to the shell will be sent to 16 BRAMs instantiated in the FPGA in the CL area. The BRAM is dual-port. The first buffer was built with DDR4 memory. The reasoning for using DDR4 memories is because of the size of the memory, the buffer would be able to be very large which would be suitable for further development. The DDR4 controller also uses AXI4s burst which will decrease the number of clock cycles that the read and write instruction would take. However, there are also downsides of using DDR4 memories, since there are a limited amount of DDR4 memories in the custom logic on
the FPGA, as previously mentioned, there are only 3 in the custom logic, while the last one is located in the shell. They are also not dual ported. Since the system is designed for handling burst of traffic, as long as the rate has a limit, there are good arguments for using block rams instead of DDR4. Since they are dual port, it is possible to read and write at the same time, which will be used to give the system a constant flow of packets, as long as there are packets coming into the system. It is important to note that time will always pass, even though no packages are coming into the system. The output will use the same technique by using 16 BRAM with input from the CL and the output to the shell logic. To achieve this the AXI4 connection has been split, the send part of the AXI4 system has been connected to the shell part of the platform, while the receiving portion has been attached to the CL. The link between the CL and shell is the PCIe slave connection.

![Shell Logic](image)

**Figure 4.6: Shell to custom logic communication**

### 4.3.2 Finding the Timestamp

The timestamp is placed at a particular position inside of the packet. A DPDK packet contains two parts, a hex file, and a header structure. The package that is sent from the NIC in this network has the timestamp inside the header, which is a problem since this system is designed by the assumption that the timestamp is stored inside the hex file. Therefore packets that will be used in the testbench for this project will have the timestamp stored inside the hex file instead. This is, of course an inconvenience but moving the timestamp from the header to the hex file is a lot of work, and it is not something that this thesis is about. But it is something that can be done in the future. The time in the header is initially in UNIXTIME, but to keep the register count low, this design will look on the maximum time to transmission instead. For example, if the packet with a timestamp of 100 ms is arriving, the maximum time the package can be scheduled at is at \( t + 100 \text{ ms} \), \( t \) is the current time.

### 4.3.3 Finding length

The length of the packet is not stored in the header, but inside of the hex file. Therefore this can just be read from the package directly and translated linearly to represent the length of the binary packet. Since the range is stored at the
beginning of the hex file, this information can be used to tell the system for how long a read sequence must be active; this is good to use when trying to optimize the system. When the length is found it will tell the address tracker block that it can start.

### 4.3.4 Address tracker

The address tracker does have a lot of inputs to the system, to find the correct address the block needs the length of the packet, the timestamp, communication to an array with all the heights of the rows and also the current time. The central part of the system is the FSM, which is described above. The state machine will remain in its idle state until all the signals are ready. When the signals are prepared, it will go to the "Find Row" state. In this state, the state machine will translate the current time to the same time frame in which the FPGA is working in; this is just a linear multiplication. When the time has been found, it will check what the current time is and give it a row address accordingly. The next step is to check if the current queue slot is full if it is full the packet will be moved backward and will be given less priority, this is done by reading from the array. This process will be looped until it can find a place for the package to fit. When the FSM has noticed a place to store the packet in the machine will update the array at the current row in which was previously calculated and give it the new height $old\_height + current\_height$. In the first design of the address tracker, the FSM was connected to block ram with AXI4 connection. However, this was scrapped. The

![Figure 4.7: FSM of the address tracker](image_url)
reason the block ram was removed is that it was possible to read and write faster directly to an array instead of going through the AXI4 system.

Insertion algorithms

**Algorithm 1:** Insertion of timing wheel

**input:** Payload, Ts(Payload)
// TS = timestamp
// Payload = data to be scheduled

if $Ts > Horizion$ then
    PBT = Time(now)
    if $(FIFO(Time(TS)) != Full)$ then
        TimingWheel[(Time(now+Horizion))] .append(Payload)
        Break.loop
    else
        while True do
            PBT = PBT + Gmin
            if $FIFO(Time(TS+PBT)) != Full$ then
                TimingWheel[(Time(now+Horizion-PBT))] .append(Payload)
                PBT = 0;
                Break.loop
            end
        end
    else
        TimingWheel[Ts].append(Payload)
    end
end

The function of the algorithm is to describe where the data should be placed. If a packet is timestamped at a time that is larger than the current time slot that is assigned at the moment, it should not be set inside of the timing wheel but should be outputted right away. However, if this queue is full, it is not possible to do this action. Instead, the packet must be re-time stamped, but it still is known that this packet is a high priority to be sent. Therefore it should instead be placed in the first available queue slot. Pseudo code for the algorithm is described in algorithm 1

**Exerting a column**

**Algorithm 2:** Exertion of timing wheel

**output:** FIFO at the current time index

while $TW[now].empty()$ do
    Output(TimingWheel[now].PopFront())
end
The output function is quite simple. Every time the timing wheel spins algorithm 2 will be called. The idea behind the function is to output all packets in the current FIFO time slot until the function is done. When reading from the DDR4 memory controller, the controller will receive a burst length which will output the whole column. Therefore there is also need for something which will divide the whole column into specific packets. Consequently, the DDR4 will write to a matrix that is 255x512 where it will be possible to store an entire column.

### 4.3.5 Separating packets

The separate packet function consists of two parts. The first part is a matrix that can store a whole column inside of it. As mentioned, the DDR4 will empty a column with the help of the AXI-burst function. The output port of the DDR4 is then connected to this matrix. After the column has been outputted inside the array the next step is to separate the packets. This function consists of two different phases. The first of which is to find the length of each packet. This is done in the same manner as previously. When this is done, the matrix will output one package at the time to a BRAM memory with AXI communication. Now the shell can read these packets. The separating packet block will then count the amount of rvalid coming from the AXI in the BRAM; once the amount of numbered rvalids are the same as the length of the function it can now safely overwrite the old values inside the BRAM, this will repeat until the matrix is empty. The same idea was used in this section as in section 4.3.1, but this time the read part of the PCIe bus has been connected to the shell part, and the writing part is connected to the CL.

### 4.3.6 Back pressure

The back pressure communicates with the shell and CL with the help of a BRAM connected to the SDA port. The backpressure in this design consists of three different variables a reset signal and a clock signal. First, the system will check if the state is a write state or a read state. If it is in a read state, it will first check if a write is ongoing at the moment, if it is it will not start a new write. After this control it must check if the last write has been written into the DDR4 memory; this is a bit of a bottleneck, but it is necessary to do this wait. If the first packet that is being sent is of considerable length, and also one that needs to be assigned a new address, then it was first appointed to it can take a few extra clock cycles. When the packet has safely been written into the DDR4, the next package can be transmitted from the shell. If the current state is a read state, it must first check if there are any packets inside the block ram that is connected to the input to the shell. If there are then it is safe to read; if there are still no packets, then it must wait. In the first iteration the block was working as described in algorithm 3.
Algorithm 3: Read or write operation algorithm

\[
\begin{align*}
\text{input:} & \quad \text{ReadWriteTime, ReadBramReady, WriteBramReady} \\
\text{if} & \quad \text{ReadWriteTime} == 1 \text{ AND ReadBramReady == 1} \text{ then} \\
& \quad \text{Write "01" to SDA-BRAM} \\
\text{else if} & \quad \text{ReadWriteTime} == 0 \text{ AND ReadBramReady == 1} \text{ then} \\
& \quad \text{Write "10" to SDA-BRAM} \\
\end{align*}
\]

This algorithm was making the system run quite fast, but there was a severe problem with this algorithm. Algorithm 3 was causing packets to drop. This was because there was not enough time for the shell to read the whole BRAM from the CL. It could manage to read up to 180x511 bits with this design depending on the length of all the packets, if the packets where longer it could read more data and if the packets where shorter it could read less. This is because of the time that the separating packet block, which is described in section 4.3.5 needs to be restarted for every new packet that is coming into the system. Therefore the read or write algorithm was remade to work as described in algorithm 4

Algorithm 4: Read or write operation algorithm

\[
\begin{align*}
\text{input:} & \quad \text{ReadWriteTime, ReadBramReady, WriteBramReady} \\
\text{if} & \quad \text{ReadBramReady == 1} \text{ then} \\
& \quad \text{Write "10" to SDA-BRAM} \\
\text{else if} & \quad \text{ReadWriteTime == 1 AND WriteBramReady == 1} \text{ then} \\
& \quad \text{Write "01" to SDA-BRAM} \\
\end{align*}
\]

In most cases algorithm 4 will work the same as algorithm 3, except for when the shell as not been able to read all the packets to the output. This will decrease the rate of the overall system, but it will minimize the packet drops.

4.4 Testbench

The testbench will be implemented in SystemVerilog code which will be connected to the shell of the FPGA. The shell will fetch packets that are being stored inside the instance. The packets that are stored in the instance has information about both the length and the timestamp. The testbench will act like a NIC, it will send packets to the system, and it will also receive them. Therefore the testbench in the shell will follow the rule given by the backpressure. It will only read packets when there are packets to fetch, and also when the backpressure says that the system can receive another packet. In the figure, 4.5 the backpressure is marked as the R/W signal connected between the read/write tracker block and the shell. To bypass using an unnecessary amount of space in the instance, the testbench will only save one line of the packets, which is 512 bits and containing both the timestamp and the length of the packet. The testbench will also keep track on the current time to verify that the schedule has done the work correctly.
The back pressure will be transmitted via a BRAM since the shell has AXI protocol to the CL we must always write to a memory in-between, the ideal would be to route it directly between the CL and the shell, but in this case, it is not possible. The Read/write block is writing to the BRAM every time there is an update on the R/W value. The testbench work in the following way.

1. Setups
2. Reset registers
3. Test loop
4. Output

The first phase of the testbench is to instantiate all the necessary models. In this case, the DDR4 SystemVerilog models are instantiated. This will take a few microseconds. Therefore a simple wait is implemented. The next step is to reset all registers, this might sound trivial and unnecessary, but it is always in good practice to do for every new test run. The test loop is where the actual testing begins. The test loop consists of three different parts:

- Read backpressure
- Write data
- Read data

The loop length can be changed, the length of the loop is decided by both how many values that will be written, and also how long the program should run. The first thing that happens every loop run is that the testbench read values from the BRAM that the backpressure is being stored in. Since the block ram is dual-ported, we can both read and write at the same time. However; we cannot read and write at the same address at the same time. This is being solved by only writing values when they are changed inside the read/write block. And the testbench
is just reading values when it is necessary, but it can still occur at the same time. Therefore every time that the testbench wants to read the value, it will read the value five times, to be sure that it has fetched the value. After the data has been read the testbench will translate the backpressure signal to indicate if it is a read, write or nothing operation. The write operation will check which hex file will be written at this time. This is done by having a generic file name with incrementing numbers in the end. An example of the n'th write operation. The filename is $xyz0000$, then the next file that will be written is $xyz0000+n$, n being the number of time the write operation has occurred before. The testbench will also give the packet time of transmission, this is only for testing and should be implemented in a live run, but it does make the verification easier. To keep the memory on the instance down, this will reset at $n=9999$, after this time the same files will be written again, however, because of that the time will be different it will still schedule this item at a different time. The read operation will first read the first line of the packet, 512 bits. This is because the testbench wants to know the length of the package so that the application will only read original packages and not trash data. The output stage of the testbench will then compare the expected result to the output of the system and tell the user if the packets have been sorted correctly. The test files are created with a Matlab script, which places the length and timestamp at the correct place. The average size of the packets is 1350B, with a minimum size of 640B and a maximum of 2000B. This was measured when creating the test data in Matlab.
5.1 Simulations result

Since the simulation time is incredibly long the simulation time cannot be too large. In most cases the more extensive testing time, the better, but there must always be a limit. In this tests that have been executed the testing time was maximum 11 ms, which took around two days to simulate. However, the corner cases of the algorithm have been tested when being developed; this is just the performance results.

5.1.1 Information about test runs

In total three tests were done, test 1 and 2 were common runtime tests, while test 3 is a corner case, were all the timestamps are placed at the same time.

Test run 1

Average packet size : 1350B  
Simulation time : 0.010935 s  
Number of packets transmitted: 3869  
Clock frequency : 250 MHz  
Timestamp: Randomised

Test run 2

Average packet size : 1350B  
Simulation time : 0.009835 s  
Number of packets transmitted: 3500
Clock frequency: 250 MHz
Timestamp: Randomised

Test run 3
Average packet size: 1350B
Simulation time: 0.009088 s
Number of packets transmitted:
Clock frequency: 250 MHz
Timestamp: 3180

Typically, the pointer will always move forward. But to simulate a corner case where all packets are placed at the same location, the arrow will always point at the same place.

5.1.2 Packets not transmitted in time

To measure if a packet has been transmitted in time the time of transmission is saved inside the payload data itself, since the actual data of the package is not of interest a line of this data is overwritten by the testbench, to make sure that the length of the packet is always the same. To see if the packet was transmitted in time, the testbench will compare the current time to the timestamp of the package and the time of transmission, also with a small offset, since there is time for the testbench to read the result. It also checks both if the packet were transmitted too early or too late. The offset for the packages was calculated by measuring the real-time between every package transmission. The main reason for a packet to be late is if the packet needs to be rescheduled when a packet is rescheduled it will move backward one position. But since the queue length is quite long, it does not occur too often. This is depending on how the timestamp is randomized over all the data, but on "Test run 1" and "Test run 2" they should be fairly similar.

Test run 1
In this run there was in total four packets which were delayed, this makes the system schedule 0.103% packets at a later point that it should have been transmitted in.

Test run 2
In this run there was in total one packets which were delayed, this makes the system schedule 0.026% packets at a later point that it should have been transmitted in.

Test run 3
In this test, there were no packets that were not transmitted in time. This is due to that the write queue will not have enough time to fill up a whole slot before the read function empties the pocket.
5.1 Simulations result

5.1.3 Packet drop

The packet drop is measured by checking how many packets that were transmitted and how many packages that have been received. This will be dependent on how many packages that have been broadcasted so far. As mentioned in section 4.3.6 the system had problems with packets drop due to that the read operation was canceled too early, this was solved with the new read and write algorithm that was mentioned in the same chapter. Exactly how many packets that were dropped in that version is a bit unclear. But it can be roughly estimated, on DDR read that had a burst length of 180; some packets were being dropped. That test run was running for 2300 packets that were transmitted, the number of burst length in which was above 180 in burst length was 11. Therefore it can be estimated to be \( \frac{11}{2200} = 0.47\% \). This means that on approximately every 200 read state there will be packet drops. In the new design, no packet drops were recorded, these measurements were made by hand. There was not a single test run were packet drop were noticed.

5.1.4 Speed of the system

The speed of the system will vary, in the beginning, it will be more moderate than it would be in the end, since the rate will settle. This is because the timing wheel needs time to fill up. In the beginning, the timing wheel might not have any payload data at the current output time, which will make the system look incredibly weak. Therefore the speed of the system cannot merely be measured by calculating the rate of a single transmission, the whole time needs to be taking into account, since rescheduling a packet will take extra time, more extended packages will need more time to be transmitted, different lengths of parcels will take longer or shorter time to be received. In 5.1 how the system is slowed down depending on time. As noticed in the plot, the system has a significant drop in the beginning and then begins to flatten out to around 3.9 Gbps. This plot was made by measuring the speed of "Test run 2" at different times six different times to get an overview of how the rate of the system is depended on how full the queue is.
When simulating a time of almost 11 ms 3869 packets where transmitted. As mentioned earlier, the average length of the packets that are being transmitted is 1350 Byte. The FPGA itself is running at 250 MHz. The speed of the system is calculated according to the equation below

\[
\frac{\text{Number of packets} \times 8 \times \text{avg size}}{\text{total time in seconds}} = \text{speed of system in Gbps}
\]  

(5.1)

If we put the measured values into the equation above the expression is

\[
\frac{3869 \times 8 \times 1350}{0.010935} = 3.82 \text{Gbps}
\]

(5.2)

This speed is measured to when the FPGA is running at 250 MHz. It is possible to use clock recepies to run a clock at 500 MHz, but this function was not tested.

Test run 2

In this runtime test of the system, the packet length is the same, but new random timestamps are generated. The packets were placed into the system for almost 10 ms. This made the system send 3500 packets. The average size was the same. According to the same equation, as used earlier, the will make the system run at

\[
\frac{3500 \times 8 \times 1350}{0.009835} = 3.85 \text{Gbps}
\]

(5.3)

Test run 3

This test had the same speed throughout the whole system; this is due to that the read burst length from the address tracker that is sent to the DDR4 controller
is always around the same size. This is an effect of that the average size of all packets should be 1350B. As mentioned in 4.3.6 when the burst length is around 180 or more, there will be overlap from the read function to the write function. Which will regulate the rate of the system.

$$\frac{3180 \times 8 \times 1350}{0.009088} = 3.79 \text{ Gbps}$$  \hspace{1cm} (5.4)
6.1 Method

To implement a design on a FPGA is a good idea when there is a calculation-heavy algorithm that can be done in parallel. Unfortunately, this algorithm does not contain a lot of parallelisms. What can be done in parallel with each other is some small calculation which is already done quite fast. The FPGA can only handle speeds up to 250 MHz, 500 MHz with a PLL[3]. So a big bottleneck was found quite quickly into the project.

Since the incoming packets is a Mbuff packet there is also already a lot of support for these on CPUs; there are functions for gathering timestamp and length, while on the FPGA they had to be remade. Writing to external memory like the DDR4 is also more easily done with a CPU. When programming an FPGA there are a few different ways to go, a developer can write in a software-based language which will then translate the code to an RTL based language like Verilog, or the developer can write in RTL straight away. In this implementation the programming was written in SystemVerilog, the advantage of using a software-based language like C is that the code is written a lot quicker. However, it is easier to customize the number of clock cycles used for certain tasks in RTL. In the original implementation done by Google, the timing wheel was using a linked list to keep track of the height of the packets. However, since writing a linked list on an FPGA comes with a lot of problems, like where to save pointers, and which memory to allocate the linked lists were replaced by FIFOs. [4].

When making the design for the system, there has been a lot of going back and forth, which in some way has helped to optimize the system. But doing this has also been very time-consuming, in hindsight, it would probably have been better first to develop the algorithm in Simulink, to see what draw back the algorithm has, and where the bottleneck will be in the system.
6.2 Result

The tests that were run on the system were each one of them would be discussed here.

- Packet not transmitted in time test
- Packet drop test
- Speed of the system

6.2.1 Packet not transmitted in time test

This was the easiest one to create and also easy to trust since the measurements are straightforward. However; the test itself is on what kind of data is begin sent in. In "Test run 3" were all data placed in the same slot, and therefore will always be emptied when the next data is being transmitted, not a single late error was noticed. When all information is placed in the same slot and also read from the same slot, this cannot be measured in the same manner as earlier, since it always read from the slot one, if packets were replaced as slot two then it would just never be read back. Therefore a new variable was implemented, that counts how many packages that have been replaced, and this variable remained 0 the whole run. Consequently, we can confirm that no packets were ever delayed.

6.2.2 Packet drop test

This test is tough to measure a result in which can be trusted on. The test itself was measured by hand and not by the testbench itself. The testbench gave the information, but it was too much data to handle to be able to measure. In the first attempt there were noticeable packet drops, and on the second attempt there was no packets drop which could be measured by hand, but when not using computing power for this, the human error should always be accounted for.

6.2.3 Speed of the system

The speed of the system was measured to be around 3.7 Gbps, which is quite fast, but not fast enough. As mentioned, the downlink data in which were to be scheduled is running at a maximum of 9 Gbps. The FPGA itself has a limiting factor to this since the simplest thing would be to increase the frequency, but in this case, it is just not possible. Therefore the design has to be remade in some way to meet the criteria. Since the simulation is incredibly long, it is hard to estimate the final speed of the system. Therefore it will be expected from the measurements that have been made earlier. In 5.1.4 the rate of the system is shown, in picture 5.1 the speed decrease is shown. It has a significant drop in the beginning, and then it begins to flat out after 8 ms slowly. In the best scenario, the simulation would run for 25 days and show how the speed will change after a whole spin around the timing wheel. But to keep the cost down of the project, not only when it comes to money but also time, this has to be estimated. Therefore
"Test run 3" was made. This test shows some average system run when most read sessions will overlap the write by just one or two read sequences. This speed was measured to be 3.79 Gbps. Which is what the final velocity is estimated to converge to.

### 6.3 Using Amazon EC2 F1

The environment is running on a Centos 7 instance, so a basic understanding of Linux is necessary for using this environment. Both SDaccel and Vivado Design suite takes some time to get used to, but they are both quite easy to use and understand. When designing for the F1 in Vivado, there are some rules that needs to followed and also a basic structure. This is necessary so that the AFI can run on the FPGA. This basic structure is pre-made and can be loaded into Vivado. This file can be customized a bit when designing since the design is quite large the simulation is taking up a lot of time. A very time-consuming task is writing to the global memory or using the DDR4. This is a big fall back when it comes to using this application; the simulation time is an incredible time consuming, especially for more significant designs. There is also a setup time for the DDR4 which is not very time consuming when doing a more extensive simulation, but when doing small test runs, they do take up a considerable amount of time.

### 6.4 Future work

There are some different ways that this project needs more work, in this chapter two of them will be discussed.

#### 6.4.1 Implementing on an FPGA

The simulation and the actual synthesized file are supposed to work in the same manner, if all the rules for the FPGA have been fulfilled and also that the FPGA itself is working correctly. In some occasions, this is the case, but in others, there can be some noticeable differences. There is always a risk of misunderstandings when designs custom hardware. Here are some of the cases what can cause problems.[23]

1. The PPL’s has not converged

2. Design rules that have not been followed

3. Not enough testing

4. Noise triggering unwanted signals

5. Hardware that does not meet the model that has been used
These are all issues that can be avoided if the design has been carefully made and that the hardware itself is built according to all the specifications. But the risk is always present. Therefore it is always important to test the design on an FPGA as well and not just in simulations, but unfortunately, because of lack of time, this what not possible in this project.[16]

### 6.4.2 Increasing the speed

A big problem in this project is to keep the system quick enough for the 5G network. And with this design, it will not be quick enough. But there are good chances to improve this. One big bottleneck of the system is the DDR4 memories, and since they are not dual-ported, they do that a lot of time to read and write from. But this is something can be solved by using multiple DDR4 memories. In this design, only one was used, but it is possible with not too much effort to redesign the system to split up the 512x32 bits signal to write 512x8x4 signal each time to four different memories. This will not exactly be an exact linear increase in speed since there is also setup time for the DDR4s when setting addresses, burst length and the other signals. However, this should increase the speed quite a lot. It also does gives the system greater chance to schedule bigger packets for a much longer time.
Bibliography


