# Real-time remote processing enabled by high speed Ethernet

Dumitra Iancu
du8702ia-s@student.lu.se &
Lina Tinnerberg
lina.tinnerberg@gmail.com

Department of Electrical and Information Technology
Lund University

Supervisor: Liang Liu

Examiner: Erik Larsson

June 15, 2023

Printed in Sweden
Tryckeriet i E-huset, Lund

# Abstract

A growing trend within the technologies acting as enablers for 6g, such as Massive MIMO and Large Intelligence Surfaces, is benefiting from both the communication and the positioning aspects that they can provide. As these kinds of systems are employing a large number of arrays which provide high amounts of data, a distributed hardware approach having near-antenna processing is explored in this work. The emulated set-up consists of two FPGA boards, where one is mimicking the local processing which would take place near the LIS panels, while the other aggregates all the data gathered from the panels to have joint processing. The channel state information is gathered through four LIS panels and is processed locally with the help of deep neural networks in order to achieve the position of a user. The next step is sending this data onwards to the board which acts as the central processing unit in order to fuse it, thus achieving a better estimation of the position.

This thesis is proposing a hardware implementation focusing on different optimization techniques that could be used in order to achieve a low-latency and high-throughput system. For real-time applications where latency is critical, such as, for example, autonomous vehicles – a good approach are hardware accelerators tailored exclusively to the function which needs to be implemented. Finally, the communication between hardware units is also managed, with the help of an Ethernet link.

Keywords : Large Intelligent Surfaces, Deep Neural Networks, FPGA, VHDL, Ethernet

# Popular Science Summary

With the rise of *Internet of Things*, which enables communications between smart devices, people have been yearning and thinking about the possibilities that may arise with controlling IoT in real time. This can only be possible by achieving more speed and accuracy within the human-machine interaction. In the technical world, this means combining different growing technologies such as *Large Intelligent Surfaces*,*Deep Neural Networks* or *Systems on Chips*.

As the developing wireless technology is approaching *6G*, the need for hardware supporting this emerging technology is directly increasing. *6G* is promising high communication data-rates and high positioning accuracy, which needs to be supported in some manner by the real-time systems. This work aims to emulate a software distributed system proposed in [1], in hardware. The system consists of four panels situated on each wall of a room which capture the information of how the signal propagates from the user to the panels. This is called CSI – *Channel state Information*. These signals have to be processed rapidly and accurately to get a precise location before, for example, the user moves. With the help of *Deep Neural Networks*, this is achieved in software in [1]. In hardware, one has to consider the limitations of a computationally-intensive algorithm which are often about how much data can be stored at once, how many operations can be performed in parallel or with which accuracy the numbers need to be represented. Moreover, the problem of transmitting data from one panel is also tackled. As the architecture of a distributed system implies having different hardware units which are processing data in parallel which is then sent to a central processing unit, the communication between them will also represent a challenge. All of these dilemmas are handled within this work.

The Field Programmable Gate Arrays – *FPGAs* are an excellent platform for developing new architectures, as it allows for fast prototyping. As the provided FPGAs boards include an Ethernet port, this protocol has been chosen to transmit the data.

# Table of Contents

# Acronyms

| | |
|---|---|
| **AXI** | Advanced eXtensible Interface |
| **AXIS** | Advanced eXtensible Interface Stream |
| **AMBA** | Advanced Microcontroller Bus Architecture |
| **ASIC** | Application Specific Integrated Circuit |
| **CCY** | Clock Cycles |
| **CPU** | Central Processing Unit |
| **CSI** | Channel state information |
| **DNN** | Deep Neural Networks |
| **DMA** | Direct Memory Access |
| **DSP** | Digital Signal Processing or Processor |
| **FPGA** | Field Programmable Gate Array |
| **FF** | Flip Flop |
| **GPU** | Graphical Processing Unit |
| **HDL** | Hardware Descriptor Language |
| **IC** | Integrated Circuit |
| **ISO** | International Organization for Standardization |
| **IoT** | Internet of Things |
| **LIS** | Large Intelligent Surface |
| **ML** | Machine Learning |
| **MIMO** | Massive Input Massive Output |
| **MAC** | Multiply-Accumulate Operation |
| **NN** | Neural Network |
| **OSI** | Open System Interconnection |
| **PCB** | Printed Circuit Board |
| **PS** | Processing System |
| **PL** | Programmable Logic |
| **RTL** | Register Transfer Level |
| **SoC** | System on Chip |
| **ULP** | Unit in the last place |

# List of Figures

# List of Tables

# Chapter 1
# Introduction

One of the most talked about developing technologies of the the last decade is the internet of things, IoT. The vision behind it is a world where every electrical component communicates with the net in order to increase efficiency and automation. A world where our watch records our heartbeat and transmits it to a health app in our phone, where reports from soil moisture sensors optimizes farmers' yields[10], sensors through out the city to allow smart city planning[11] and so on. By 2019 the reported number of connected devices outnumbered the world population, and around a quarter of industries were using IoT technologies in their business[10] and according to all predictions this will just keep on increasing[12][10].

To support this ever-growing amount of devices, the continual development of communication networks is necessary. The newest generation of communications is 5G, which is being continuously adopted all over the world[13], but it is not believed to be enough to support all these devices[12]. Therefore researchers have already began looking at 6G, to both determine what standards are needed to meet the requirements of the world's growing communication demands, and the key technologies that have the potential to allow 6G to meet those standards[12].

A trending research topic in relation to the next generations' communication systems is the ability to combine the network architecture with localisation and sensing procedures. This is practical as modern communication systems have inherent localisation and sensing benefits[14]. 6G, for example, is expected to be made up of large antenna arrays with high spatial resolution[1], and a common infrastructure is therefore beneficial from both a cost and maintenance perspective[1]. Several interesting use cases that benefit from positioning information were also identified with the launch of 5G such as asset tracking, transportation and logistics systems, augmented reality and so on[14].

One promising technology for enabling the high data rate of 6G that is also highly beneficial for a positioning system is massive MIMO[1]. In massive MIMO the number of antennas at the base station is much larger than the number of devices connected to it. This allows for high spectral efficiency as the high amounts of antennas can focus the radiated energy toward the user which minimizes the interference between signals[15]. There are many possibilities in how the antennas could be deployed across an area as demonstrated in Fig.1.1[2]. A concept called Large Intelligent Surfaces, originally a concept of making the environment intelligent by making man-made structures electronically active, could also be implemented through massive MIMO principles by spreading antenna arrays across

the desired surfaces[16].

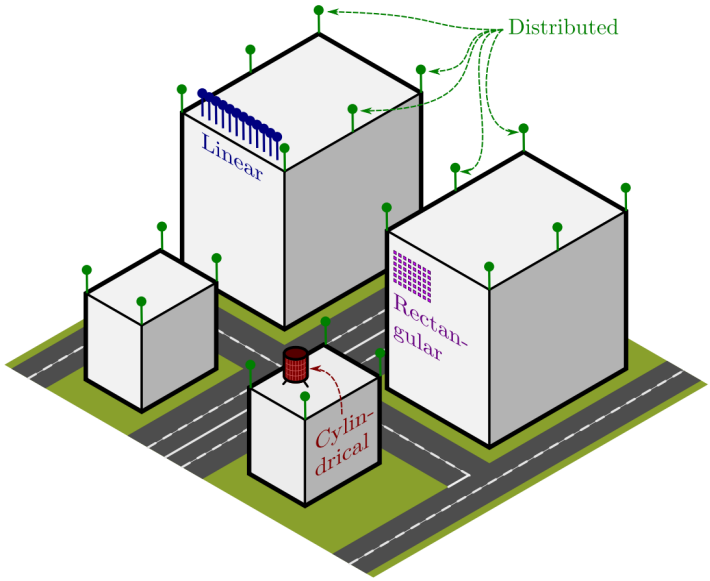



**Figure 1.1:** Different ways to distribute antennas for a massive MIMO system. Picture taken from[2].

Both LIS and massive MIMO provide CSI with high spatial resolution[1] and this data can be used by a machine learning algorithm to determine the position of a connected device[14]. The data provided by both the LIS and the massive MIMO system are low-level raw measurements about the state of the communication channel. It's very hard to build an accurate mathematical model for that, but as there is plenty of available data ML doesn't face the same problem and can achieve high accuracy[14].

However, there are many challenges for these techniques that need to be solved before they can reach their full potential[14][2][1]. One interesting thing to look at is what the hardware demands for this type of system would be. As the systems are composed of antennas deployed at multiple locations the data needs to be transmitted to a CPU. This makes the necessary interconnection a key bottleneck for any implementation. In order to lessen the demand for high interconnect bandwidth a panelised LIS connected with a tree topology has been proposed. Panels could then utilise local processing, to reduce the amount of data that needs to be transmitted. The data from different panels can be aggregated together along the transmit paths, further reducing the needed bandwidth compared to a centralized approach where the raw baseband samples are directly transmitted to the CPU. The amount of needed interconnection bandwidth is still likely to be high, which makes it vital that the positioning part of the system also tries to minimize the data it needs to transmit[1].

Another main bottleneck will be the amount of computational resources needed[1]. ML algorithms are notoriously computationally heavy, memory intensive and re-

source consuming[17]. Due to this, it is vital to look into the current research for how to optimize the resource utilization for ML systems while still achieving high throughput and utilize all the available techniques that are suited for the desired system. Position applications often also require low latency to work as intended, which create yet another constraint that the design needs to be optimized for.

What this work proposes to do is to explore the performance that can be achieved for a distributed system, while catering for both the computational and communication demands that arise.

## 1.1 Background Work

This thesis aims to do a hardware implementation of a positioning system presented in [1]. A NN positioning system has been designed which is meant to work concurrently with a LIS system as described above.

Previous works have presented neural networks that enable wireless positioning, but most of these are processed centrally while the one presented in [1], is distributed. The concrete difference is that the NN runs locally on the antenna panel and outputs a probability of the position. Thus, the raw data from the massive MIMO systems is not sent to the central processing unit anymore. The four resulting probabilities are then fused together on the CPU, yielding a combined probability function and a more accurate estimate for the position.

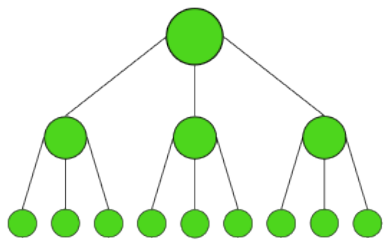


**Figure 1.2:** An example of tree topology where every node has three children. The amount of children can vary between different systems.

This implementation reduces the necessary interconnection bandwidth massively. It is also simple to scale, as the fusion outputs the same type of probability function as it demands as input. If the panels are connected with a tree topology as displayed in Fig. 1.2 fusion can be done in every non leaf node thus making sure that every node only sends one probability function to its parent no matter the amount of children.

The scenario considered in the article is when four separate antenna arrays are used, containing 64 antennas each, and deployed in a square room as displayed in Fig. 1.3. The data for the scenario is simulated as the corresponding massive MIMO set up has not been implemented. An interesting topic to explore is running the system with acquired data from the real system.

Two FPGA boards are used to simulate the scenario presented in [1]. Two NNs are implemented on each board and a Ethernet communication link is established

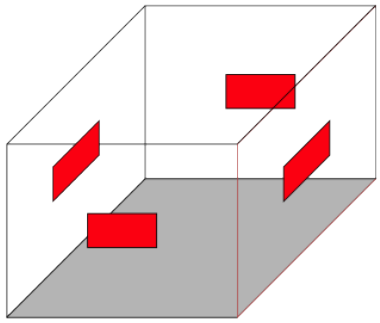


**Figure 1.3:** The set-up of 4 antenna arrays in a square room.

between them. One of the boards acts as the CPU, containing the processing for the fusion as well as for two of the NNs.

The simulated massive MIMO system is set to generate an up-link pilot, the data used for the positioning system, every 0.5ms. It's assumed that there are 16 simultaneous users. This gives a throughput requirement of 16 positioning inference calculated every 0.5ms or 32 000/second. As it is a real time system, it is desirable to minimize the latency as much as possible.

## 1.2 Thesis Structure

The thesis, excluding the introduction, is organized as follows:

- **Background** – In this chapter theoretical aspects of digital hardware design targeting FPGAs are presented.
- **Deep Neural Networks** – Here, the theory behind the most used types of neural networks is introduced as well as the necessary techniques to implement these efficiently in hardware. Moreover, details about the implementation on the RFSoC2x2 FPGA are presented.
- **Gaussian Probability Fusion** – Aims to familiarize the reader with the concept of conflation of gaussian probability distributions, what operations are needed in hardware and what compromises have been made regarding accuracy and speed.
- **Communication System Using Ethernet** – This chapter describes the software implementation of the Ethernet on the Processing System.
- **System Integration** – is about how all the parts described previously come together to emulate the scenario of a real-time system with 4 LIS panels.
- **Results** – here, different results of the subsystems are presented and discussed.

# Chapter 2
# Background

This thesis presumes that the reader has a basic understanding of the theory behind digital hardware design, but if not, this chapter will briefly cover some key concepts regarding it. More advanced concepts and concepts specific for digital design targeting FPGAs will also be covered in this chapter.

## 2.1 Introduction to Digital Hardware Design

The world of digital hardware design has evolved rapidly the last century as the capability of the enabling technology has developed exponentially. This rapid development has made it necessary for digital hardware designers to develop many different techniques and to divide the work into several different abstract hierarchies. To get a wider context of how these techniques are related to a finished design it is good to take a brief look at how the technology has developed in the last half century.

The underlying technology of all modern digital, and analog, hardware is the transistor which was invented in 1947. In 1958 the first integrated circuit, a structure in which all components were integrated on a one semiconductor substrate, was conceived. Then in the 1960s Gordon Moore made a prediction that later came to be called Moore's law, which states that the number of transistors that can be integrated on a single die would grow exponentially with time[18]. This turned out to have held true, with integration complexity doubling approximately every one or two years until just recently when it has slowed down to every three years[19].

This massive increase in the amount of transistors per chip has made it necessary to adopt new and more advanced techniques for digital design. In the early years, every transistor could be laid out and optimized individually. This approach is clearly no longer feasible with billions of transistors on one chip. Instead, a hierarchical approach has been adopted, in which the levels are designed to interfere with each other as little as possible and where as much as possible in the lower levels can be reused[18]. The used hierarchy can be seen in Fig. 2.1

Engineers working in the lower levels create a library of standard cells which the RTL code can be mapped to in an automated process called synthesis. These standard cells can be more complicated modules like full adders or simpler gates like an *AND* gate. This means that cells with the same functionality, using the

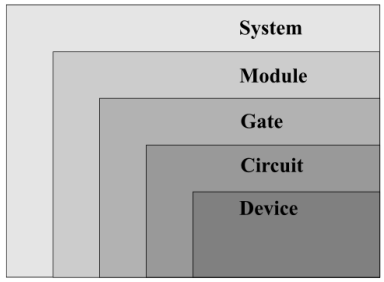



**Figure 2.1:** The different hierarchy levels of digital hardware design.

same technology, do not have to be designed multiple times for different systems and can instead be reused. The performance of these cells has to be taken into consideration by the system level designer as it impacts the performance and timing of the larger design. The last step is usually called place and route, or implementation in designs targeting FPGAs, consist of an automated process that places these cells and creates the appropriate routing between them[18].

This work process follows a divide and conquer approach and allows system designers to consider the cells as black box systems following certain specifications instead of having to deal with their full complexities. The level of most interest to us is the system level in which an overarching description of the system logic is described. This is done using a HDL to write RTL code, describing the behaviour of the digital signals, and the operations done upon them, between hardware registers. In this thesis, RTL code is written to design the different hardware systems and automated tools for FPGA design is used to map it to the lower levels.

## 2.2 Digital design for FPGAs

When doing a hardware design different platforms can be targeted, either an ASIC or a FPGA. An ASIC is a IC designed for a specific task, while an FPGA is an already produced chip composed of configurable logic blocks connected via programmable interconnects[20]. This allows the FPGA to be reprogrammed to implement whatever hardware design the user desires, as long as the available resources are enough. The FPGA chip is often put onto a larger PCB that can contain a varied amount of extras, like a usb port, an ethernet port, off-chip memory and so on.

The implementation process for both ASIC and FPGA usually starts with writing RTL code. For an ASIC, the RTL code will then be mapped to a set of standard cells, which depends on the library used, while for an FPGA the RTL code will be mapped to the available resources on a chosen FPGA. The resources available on a FPGA depends on which vendor the FPGA was bought from and what are the FPGA's specifications. The target FPGA for this thesis is a Xilinx FPGA and these following components are the most relevant for this thesis.

- **CLBs -** These are also called LUTs by Xilinx's tool, *Vivado*. These are the core components of the FPGA, the logic blocks that can be configured to follow the hardware specifications in the RTL description.
- **DSP slices -** These are hardware accelerators for multiply-and-add operations, and will be presented in more detail further down in this section.
- **FF -** These are simple FFs, the registers written in the RTL code are made up of these components.
- **Block RAM -** These are specifically useful when larger memory chunks are needed. When specifying the available memory in a FPGA, both the FF capacity and Block Ram capacity are often calculated together in the same category which is showing the available on-chip memory.

When creating and designing a digital system targeting a specific FPGA, it is important to start by checking the resources available on the target FPGA, since that is an inherent limiting factor. Another important factor is that a FPGA is not usable at a 100% utilization rate, as the routing – the interconnects between the different components – becomes extremely hard to execute and usually does not allow the system to fit within timing constraints.

When evaluating the performance of a digital system there are several important factors. In the design stage it's important to decide which of these factors are a priority, as improving one factor in a design leads to worse performance for another one.

- *Throughput* – Throughput is one of the aspects of performance the digital design. It measures how much data gets processed per time unit, usually per clock cycle or per second.
- *Latency* – Latency is the second aspect of performance. It measures the time it takes for a package of data to travel through the system.
- *Resource Utilization* – In a FPGA design, this keeps track of how much of the available resources are used. It is also included to keep track of how well the used resources are utilized.
- *Power Usage* – This measures the amount of power used by a system.
- *Accuracy* – This metric is particularly important in arithmetic systems. The accuracy of an arithmetic operation depends on many factors, such as the number representation or the chosen algorithm for a more advanced operation. These choices impact the previously mentioned factors.

### 2.2.1 DSP slices

FPGAs are significantly used in Digital Processing Systems, as the custom algorithms that can be implemented can achieve a high level of parallelism. DSP systems are extensively using MAC operations, which can be both time-consuming and resource-demanding to implement with standard CLBs. Therefore, to keep high performance, it's better to have dedicated resources for these operations, and this is the function the DSP slice provides.

The DSP slices are a core component of Xilinx FPGAs. It is a dedicated DSP processing block implemented in full silicon. In the UltraScale architecture the DSP slice used is the DSP48E2(source). A block diagram showing the base functionality of this DSP slice is presented in Fig. 2.2[3].

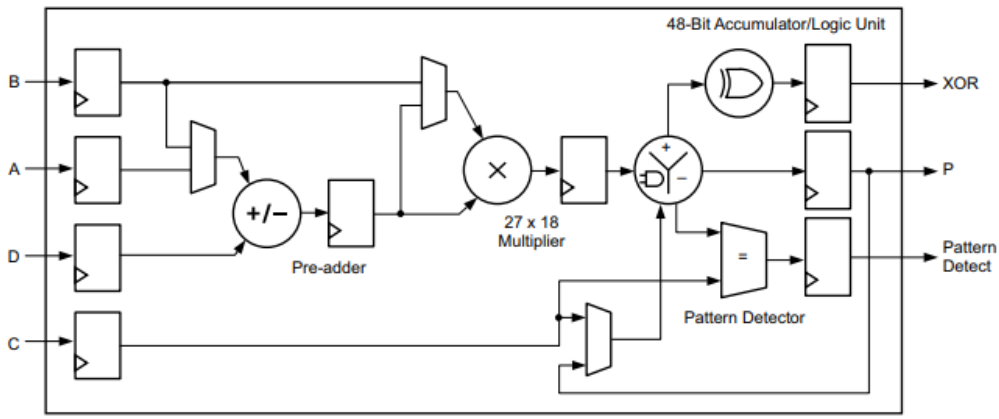



**Figure 2.2:** The base functionality of the DSP48E2[3].

For example, Xilinx mentions in their manuals applications such as wide dynamic bus shifters, memory address generators, wide bus multiplexers, and memory-mapped I/O registers. Another area in which they have proven efficient are Neural Networks, as they are mainly employing matrix multiplications which are basically being translate into a multitude of MAC operations. The multiplication unit inside the DSP slice is 27x18bits wide[3], and this uneven number can be utilized to optimized the DSP slice to run two concurrent INT 8 operations[21].

### 2.2.2 Target FPGA

The target FPGA for this thesis was originally the Zynq UltraScale+ MPSoC ZCU102 board. Due to issues with licensing, which will be described in detail in the Ethernet section, it was changed to the Zynq UltraScale+ RFSoC XCZU28DR, also called the RFSoC 2x2. This was changed quite late in the work process for the thesis, which means that large parts of the system was design in accordance to the available resources for the ZCU102 board. Therefore the available resources for both the boards will be presented in the section.

The available resources in the ZCU102 board are presented in Table 2.1 and the available resources in the RFSoC 2x2 is presented in Table 2.1. The three types of resources that are relevant in this project is the *System Logic Cells*, also referred to as CLBs earlier, *Memory*, which includes both FFs and Block Ram, and the *DSP slices*. The RFSoC 2x2 board has more available resources in all categories, which means that all designs that fit on the ZCU102 board also fit on the RFSoC 2x2 board.

The FPGA chip on both boards belong to the Zynq UltraScale+ series, which are all System on Chips. A SoC refers to an IC that integrates many different parts

| System Logic Cells(K) | 600 |
|---|---|
| Memory (Mb) | 32.1 |
| DSP Slices | 2520 |

**Table 2.1:** The Resources available on the ZCU102 board[6].

| System Logic Cells(K) | 930 |
|---|---|
| Memory (Mb) | 60.5 |
| DSP Slices | 4272 |

**Table 2.2:** The Resources available on the RFSoC 2x2 board[7].

of a computer on one chip. Xilinx SoCs are chips that integrate an FPGA unit, an arm processor and a I/O programmability on the same chip. The arm core can be configured to run bare metal or run a Linux variant. The FPGA fabric of the ZYNQ UltraScale+ is referred to as the PL and the arm processor is referred to as the PS.

Xillinx has two separate tools that can be used to program the PS, Vitis and PYNQ. PYNQ cannot be run in a bare metal configuration as it requires an operating system that contains a set of software libraries. These operating systems are read from an SD card on the board, and for some boards, Xilinx provides already generated SD images with it. The RFSoC 2x2 board is one of these, which makes PYNQ a simple option for controlling its PS. For other SoC boards, like the ZCU102, these images can be generated through a Xilinx tool called *Petalinux*.

## 2.3 Arithmetic in Hardware

To implement complex arithmetic operations in hardware it is necessary to break the operation down into simpler sub-operations that can be mapped to the available resources. For FPGAs these smaller sub-operations are often addition, subtraction, multiplication or shifting. The sequence of these sub-operations are then detailed in an algorithm. For most complex arithmetic, there are several different standard algorithms that can be used, with different advantages and disadvantages. In the following subsections, several algorithms for more complicated arithmetic operations which are used later in the thesis are presented.

### 2.3.1 Division Algorithms

When broken down, division is essentially a series of subtractions, where the dividend is repeatedly subtracted by the divisor to calculate the reminder and the quotient. There are several different algorithms for division and they can be divided into two main groups, slow division and fast division. The slow division algorithms are simpler and calculate a bit of the result at a time through a series

of subtractions, while the fast algorithms start with approximating the answer and then calculates two bits of the result every iteration. Both a slow and fast division was implemented for this thesis in different sections[22].

### Non-Restoring Division algorithm

The two most common algorithms among the slow algorithms is the Restoring Division algorithm and Non-Restoring Division algorithm. The two algorithms are very similar both in execution and resource requirements. Both are executed iteratively and one bit is calculated per iteration[23].

The Restoring Algorithm has a somewhat higher resource utilization due to the nature of the algorithm. As in it both subtraction and addition can be done in the same iteration, while the Non-Restoring Algoritm only does either one addition or subtraction per iteration. Due to this the focus was on the Non-Restoring Algorithm, and the steps that were followed while implementing this algorithm are explained in detail in [23].

### Newton–Raphson algorithm

Newton-Raphson division falls into the fast-division category of algorithms. Its core is the Newton–Raphson method, which produces the root approximation of a real–valued differentiable function in an iterative manner. The algorithm starts with an initial guess $x_0$, which will determine the convergence of the algorithm. Based on the successive approximations, the tangent line at $(x_0, f(x_0))$ is constructed and the next iterative value, $x_1$ is approximated, by taking the intersection of the abscissa with the newly found tangent line.

For the general case, given the differentiable function $f(x)$, when the tangent line to the curve $f(x)$ at $x = x_n$ intersects the x-axis at $x_{x+1}$, then the slope becomes[4]:

$$f'(x_n) = \frac{f(x_n) - 0}{x_n - x_{n+1}} \tag{2.1}$$

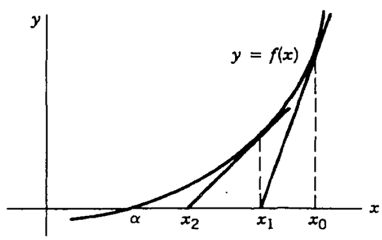



**Figure 2.3:** The geometrical principle of Newton's approximation [4]

Rearranging the terms,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{2.2}$$

where $x_n$ is the current approximation, and $x_{n+1}$ is the next approximation. A graphical interpretation of the algorithm can be seen in Fig. 2.3.

For the case at hand, the division can be seen as a multiplication with the inverse of the divisor. Therefore, the equation that needs to be solved, $f(x) = \frac{1}{x} - y$, where $y$ is the initial approximation, becomes:

$$x_{n+1} = x_i(2 - y \times x_i), \tag{2.3}$$

thus arriving at a series of additions and multiplications from a hardware perspective.

As stated previously, the convergence speed of the algorithm depends on the initial estimated value, which, if not chosen adequately, can pose a disadvantage when choosing the fast division algorithm. From equation (2.3), it can be intuitively deducted that the error decreases quadratically with each iteration. The advantage of the Newton–Raphson method is the speed and the relatively accurate achieved result.

### 2.3.2 Logarithm and exponential approximation using Taylor series

The Taylor expansion around a certain point $x_0$ gives the opportunity to approximate any function within a small neighbourhood with the help of polynomials[24]. Given a function $f(x)$, the corresponding Taylor polynomial around a point $a$ and order $k$ is

$$f(x) = \sum_{k=0}^{\infty} f^{(k)}(a)\frac{(x-a)^k}{k!}. \tag{2.4}$$

Since polynomials are essentially a series of multiplications and additions the problem of describing non-linear functions in hardware becomes a compromise between having enough accuracy by considering enough Taylor terms at the cost of resources.

For the exponential function, equation (2.4) becomes:

$$e^x = e^a \sum_{k=0}^{\infty} \frac{(x-a)^k}{k!}, \tag{2.5}$$

and for the logarithm:

$$ln(x) = ln(a) + \sum_{k=1}^{\infty} \frac{(x-a)^k}{a^k \times k!}. \tag{2.6}$$

The point of expansion has been chosen according to the range of values in which $x$ resides, thus aiming for a high accuracy for a specific interval of values.

### 2.3.3 Number Representation

Another important consideration when using arithmetic operations in hardware is how to represent the numbers. In order to be able to reach an accurate result when processing the real data fed into the neural network, a strategy of how to represent and manipulate real numbers has to be employed. When doing signal processing, decimal numbers are usually represented and computed either by using fixed-point or floating-point arithmetic. There are certain aspects to weigh and consider when picking the right notation for the design in cause.

- *Dynamic Range and Precision* – The floating-point arithmetic inherently possesses the ability to cater to a much bigger dynamic range – that is,

the smallest and biggest numbers that can be represented. The "gaps" between the numbers or the *ulps* can be, therefore, bigger in a fixed-point representation resulting in a more pronounced rounding-off error[25].

- *Cost and resource utilisation* – Since the fixed-point arithmetic usually requires fewer bits for the same accuracy as the floating-point can achieve with more bits for a given number range, the fixed approach is more computation and memory effective. Usually, floating-point arithmetic requires substantially more gates than fixed-point.
- *Performance* – The speed with which a processing unit executes operations is an essential consideration. Depending on the application, floating-point arithmetic might be more efficient for certain algorithms that are computationally extensive[25], but fixed-point can be deemed comparable since it requires less computational power but being capable of achieving similar results.

When considering which type of number representation to use, one ought to ponder the requirements of the application at hand. While fixed-point is cost-efficient and can prove accurate for a certain number span, there is always a limitation given by the number of fractional bits. Thus arises the need to perform floating-point computations which give more flexibility in regards to the dynamic range.

Following the analysis we have conducted, a mix of both fixed-point and floating-point ought to be used to be able to accurately and efficiently describe the neural network and the probability distribution fusion computations.

### Fixed-point representation

As the real numbers have a fractional component, a way had to be found in order to be able to represent them in digital hardware. All the numbers are stored in binary words and in fixed-point they have a length, a binary point and are either signed or unsigned. A fractional number $N$ can be represented with an infinite number of bits in binary form in the following manner:

$$b_n b_{n-1} b_{n-2} ... b_2 b_1 b_0 . b_{-1} b_{-2} ...$$

When choosing the parameters for fixed-point arithmetic, such as word length and fractional length, there are certain aspects that need to be taken into consideration such as precision, resolution and dynamic range. **Precision** is the maximum number of non-zero bits representable, and for fixed-point this value is the word length. The **resolution** is the smallest representable value, and it is given by

$$\text{resolution} = \frac{1}{2^f},$$

where $f$ represents the number of fractional bits. The **dynamic range** is the ratio of the maximum absolute value and the minimum positive absolute value that can be represented[26].

Another aspect to consider is how the numbers behave when doing multiplications and additions. For **fixed-point addition**, if two numbers that have the same amount of bits are considered, then to preserve the correct added value in case of an overflow, one extra bit is needed. To generalize, if we have $N$ number of additions of numbers, then the result requires the following $M$ number of extra bits:

$$M = \log_2 N.$$

For **fixed-point multiplications**, if one needs to multiply two numbers of $A$ bits and $B$ bits respectively, to preserve the full-precision of the result, $A + B$ bits are needed. Different strategies are implemented for different layers, which will be discussed in the later sections.

### Floating-point representation

In contrast to the fixed-point notation, in floating-point the numbers are depicted in a way which allows the binary point to not be fixed in a certain position. Generally, a floating-point number can be represented as the following[27]:

$$x = (-1)^s \times m \times \beta^e, \tag{2.7}$$

where:

- $s \in \{0, 1\}$ which represents the *sign* of the number;
- $m$ is the *significand* or the *mantissa* of the representation;
- a base $\beta \geq 2$;
- $e$ is the exponent of the number such that $e_{min} \leq e \leq e_{max}$. These extremal exponents are usually specified as being $e_{min} = 1 - e_{max}$.

Oftentimes, some numbers may not have unique representation when using this notation. In order to solve that, one can use *normalization*, therefore choosing the depiction in which the exponent is minimum, and consequently, there exists no leading zeroes[27]. If we consider the base $\beta = 2$, then any number can be seen as[28]:

$$1.xxxxxxxx_{two} \times 2^{yyyy}$$

When designing floating-point arithmetic in hardware, one must find a compromise between precision and range – meaning a compromise between the size of the fraction and the size of the exponent. Increasing the size of the fraction adds precision to the number, while increasing the exponent extends the range[28].

The *IEEE 754 floating-point standard* which is found in virtually every computer adds different elements to the base representation increasing the portability as well as the quality of computer arithmetic[28]. For example, the standard makes the mantissa 24-bits long by making the leading 1 bit of normalized number, implicit. Moreover, the IEEE 754 can employ different rounding modes where different rounding strategies of the numbers are employed[27]. For simplicity, these features were left out of the implementation of the system.

Floating-point arithmetic requires additional hardware compared to fixed-point arithmetic due to the fact that the numbers have to go through *normalization* or *denormalization*. For example, for **floating-point addition**, numbers have to be rescaled to the same exponent to achieve the correct result of the addition. The *normalization* of the numbers is an expensive operation since it is basically implementing a general shifter. After that, the mantissas are summed with a basic integer summation. The **floating-point multplication** is less hardware demanding as the significands can be multiplied immediately without any pre-normalization. The resultant exponent is calculated by adding the operands' exponents.

## 2.4 On Chip Communication Protocols

A complicated digital design is divided into multiple functional blocks. Some blocks might be custom designed for a very specific function, while others implement a general functionality that can have wide usage. Xilinx offers these kinds of general-usage IPs that have their own communication protocol, which is a standard defining how their blocks transmit and receive data.

The on chip communication protocol used by Xilinx is the *AXI protocol*, which was created by ARM as part of their AMBA standard. There are multiple variants of the AXI protocol as it has been in use for many years, but the ones used by most Xilinx IPs belong to the fourth version of the AMBA standard. The version is made up of three different protocols, the full AXI4 protocol, the AXI4-lite protocol and the AXI-Stream protocol[29]. Only the AXIS and AXI-lite protocol is employed in this thesis, and they are presented in detail further down.

There are some things in common to all three AXI4 protocols. They all have two types of ports, subordinates and managers. These interfaces are symmetrical, and a connection is always formed between subordinates and managers. This makes the integration of IPs relatively simple[30].

Another thing they have in common is that a data transfer is always preceded by a handshake mechanism which is the same for all three protocols. It is based on the use of a valid and ready signal, the valid originates from the source of the data and the ready originates from the destination. The valid is asserted when the source is ready to transmit data and the ready is asserted when the destination is ready to receive new data. Both the valid and ready signal can be asserted first, but the transmission of data is only valid when both are raised. This handshake is not asynchronous so it also requires the rising edge of the clock for the handshake to be completed[31].

Both the subordinate and the manger port can work as a source and destination depending on where the data originates from for the specific transfer as specified in the individual protocol.

### 2.4.1 AXI-lite Protocol

The AXI4 Lite protocol defines a simple low throughput memory mapped interface. The protocol defines a series of transfers for both reading and writing to the memory that includes the transmission of the address.

Many of Xilinx reconfigurable IPs are controlled by status and control registers coupled to an AXI-lite subordinate port. It connects to a manger port that can either be implemented in the FPGA logic or if the FPGA is a SoC it can be connected to a manger port in the PS. When controlling it through a program written on the PS, different classes exist that remove the need to know all the signals defined in the protocol in detail. In this project the IPs with AXI-lite ports are all connected to the PS.

### 2.4.2 AXI-Stream Protocol

The AXIS protocol defines an interface made for high speed streaming data. It defines several different signals wherein many are optional[8]. In this thesis only four signals, excluding the clock and reset signal, are used and these are presented in Table 2.3. In the AXIS protocol the manager is always the source and the subordinate is always the destination.

| Signal | Source | Description |
|---|---|---|
| TVALID | Transmitter | Indicates that the data is ready to be transmitted. |
| TREADY | Receiver | Indicates that the receiver can accept new data. |
| TDATA | Transmitter | The signal transmitting the data. It has a variable width with recommended values being 8, 16, 32, 64, 128, 256, 512 or 1024. |
| TLAST | Transmitter | Indicates the last valid data for a package. |

**Table 2.3:** The four used signals in the AXIS Protocol and their function[8].

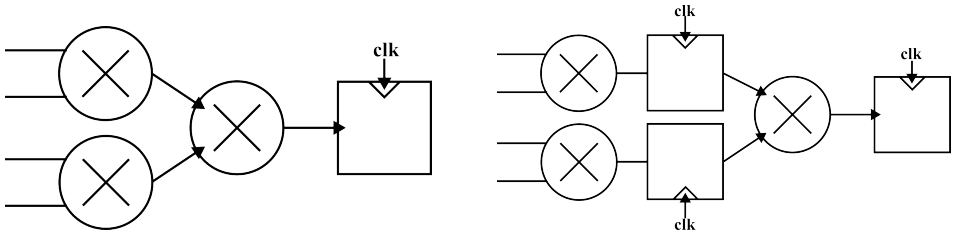
## 2.5 Pipelining

One important technique to improve the performance of a system is pipelining. The idea is to run several tasks simultaneously, therefore increasing the throughput of a design[32].

In Fig. 2.4a a simple non-pipelined data path is shown and in Fig. 2.4b the same data path is showed but pipelined. In Fig. 2.4a the clock has to be slow enough to allow two multiplications per cycle while in Fig. 2.4b it can be twice as fast, thus increasing the circuit's throughput. Theoretically, the design's throughput will continue to increase as more pipeline stages are added, but in practice there is a limit. This is due to both set-up and hold time of the register, which has a larger impact on a higher clock frequency, and due to the fact that it is often to impossible to keep dividing up the design into these pipeline stages[32].

The latency of the system is often negatively impacted by pipelining, though executing it well can decrease this impact. In the example above, the latency of the system would not be affected as the tasks can be divided evenly into the pipeline stages, but most times latency would increase, as one pipeline stage would be more time consuming than the others[32].

A digital system often has a target clock frequency and if a design has long data paths, pipelining is a useful technique to ensure that the design meets its timing requirements.




**(a)** Non-pipelined data path. **(b)** Pipelined data path.

**Figure 2.4:** Pipeline strategies

# Chapter 3
# System Overview

This chapter covers a detailed description of implemented system as whole, alongside the interface between the developed IPs and the data generation.

## 3.1 Implementation

Following the idea of a distributed system, two RFSoC2x2 boards have been used to emulate the idea of distributed computing for the LIS panels. One board represents the sending side, depicted in Fig. 3.1 and the other board represents the receiving side, shown in Fig. 3.2. Each board contains two *NN hardware accelerators*, each accelerator mimicking the local processing on a LIS panel. The receiving board also acts as the central processing unit performing a *Gaussian Fusion* with the aggregated data from the four NNs to calculate the final position.

The three main modules of the system, the Neural Network, the Fusion and the Ethernet connection between the FPGAs are presented in chapter 4, 5 and 6. In order to obtain a functioning system on the SoC, these three parts require smaller interface modules acting as bridges between the different types of data and on-chip communication protocols.

### 3.1.1 The sending side

The set-up on the sending side as well as on the receiving side has been created to accommodate all the implemented programmable logic within the resources of the used FPGA. Thus, on the transmitting side, there are two NNs corresponding to two LIS panels and certain blocks which are managing the interfacing with the NNs and DMA, or are generating the data. These will further be explained briefly.

- **M1/M2** are in fact, one single IP and their task is storing and sending the data required for the input of the neural network. Therefore, they act as a memory comprised of the CSI data needed for the prediction of the position coordinates. To emulate the real-time generation of the data, these send onwards 16 inputs each every 2.5ms. Moreover, these memories are controlled with the *Jupyter Notebook*, in the sense that they are enabled through python with the help of an AXI Lite port.

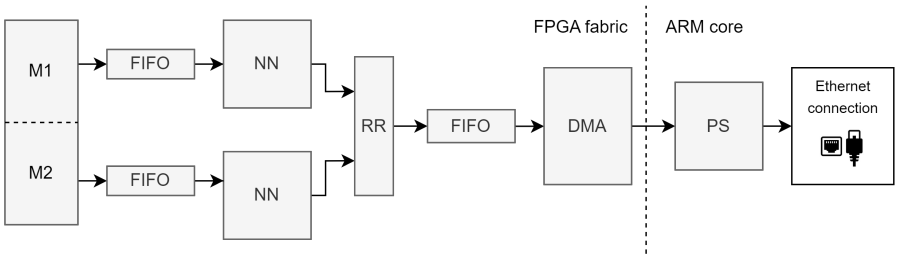



**Figure 3.1:** Block diagram of the sending side, where *FIFO* is a memory of the first-in, first-out, *RR* is a round-robin router, *DMA* is direct memory access and *PS* is the processing system.

- **FIFO** stands for *first-in, first-out* and is a Xilinx IP that acts as a buffer. This IP is parametrizable and it can be chosen to be used with the full AXI protocol or just with certain signals.
- **NN** is the neural network implemented which is presented in depth in chapter 4.
- **RR** is a *round-robin* router which handles the AXI interface between the NN and FIFO. In addition, it packages the 16-bit outputs from the NN into a 32-bit number so that it can be easily received by the DMA.
- **DMA** is the *direct memory access* IP provided by Xilinx, which is controllable through python. It is a centeral part of the communication system and presented in depth in chapter 6.
- **PS** is the *processing system* on which python is ran on. Here the Ethernet connection is handled via the socket class.

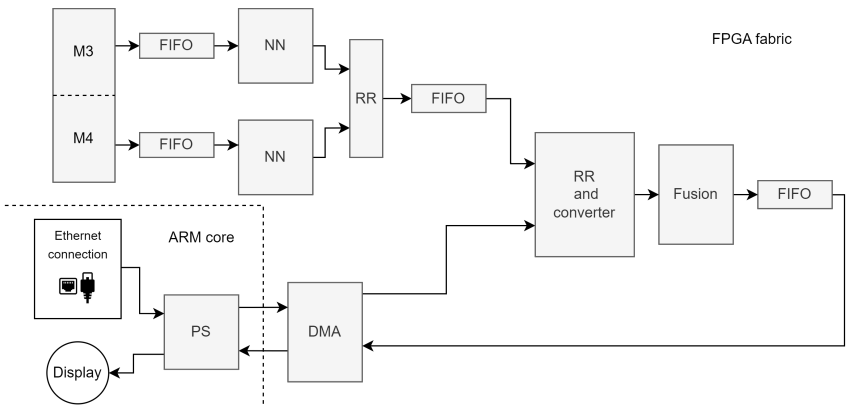



**Figure 3.2:** Block diagram of the receiving side, where *FIFO* is a memory of the first-in, first-out, *RR* is a round-robin router, *DMA* is direct memory access and *PS* is the processing system.

### 3.1.2 The receiving side

The board which will be receiving the data from the two LIS panels has a similar overlay, basically containing all the elements from the transmitter side. Additionally, as it can be seen from Fig. 3.2, consists of:

- **RR and converter** whose task is to unpack and store the output of the four NNs received from the Ethernet as well as from the NNs present locally. Moreover, it converts the 16-bit fixed-point numbers in floating-point equivalents, preparing the values for the fusion.
- **Fusion** which aggregates all the data from the panels with the aid of the conflation operation yielding the approximated mean values for the position. This was presented in detail in chapter 5.
- **Display** which is a software function that basically prints the values from the hardware sent via DMA in a readable manner.

# Chapter 4
# Deep Neural Networks

In this chapter the theory behind Neural Networks will be covered briefly. Then the NN that shall be implemented is presented in detail. Some general optimization methods for implementing NNs in hardware are mentioned and how they apply to the case at hand. Lastly, the hardware implementation of the NN is described.

## 4.1 Theoretical aspects of deep neural networks

Since the breakthrough of Deep Neural Networks(DNN) in modern artificial intelligence, the use of them has grown extensively within a mass of different applications. Their predicting accuracy puts them at the forefront of most AI tasks. However, this comes with a high computational complexity which cannot be further satisfied only with GPUs [33].

As their name suggests, DNN are inspired from how the brain functions and does calculations. The fundamental computational element is the perceptron, whose functional principle resides in the basic idea of a weighted sum. Thus, given a number or inputs, $N$, each having its own corresponding weight, $w_i$, the resultant output will be the scaled sum of the inputs with these weights, on which a non-linear function $\sigma$ is applied:

$$y = \sigma(\sum_{i=1}^{N} w_i x_i). \tag{4.1}$$

Neural networks are arranged in groups of units called layers, usually in a chain structure. That means that the current layer is a function of the layer that preceded it[34]. Fig. 4.1 shows the general architecture of a feedforward-network which is also called a multi-layer perceptron. The goal of these kinds of networks is to approximate a given function. For example, if the task is classification, then the network maps an input $x$ to a category $y$. Besides the input layer and the output layer, there exists hidden layers which define the depth of the network. By and large, networks with more hidden layers than one are capable of learning high-level features with more complexity and abstraction than shallower neural networks[33].

The general equation for a certain layer with multiple neurons $h_{i+1}$ within a DNN is:

$$h_{i+1} = a_i(W_i^T h_i + b_i), \tag{4.2}$$

where $i$ is the layer, $a_i \colon \mathbb{R}^{N_i}$ is the activation function applied for given layer,$W_i \in \mathbb{R}^{M_i \times N_i}$ is the weight matrix, $h_i$ is the preceding layer and $b_i \in \mathbb{R}^{N_i}$ is a bias correction applied to the sum.

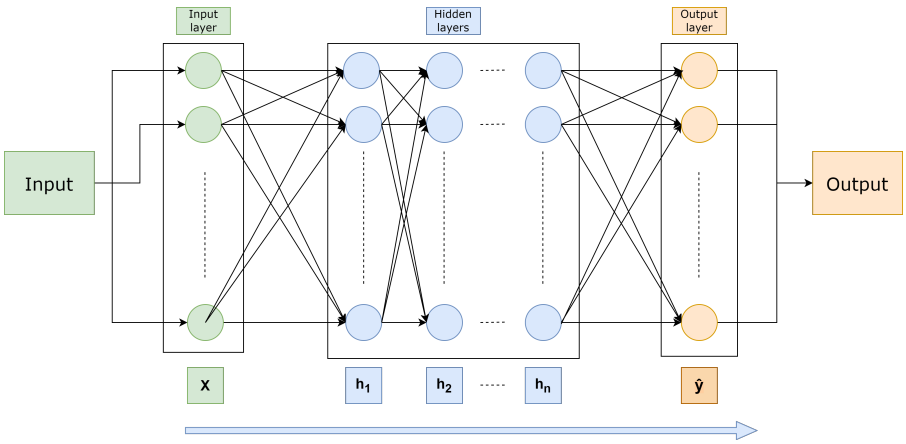



**Figure 4.1:** The general architecture of a feedforward-network.

Since DNNs are comprised within machine learning algorithms, they follow similar learning schemes. Training the network means learning its parameters, which are the weights and biases. The most utilised algorithm for updating these parameters is *gradient descent* with different added optimization techniques. Based on this, the weights and parameters are adjusted in a way such that it minimizes a chosen cost function. The newly updated values can be, therefore, expressed as:

$$w_{i,j}(t+1) = w_{i,j}(t) - \mu \frac{\partial J}{\partial w_{i,j}}, \tag{4.3}$$

where $w_{i,j}(t)$ is the previous value for layer i and neuron j, $\mu$ is the *learning rate* and $\frac{\partial J}{\partial w_{i,j}}$ is the partial derivative of the cost function in regards to the specified weight. The *learning rate* represents one of the hyperparameters which needs to be chosen carefully when designing and training the neural network.

Once trained, the network is able to perform the task of, for example, classification or prediction, with an achieved accuracy, on new inputs. This is referred to as inference. This work is focusing on the latter, since DNN inference is often performed on embedded devices, where resources are limited and performance is critical.

## 4.2 Architecture of the implemented neural network

As stated previously, this work is proposing a hardware implementation of a deep neural network architecture which was first presented in [1]. Since NNs have started to be employed tremendously for user positioning in wireless systems, the need of hardware which can deploy and support these kinds of algorithms had arisen.

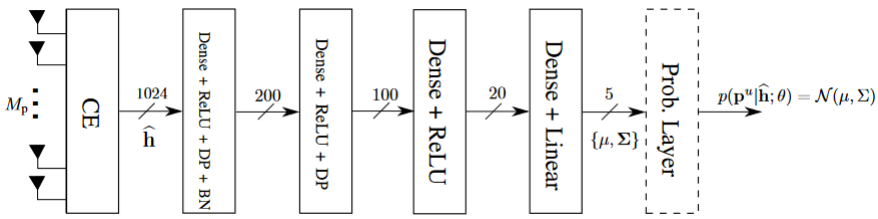



**Figure 4.2:** The architecture of Jesus distributed positioning DNN[1].

The architecture of the feed-forward deep neural network is shown in Fig. 4.2. After the channel has been estimated by the software simulated CE block, the feature vector $\hat{h}$ represents the input to the DNN. Four dense layers with fully connected neurons, modeled by the equation (4.2), are used. During training, different optimization techniques are used such as employing dropout layers(DP) and batch normalization layers(BN).

*Dropout* has proven really effective when training large NN with small datasets. As a regularization method, dropout refers to dropping out units, which is removing neurons from the network, along with all its incoming and outgoing connections[35]. This layer is disregarded during test time.

*Batch normalization* assists with the "whitening" of the network's distribution for each layer, improving the speed of the training as well as acting as a regularization method. Since the change in the distribution of the layers' inputs presents a problem as they continuously adapt, the batch normalization layer aims at correcting this issue. The first step is normalizing each scalar feature independently, where the expectation and variance are computed over the training data set[36]:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}\left[x^{(k)}\right]}{\sqrt{Var\left[x^{(k)}\right]}}, \tag{4.4}$$

and $x$ represents the input vector of dimension $k$. To preserve the proprieties of the normalized layer, an additional step is required:

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}, \tag{4.5}$$

which is essentially scaling and shifting with the parameters $\gamma^{(k)}$ and $\beta^{(k)}$ which are learnt during training. Furthermore, these trained parameters are used during inference.

*RELU* stands for rectified linear unit and acts as the activation for the first three dense layers. It implements the function:

$$f(x) = max(0, x). \tag{4.6}$$

The last layer depicted in the picture is the probabilistic layer which provides the values for the probability density function described by the parameters $\mu, \Sigma$ needed by the loss function that is minimized during training[1]. This is not taken into account during inference, as the last dense layer provides the mean – $\mu$ and the lower triangular matrix – $\Sigma$ from which the covariance can be computed that are required for the computation of the gaussian fusion.

The number of parameters used by each layer and the number of multiplications required in hardware is shown in Table 4.1

| | Layer 1 | BN Layer | Layer 2 | Layer 3 | Layer 4 | Total |
|---|---|---|---|---|---|---|
| Mul. | 204 800 | 200 | 20 000 | 2000 | 100 | 227 100 |
| Param. | 204 800 | 800 | 20 000 | 2000 | 100 | 227 700 |

**Table 4.1:** Parameter breakdown of the neural network shown in Fig. 4.2

.

## 4.3 Optimization techniques for NNs in Hardware

Before designing the architecture of any hardware accelerator it is important to consider the performance requirements, the constraints and what type of system is being designed. Fig. 4.3 shows the design methodology used for this hardware accelerator. First, the structure of the NN was analysed, as presented in section 4.2. Then, different optimization techniques for the relevant performance factors and constraints were studied, as presented in this section. Several of the presented techniques were especially designed for NN implementations on FPGAs, since FPGAs are an inherently suitable platform, due to their flexibility and efficiency.

Lastly all of this information was considered together with the system requirements and a trade off between the different performance factors had to be done to create a final *Hardware Architecture*, as presented in section 4.4.

### 4.3.1 Pipelining the NN

One important technique, enabled by the large size of modern FPGAs, is to implement all layers of the DNN on the FPGA in a pipeline structure. This enables one to adopt different optimization strategies for each layer, as they have their own dedicated logic. As the layers of a DNN can be widely different this allows for better optimizations which in turn maximise throughput, and decrease the frequency of memory access[17].

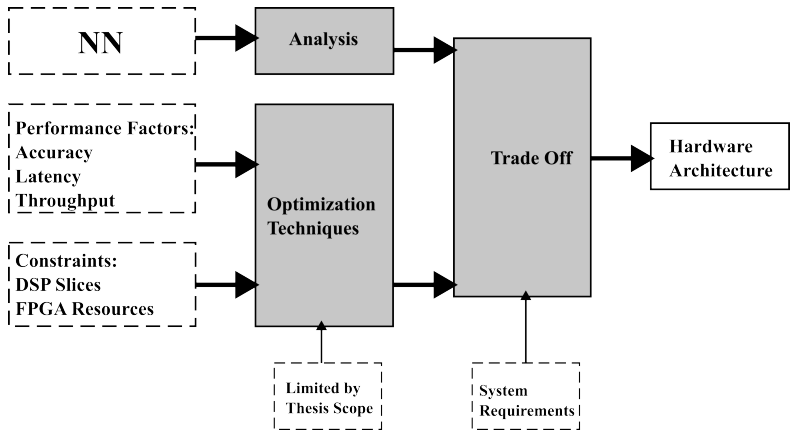



**Figure 4.3:** Design considerations for the NN hardware accelerator.

### 4.3.2 Distribution of the DSP slices

When creating the pipeline structure, a major consideration is how to best utilize the DSP slices. To maximise the throughput of a pipeline, the execution time of each pipeline stage should be around the same. The DSP slices should also be distributed in a way such that they are utilized every clock cycle and do not lie dormant[17].

Equation (4.7) and (4.8) can be used together to determine how to best distribute the DSP slices through the layers of the DNN. $N^i_{in}$ and $N^i_{out}$ denotes the amount of input, respective output neurons of the i-th layer and $CCY$ denotes the amount of clock cycles it takes to compute a layer. As $CCY$ should be the same for every layer to best utilize the pipeline its value is not tied to the actual layer. $Para_{in}$ and $Para_{out}$ denote the amount of neurons inputted from the previous layer every CCY and the amount of result neurons the results are sent towards.

$$Para^i_{in} \times Para^i_{out} = \frac{1}{CCY} \times N^i_{in} \times N^i_{out} \tag{4.7}$$

The target is to determine $Para_{in}$ and $Para_{out}$ for every layer so that the following requirement are met. $Para^i_{in}$ and $Para^i_{out}$ should be divisors of $Nin^i$ and $N^iout$, this guarantees that all DSP slices are used every clock cycle. The amount of multiplication operations done per clock cycle also needs to be smaller than the amount of available DSP slices, as shown by equation (4.8).

$$\sum_{i=1}^{N_{layer}} Para^i_{in} \times Para^i_{out} \leq DSP_{total} \tag{4.8}$$

This NN was designed with the ZCU102 board in mind, which is described in detail in section 2.2.2. One board needs to be able to fit two NNs and the Gaussian Fusion. In total, it has 2520 DSP Slices available, but one NN can use at most 1200 slices, and preferably less to increase the ones available for the fusion.

In Table 4.2 an analysis over how many DSP slices will be needed per layer depending on the CCY, calculated using equation (4.7), is shown. The result of $Para^i_{in} * Para^i_{out}$ shows how many multiplications are executed in parallel for the chosen layer. How to distribute them between $Para_{in}$ and $Para_{out}$ can be decided later depending on what is most suitable.

| | $Para^i_{in} \times Para^i_{out}$ | | | | |
|---|---|---|---|---|---|
| **CCY** | **i=1** | **i=2** | **i=3** | **i=4** | Sum |
| 100 | 2048 | 200 | 20 | 1 | 2269 |
| 160 | 1280 | 125 | 12.5 | 0.625 | 1429 |
| 200 | 1024 | 100 | 10 | 0.5 | 1135 |
| 320 | 640 | 62.5 | 5.25 | 0.3125 | 710 |
| 400 | 512 | 50 | 5 | 0.25 | 568 |

**Table 4.2:** The amount of DSP slices need per layer depending on the amount of clock cycles the execution per layer takes.

### 4.3.3 Optimizations for Fully Connected Layers

After determining the overarching structure of the system it is important to look at how to optimize every individual layer. When optimizing the individual layers the technique that is most suitable depends on what type of layer it is used on. For Dense layers, also called Fully Connected layers, the largest bottleneck is the large memory bandwidth needed to access the weights during computation. One way to solve this is to use a batch-based computing method, which trades higher latency and more on chip memory for a reduced required memory bandwidth[17]. As implemented network is fairly small it is feasible to keep all parameters in the on-chip memory, which removes the need for this optimization.

### 4.3.4 Number Representation

Another major consideration is the manner in which the numbers are represented. This impacts both the amount of memory needed, and the amount of resources needed for computation. Multiple works have shown that NNs are inherently noise tolerant and that low precision quantization can be used without sacrificing high amounts of accuracy.

To determine which number representation to use in the implementation a behavioural model was created where the accuracy of different options was tested. The result of this is displayed in Table 4.3. To achieve higher accuracy for the lower resolutions it is desirable to fine tune the network. That is an added process at the end of the training where the parameters are adopted to the picked quantization, but that was outside the scope of this thesis as training the network is not included in the work process.

When doing multiplication in two complement fixed-point, it is important to decide how to round the numbers. When two 8-bit numbers are multiplied it is

| Number Representation | average error(cm) | max error(cm) |
|---|---|---|
| 32 bit Floating Point | 2.30 | 9.04 |
| 32 bit Fixed Point - full precision | 2.25 | 8.04 |
| 16 bit Fixed Point - full precision | 2.26 | 7.93 |
| 8 bit Fixed Point - full precision | 3.39 | 12.30 |
| 8 bit Fixed Point - INT8 | NaN | NaN |
| 8 bit Fixed Point layer 1, 16 bit for the rest - full precision | 2.26 | 7.93 |
| 8 bit Fixed Point layer 1, 16 bit for the rest INT8 for layer 1 | 2.26 | 7.93 |

**Table 4.3:** The accuracy for different quantization for the NN.

possible for the result to overflow, and in that case the result will wrap around and become negative. To keep full precision one can for example double the bits just before multiplication therefore increasing the range to all the possible results. If nothing else is specified, *Vivado* executes the multiplication in full precision.

The INT8 notation refers to a number representation where both the inputs and outputs of a multiplication are 8 bits, which means that full precision is not guaranteed as overflow is possible. If this representation is picked, it is possible to utilise one DSP slice for two multiplications, which doubles the amount of parallel computations without needing more resources[21].

The ability to utilize a DSP slices for two multiplications is especially suitable to NNs. This is due to the requirement that one of the inputs needs to be the same for both multiplications, a scenario that's common for NNs. For the second input vector, which needs to contain two different values, the 27 bit input to the DSP slice is utilized. It's possible to pack the first value in the least significant eight bits and then leave 8 empty bits before packing in the second value. This empty space means that the results of the multiplications can be taken directly from the output vector as the result bits from the first calculation does not impact the second[21].

### 4.3.5 Latency Optimization

The previous optimization techniques are focusing on increasing throughput, decreasing the needed memory or the resources used, as these are usually the most relevant factors. This network though is designed to run in real-time which makes latency a relevant factor to optimize for as well.

The pipeline structure discussed in the beginning would divide every individual layer into its own stage, which for this network would create five pipeline stages. But by utilizing the inherent structure of a fully connected layer it is possible to combine two layers into one pipeline stage, thus decreasing the latency. In a fully connected layer all output nodes from the previous layer are used to calculate the results of the current layer. If the first layer was to focus on computing its

first output node as rapidly as possible, the second layer can start by calculating the values related to that node. This reduces latency dramatically, by allowing the second layer to start computations much earlier. It also does not affect the throughput as long as the time to calculate an output node of the first layer matches the time it takes to calculate all operations regarding that node for the second layer.

## 4.4 Implementation

The performance requirements for the NN are the following: Performance wise the throughput needs to be 32 000 inference calculated per second and the latency should be minimized as much as possible. The resource utilization to archive this performance should also be minimized if possible, as there needs to be resources left for the other parts of a Massive MIMO system. This should be achieved without degrading the accuracy of the system too much. The final consideration is power usage, and for this implementation, that is not a prioritized parameter. It will only be taken into consideration when it does not affect the other factors.

To fulfill these requirements, it is important to use the optimization techniques presented in the previous section. To minimize the resources usage for both memory and computational resources, it is desirable to pick the smallest number representation that doesn't sacrifice too much accuracy. It is most important to optimize the first layer, as it does 90% of the operations. Therefore, it was decided to pick the representation depicted in the last row of Table 4.3, with INT8 representation for layer 1 and then 16-bit full precision for the rest. The accuracy is reduced somewhat, but it allows for the optimization technique where one DSP slice can be utilized for two operations for the first layer.

The next step is to decide how to distribute the DSP slices through the layers. Table 4.2 shows the amount of DSP slices utilized in different layers, depending on how many clock cycles it takes to compute on layer. As only half of the DSP slices will be used for the layer 1, the numbers on the second column of the table do not apply anymore. Keeping all the numbers presented in Table 4.2 will result in a low-enough throughput which is way better than the requirement, so the trade-off becomes trading resource usage for low latency. Finally, $CCY = 200$, which needs 623 DSP slices and gives a throughput of 500 000 inference per second when clocked at 100MHz.

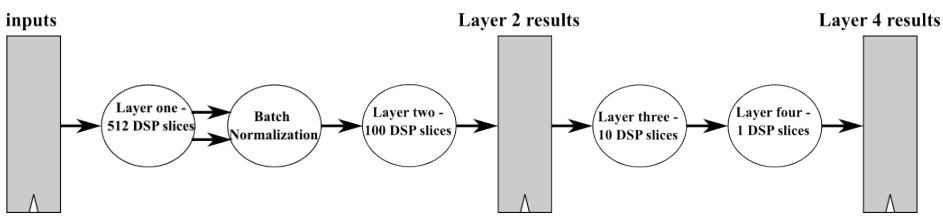



**Figure 4.4:** An overview of the data path of the Neural Network.

Fig. 4.4 shows an overview of the data path of the Neural Network. Two fully connected layers are merged into one big pipeline stage to reduce latency as

described in the previous section. This gives us two big pipeline stages in the NN, one that contains layer one, the batch normalization calculations and layer two, and one that contains layer three and layer four. The next step now is how exactly these separate modules are designed to fit the design requirements.

Previously, the result for $Para^i_{in} * Para^i_{out}$ has been shown for every layer, but not their individual values. This decision has a big impact on the implementation, and to fully illustrate it a combination of figures and equations will be used.

The general equation for a fully connected layer with multiple neurons is shown in equation (4.2) and the equation for a single neuron, without the bias, is shown in equation (4.1). In Fig. 4.5 the calculations for one layer, excluding the bias and activation function, is shown as the matrix multiplication it is. The input matrix has dimension $1 * n$ and the weights matrix has dimension $n * m$. Where $n$ is the amount of input nodes, same as $Nin^i$, and $m$ the amount of output nodes, same as $Nout^i$.

When deciding $Para^i_{in}$ and $Para^i_{out}$, the decision is in which order all the matrix operations are done in. One can choose the traditional approach of matrix multiplication which is taking the row of the input and multiplying it with the columns of the matrix. If we consider, for example, 4 DSP slices, then they will be mapped to the first 4 column values in the matrix, which will be further traversed vertically. Following this approach will yield a partial product of a one output node. A different perspective is traversing the matrix horizontally, in batches, illustrated with the grey box in Fig. 4.5. Considering the same number of DSP slices, two inputs are processed every clock cycle to calculate intermediate values towards two separate output nodes. In this case, $Para_{in} = 2$ and $Para_{out} = 2$ and two output nodes would be fully calculated after $n/2$ clock cycles.

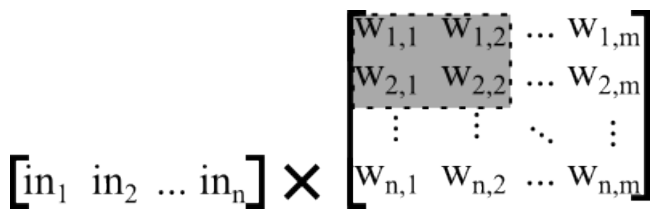



**Figure 4.5:** General matrix multiplication

The final values used when designing are displayed in Table 4.4. Two different strategies have been employed for the layers at the beginning of the pipeline, and at the end of the pipeline, respectively. Layer one and layer three are focusing on computing one output node at the time, so layer two and four can use the results in their calculations. Therefore the matrix should be traversed column-wise. $Para^i_{out}$ is 2 for layer one, since the DSP slices in this layer are configured to run two separate operations, which is only possible if one parameter is the same. This happens when the NN is calculating values for two output nodes, as they have the same inputs, but different weights. In layer three and four the opposite structure is needed. As only one input is available when the calculations start, the matrix has to be traversed row wise instead.

There are different implementation challenges for traversing the matrix row-

| Layer | $Para_{in}^{i}$ | $Para_{out}^{i}$ |
|---|---|---|
| 1 | 512 | 2 |
| 2 | 1 | 100 |
| 3 | 10 | 1 |
| 4 | 1 | 1 |

**Table 4.4:** The values of $Para_{in}^{i}$ and $Para_{out}^{i}$ for every fully connected layer.

wise or column-wise. When traversing it column wise, memories for all partially calculated operations are needed. In this case though the DSP slices accumulate register can be utilized for that. Thus, no extra resources have to be used for this memory. When traversing it row-wise, the amount of additions done to sum up the values from the multiplication are numerous. To decrease the length of the critical path an addition tree is implemented as shown in Fig. 4.6.

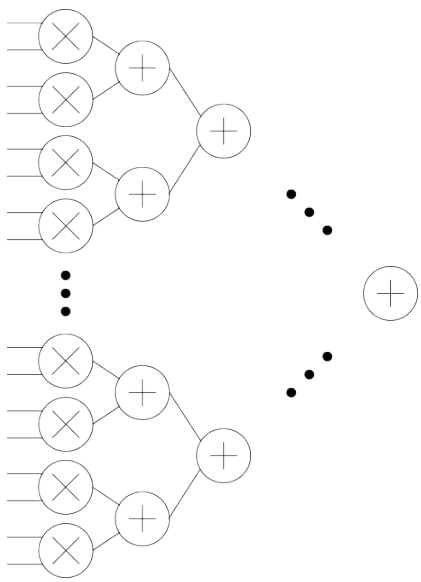


**Figure 4.6:** The data path of layer one and layer three.

The last layer to implement is the batch normalization layer. The equations for batch normalization are equation (4.4) and (4.5). The implementation needs to have a throughput of one input calculated every 200 clock cycles. As there are 200 input values per input as shown in Fig. 4.2, the batch normalization needs to output a new value every clock cycle. This is a very high throughput as the batch normalization is made up of notoriously slow arithmetic operation, division.

To achieve the throughput one simplification is first done. As $Var[x^{(k)}]$ is a constant, the result of the square root is also a constant. Therefore this value is

pre-calculated removing the need for implementing square root in the hardware implementation.

The division is implemented using the Non-Restoring Algorithm, as described in section 2.3.1. It's an iterative algorithm that is performed over multiple clock cycles. One bit in the result is calculated per iteration, thus the execution time in clock cycles is the same as the amount of bits in the dividend. To increase throughput a combination of pipelining and loop unrolling is utilised. The division can be loop unrolled slightly, as more than one iteration can be executed per clock cycle without timing violations. The limit was concluded to be three iterations per clock cycle. To further improve the throughput the division is fully pipelined, and therefore able to accept a new input every clock cycle.

The division is not the only module that needed to be further pipelined to meet the requirements. A degree of pipelining was necessary for the DSP slices and some of the longer critical paths inside the adder trees to achieve good throughput. This creates a structure of a "big" pipeline, which is shown in Fig. 4.4, and smaller pipeline stages inside of these to allow the design to meet its timing requirements. To prevent a decrease in throughput, the first stage in the big pipeline accepts new inputs as soon as the first "small" pipeline stage is empty instead of waiting for all the calculations inside of the stage be done.

# Chapter 5
# Gaussian Probability Fusion

The goal of the implemented system is to provide an accurate estimation of a user location from the probability densities provided by the four LIS panels. All these outputs are fused together by a central processing unit to attain more precise position coordinates. The design and the theory behind the fusion of the four outputs from the panels will be presented in the current chapter.

## 5.1 Introduction

The fusion of the results of the panels is done by combining a finite number of probability densities into a single one through a method named *conflation*[1]. The user position is modeled as a multivariate normal distribution:

$$p_i = \mathcal{N}(\mu_i, \Sigma_i), \tag{5.1}$$

for the $i$-th panel, and $\mu_i \in \mathbb{R}^2$ is the mean, $\Sigma_i \in \mathbb{R}^{2\times 2}$ is the covariance. Consequently, each panel estimation is represented by the tuple $\{\mu_i, \Sigma_i\}$ which will further be the input of the fusion module.

## 5.2 Theoretical aspects of probability fusions

Conflation is essentially consolidating the information from several independent experiments, all measuring the same unknown quantity[37]. This technique preserves the type of the individual distributions, thus the resulting distribution having the same proprieties as the individual ones. For this particular case, since the distributions outputted by the NNs are gaussian, the combined distribution will also be gaussian. The covariance and mean are represented as[1]:

$$\Sigma_f = \left( \sum_{i=1}^{P} \Sigma_i^{-1} \right)^{-1}, \tag{5.2}$$

and, respectively,

$$\mu_f = \Sigma_f \left( \sum_{i=1}^{P} \Sigma_i^{-1} \mu_i \right). \tag{5.3}$$

In the software model, all computations for both the NN and the fusion calculations are done using python. To be able to represent and use these entities in python while training the neural network, the Tensorflow library offers this possibility through the *distributions* class[38]. In [1], a distribution – *MultiVariateNormalTriL*[39] – which is entirely characterized by the mean of the random variables and a lower triangular matrix denoted as *scale*, is used. These variables are obtained after the last dense layer pictured in Fig. 4.2. The class provides the possibility of retrieving different parameters such as samples, the standard deviation or the covariance. However, the way the computations when calling these parameters are done has to be fully known in order to be able to completely characterize the hardware describing it. Thus, a behavioral model written in matlab, emulating the hardware has been used as a bridge between the hardware description in VHDL and the software implementation in python.

## 5.3 Probability fusion implemented in hardware

Given that the NN python model containing the multivariate gaussian distribution does not output the covariance directly, the resulting values have to be preprocessed since they are needed for the fusion according to equations (5.2) and (5.3). The five resulting values from 4.2 are the mean values for the two coordinates, and the lower triangular matrix values *scale*, such that:

$$\text{covariance} = \text{scale} * \text{scale}^T, \tag{5.4}$$

where * denotes matrix multiplication and $^T$ is the transpose operator.

Before computing the covariance accordingly, preprocessing of the resultant diagonal values of the *scale* matrix, within the *MultiVariateNormalTriL* python class is conducted in conformity with the source code. First, a *softplus* transformation is applied and then a small number is added to the resulting diagonal values. Thus, the diagonal numbers in the *scale* matrix become:

$$\text{scale\_diag\_new} = \log(1 + \exp(\text{scale\_diag})) + 1^{-5}. \tag{5.5}$$

To be able to describe any given function in hardware, one must think about the elementary building blocks : adder, multiplier and shifter. In the following sections the manner in which these blocks have been implemented with the help of VHDL and further used will be presented.

### 5.3.1 Floating-point arithmetic

In order to capture the whole dynamic range of the numbers, the fusion operation is implemented in floating-point arithmetic. The number distribution for the covariance values is shown in Fig. 5.1 for all the panels, each having 100 inputs. While the general curve shape is preserved, the dynamic range changes as much as from $10^{-5}$ to $10^{3}$. To represent this full range accurately, 38 bits would be required. This would mean more than doubling the number of bits resulted from the

neural network, so a compromise between precision and accuracy has to be done. If 16-bit or 32-bit fixed-point would be used, the operations which require rounding would cause a bigger loss in accuracy in the long run. This makes floating-point the preferable choice that can cater to the maintaining of the accuracy.

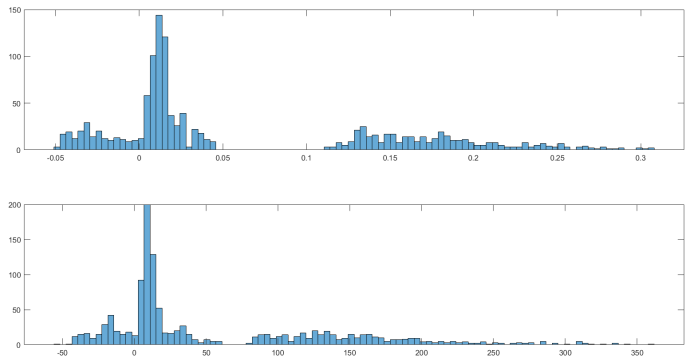



**Figure 5.1:** Histogram of the covariance values before the matrix inversion(top) and after the matrix inversion(bottom)

## Fixed-point to floating-point converter

Usually, a number represented in floating-point is stored in hardware like in Fig. 5.2. Given the fact that the result from the NN is in fixed-point arithmetic, a **to_float** converter has been implemented which takes the signed 16-bit number and splits it into the fields as shown in Fig. 5.2, by converting the signed value into sign, signed exponent, and unsigned mantissa. For simplicity, no rounding scheme was used, and instead the numbers are truncated during different computing stages.


**Figure 5.2:** Bit fields in the floating-point representation

### Normalization

The normalization of the numbers is implemented by shifting the number of leading zeroes out of the mantissa[40]. The leading zeroes are counted by checking each bit starting from the most significant one. Then the number of leading zeroes are added to the exponent.

### Floating-point adder

The floating-point addition requires more steps then integer addition and thus, more clock cycles. The hardware implementation follows the control flow depicted in Fig. 5.3 and is based on the algorithm shown in [28]. The sign is computed based on the larger number. The whole process starts once both operands are available. When the addition is done and the result stored in the register, the *manager_valid* is raised. The reason of connecting a register directly to the addition's result is to utilize the capability of the DSP slices.

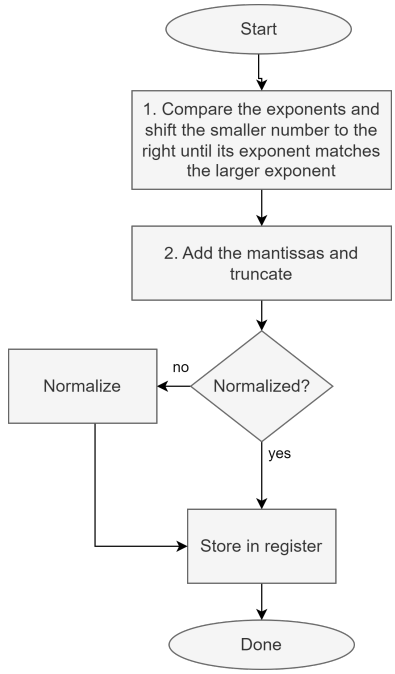



**Figure 5.3:** Floating-point addition

### Floating-point multiplier

The floating point multiplier is simpler than the adder, due to the fact that denormalization is not necessary before multiplying the mantissas. The resulting sign is the *XOR* operation between the inputs' signs and the exponent is calculated by summing the operands' exponents. After the multiplication operation, the result is normalized.

## 5.3.2 Exponential and logarithm

The exponential and logarithm functions are implemented following alike strategies, namely approximating them with the aid of the Taylor series presented in chapter 2. Since the basic principle of the expansion is achieving accurate values around a given expansion point, this has to be chosen according to the expecting range of the input values. Thus, for the exponential, the expansion point was chosen to be -1.5 and by rearranging the terms in the equation (2.5), we obtain:

$$e^x = e^{-1.5}\left[1 + q\left(1 + q\left(\frac{1}{2} + q\left(\frac{1}{6} + \frac{q}{24} + q\left(\frac{1}{120} + \frac{q}{720}\right)\right)\right)\right)\right], \qquad (5.6)$$

where $q = x + 1.5$. This is a well-known method to reduce the number of multiplications in Taylor series[41].

Given that the inherent operation that needs to be performed is $\ln(1 + x)$ according to equation (5.5), then the Taylor series with the chosen expansion point being 0.625, becomes:

$$\ln(1+x) = 0.1009 + \frac{8x}{13} + \frac{32t^2}{169}\left[-1 + \frac{16t}{13}\left(\frac{1}{3} + \frac{2t}{13}\left(-1 + \frac{32t}{13}\left(\frac{1}{5} - \frac{4t}{39}\right)\right)\right)\right] \tag{5.7}$$

where $t = x - \frac{5}{8}$. Since most of the divisions here are done with the divisor being a constant, it is only practical and resource efficient to perform multiplication with the pre-calculated inverse value rather than division. As it can be seen from the equations, the non-linear functions that needed to be implemented are becoming a series of floating-point addition and floating-point multiplications.

The expansion points have been chosen by estimating the range of the numbers at the prefusion stage with the help of the behavioral model. The schematic for both the exponential and the logarithm function can be seen in Fig. 5.6. Only the adders and the multipliers are depicted to simplify the diagram, but registers are also present in certain places so that the timing requirements are met. Moreover, the result is normalized before being sent on to the next module.

### 5.3.3 Matrix inversion

Once the covariances are calculated from the scale matrices, they have to be inverted according to (5.2) then added together and inverted again. Given the fact that $\Sigma_i \in \mathbb{R}^{2\times 2}$, the matrix inversion can be achieved by only calculating the inverse of the determinant and multiplying it with the adjoint matrix. Let

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

be a full-rank $2 \times 2$ matrix. Then $\det A \equiv |A| = a_{11}a_{22} - a_{12}a_{21} \neq 0$ and

$$A^{-1} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}^{-1} = \frac{1}{|A|}\begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix}.$$

The determinant is easily calculated with two multipliers and an adder. For calculating the inverse of the determinant the Newton-Raphson method, explained in chapter 2, was chosen. To perform floating-point division, it is only necessary to calculate the inverse of the divisor's mantissa, multiply it with the dividend's mantissa, and then subtract the exponents. The sign is computed as it is for multiplication, using a *XOR* gate. Initially, the division function implemented in the batch normalization layer for the NN was chosen for this step as well. Since the mantissa contains 23 bits, 46 iterations would have been necessary to achieve the correct result of the division. This would have impacted the latency and throughput significantly. Furthermore, with the fast division method, the inverse

of the determinant has to be calculated once and within the span of notably less clock cycles, by just iterating through the algorithm three or four times and achieving similar accuracy.

The limitation of the chosen algorithm is that it is very sensitive to the initial estimation[42]. In hardware this is done with the help of a look-up Table, as it has been done in [43]. To achieve higher accuracy, one can run more iterations. When following equation 2.3, one must be careful about the how the intermediate results' scaling behaves. Since the floating-point numbers are always used in their normalized form one can assume the following about the divisor and the initial inverse approximation:

$$1 \leq \text{divisor} < 2, \mathbf{Q1.22}$$

$$0.5 \leq \text{inverse} < 1, \mathbf{Q23}$$

If we regard the numbers as being represented in fixed-point and we keep the most significant bits of the multiplication result, then:

$$0.5 \leq \text{divisor} \cdot \text{inverse} < 2, \mathbf{Q1.22}$$

To represent the number 2, 2 bits are required for the integer part. Thus, we shift the multiplication result to match the number of 2's integer bits and we get:

$$0.25 \leq \text{divisor} \cdot \text{inverse} < 1, \mathbf{Q2.21}$$

$$1 \leq 2 - \text{divisor} \cdot \text{inverse} < 1.75, \mathbf{Q2.21}$$

The subtraction result is smaller than 2, so the first bit can be shifted out, yielding:

$$0.5 \leq 2 - \text{divisor} \cdot \text{inverse} < 0.875, \mathbf{Q1.22}$$

$$0.25 \leq \text{inverse} \cdot (2 - \text{divisor} \cdot \text{inverse}) < 0.875, \mathbf{Q1.22}$$

Since both the operands are smaller than 1, then the first bit can be discarded, leaving us with the generated **Q23** inverse which is going to be the input to the next iteration. At the end of the algorithm the result is normalized and the *manager_valid* is asserted.

### 5.3.4 Prefusion block

This block implements the equation (5.5) followed by the matrix multiplication in (5.4) and finally, it inverts the calculated covariance matrix as explained in the previous section, preparing it for the computations in (5.3) and (5.2). Because the matrix contains the same values on the secondary diagonal, only three values are needed to be able to fully represent it. To match the throughput of the NN, all the computations had to be pipelined. One pipeline stage lasts 40 clock cycles, as it matches the slowest module's speed, which is the logarithm . The data changes based on a input *subordinate_ valid* signal which is also going through the pipeline stages. Although the values of the means do not get changed within this block, they are still pipelined to ensure they correspond with the values of the covariance matrix. The data path is illustrated in 5.4

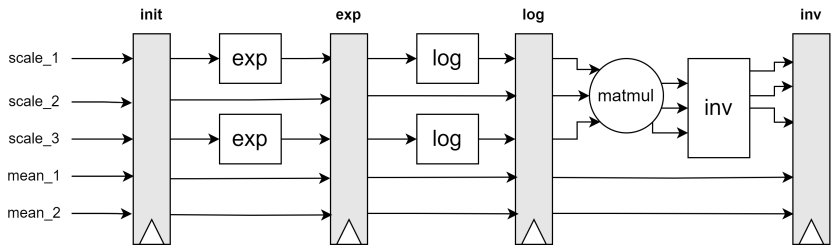



**Figure 5.4:** The prefusion module with the pipeline stages

### 5.3.5 Fusion block

The fusion itself does not contain any complicated functions or operations, apart from calculating the inverse of the resultant covariance needed to compute the fused mean. A simplified version of this block is presented in Fig. 5.5, with only the pipeline registers being shown. Since the arithmetic takes less than the required clock cycles, the adders are being reused for each value of the covariance matrix, and the mean matrix, respectively. This gives the opportunity to save resources on the FPGA as well. For example, the top adder will take 12 clock cycles to add all the three covariance values. A select logic based on two counters is employed to choose which of these three values are being added at a specific moment. Moreover, it decides if the addition will take place for fusing the first two panels, or for the third panel and the fused first two panels, and so on. Naturally, more registers are used to store the intermediate values computed during different stages. This module is easily scalable to be able to fuse extra panels maintaining the same hardware resources, at the expense of latency.

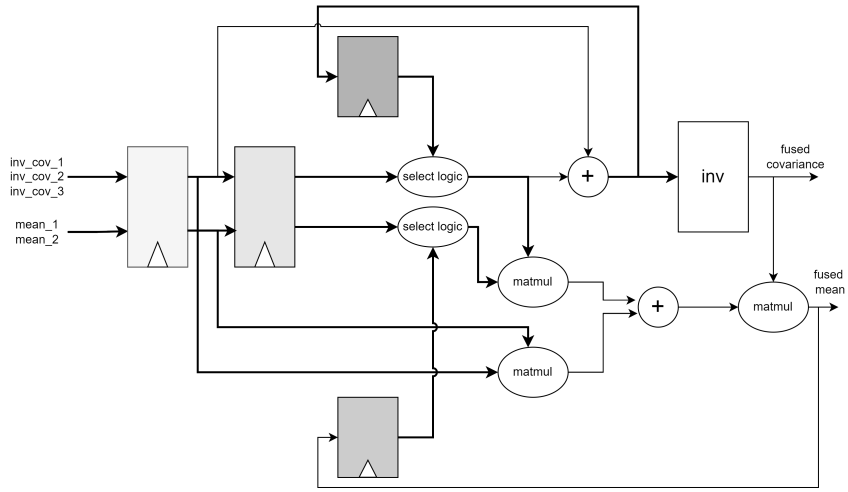
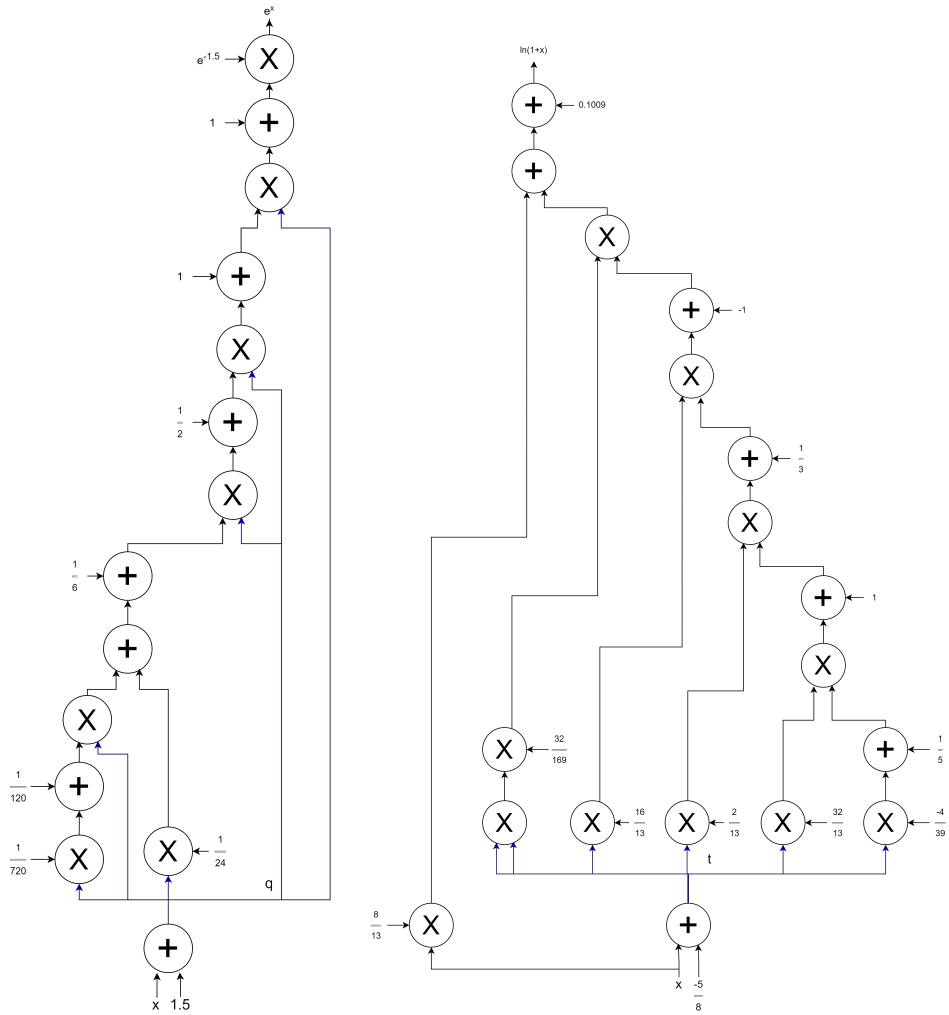



**Figure 5.5:** The fusion module. For the simplicity of the figure, the three individual covariance values are regarded as one signal, depicted with a thicker line. The same applies for the values of the means.




**(a)** $e^x$ **(b)** $\ln(1+x)$

**Figure 5.6:** Taylor series implementation in hardware

# Chapter 6
# Communication System Using Ethernet

In the current chapter, a brief description of the Ethernet is introduced, alongside with the presentation of the implemented module within the system.

## 6.1 Introduction

As the usage of computers continued to grow, people started to realize the need of a common standard of how computers talked to each other regardless of manufacturer or operator. The ISO therefore started to develop a standardized architecture called OSI. Modern communications do not follow this standard and it is instead used as a reference model for how a communication network ought to be built[44].

The OSI layer is divided into seven different layers. Every layer can contain one or more different protocols, and they should work independently of each other. This means that if the protocol on the lowest layer, the physical, was switched due to changing transmission medium, this should not affect any other layer[44].

The Ethernet protocol resides in the physical and data link layer, therefore defining the lowest abstraction levels of a communication link. The Ethernet standard contains several different specifications differing by both data rate and different interconnect medium interfaces[45]. These are all defined in the IEEE 802.3 standard.

## 6.2 Original plan and issues

The aim of the communication system is to use an Ethernet port to provide a communication channel between two FPGA boards. Depending on which board is in use, it is possible to implement this in several different ways.

The ZCU102 board has two sets of Ethernet Ports, it has an 4xSFP+ cage connected directly to the PL and a RJ45 Ethernet Port coupled to the PS. The RJ45 port can support a data rate up to 1Gb/s while the 4x SFP+ cage contains four ports that can each support 10Gb/s[6]. The RJ45 also has to take an inefficient detour through the PS system of both FPGAs causing both increased latency and being wasteful in regards to resources utilisation. The only benefit of the RJ45 is that it allows the full usage of the PS and therefore lowers development time as the Ethernet can be controlled through programming instead of hardware.

Originally, the approach was to use the SFP+ cage and to control the Ethernet transmission entirely over hardware. This decision was made for the following reasons: It gave the best performance both in regards to throughput and latency, which is desirable. This is not only due to the fact that the link itself can support a higher bandwidth, it's also because the link would be implemented on lower level removing the unneeded procedures that exist on the higher levels created for general Internet communication.

It's also the most suitable approach to work with an actual massive MIMO system, as the bandwidth would then have to support both the positioning system and the information for the MIMO processing. Lastly it also leaves the processing system unoccupied, which means it can be utilized for something else like a real time graphical display of the ongoing transfer.

To implement this system, the Xilinx IP for controlling the Ethernet would be utilized and a control system for it would be created in the FPGA logic. A Xilinx example design existed for this IP when it was configured in a loop back mode. This one was to be used as a basis for the control block, but when a bitstream was to be generated to test its function on the FPGA that did not work. This is when it was first noticed that the available license only covered simulation and not actual bitstream generation. This was discovered at a point when it was too late to get the actual license, which meant that the communication system would need to be implemented in a different way.

At this point the time left to implement the communication system had decreased dramatically, which meant it was necessary to pick an option that would be time efficient. It would be necessary to implement it through the PS and both available ways to do that with the ZCU102 board, through vitis or generating a PYNQ image through petalinux, was time consuming. So instead it was decided to switch boards to the RFSoC 2x2 that already had an available PYNQ image which would cut down implementation time dramatically.

## 6.3 PYNQ

PYNQ is a fairly new open source framework created by Xilinx, and it has been specifically designed to control their SoC FPGAs through python code written in Jupyter notebook. To work with PYNQ, certain requirements must be met. First, the FPGA needs to be a SoC and thus contain a processor. Second, a slot for a SD card is necessary as the operating system for PYNQ is stored in an SD card image. The image contains a Linux variant with all the necessary files for running PYNQ. For some boards, the RFSoC 2x2 among them, there exist pre-made PYNQ SD card images that can be downloaded and put directly on the board, while for others the image has to be made manually through their petalinux toolset[46].

The functionality of PYNQ is easily visualised through looking at its hierarchy levels, which are shown below in Fig. 6.1[5]. The lowest level contains the actual hardware designs. The desired IPs need to be put together with the processing system in a block diagram, called an *overlay*. The next two hierarchy levels are made up of the Linux kernel and the python libraries that make up PYNQ. These bind the software written by the designer to the hardware. If the standard li-

braries are not enough for the designer it is possible to add new python files to the libraries. One major example is creating a device driver for a complicated custom IP. The highest hierarchy level is the PYNQ notebooks that are written in Jupyter notebooks, an interactive web based environment. The computer connects to a server on the FPGA where all the files are saved and the code is run thus being only a visual interface.

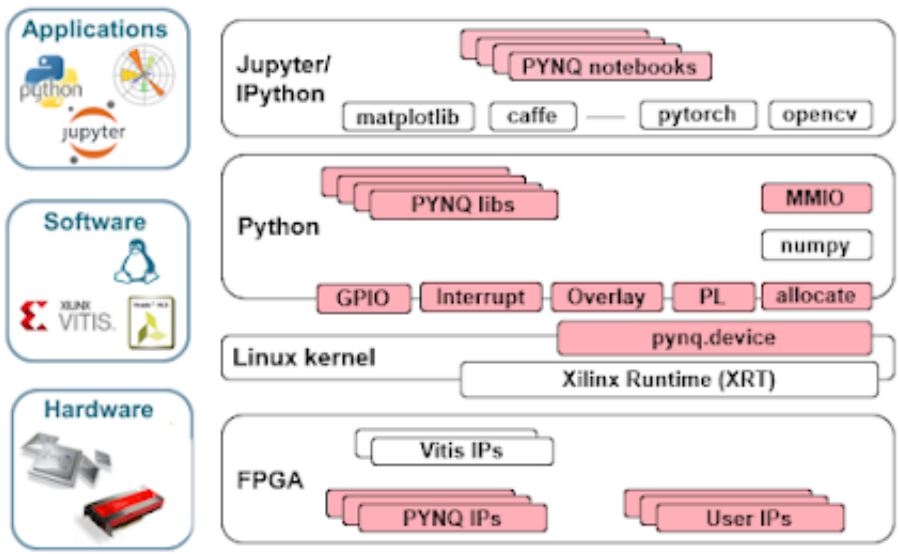



**Figure 6.1:** The hierarchies of the PYNQ framework[5].

PYNQ has been designed with several different purposes, some of them being helping hardware designers reach a wider possible audience, as their finished designs can be controlled and customized purely through software, creating an easy software interface that can help with rapid prototyping and development[5], and allowing for run time configurations of hardware.

## 6.4 Implementation

The implementation of the communication system is different from the previous modules. Since as it goes through the processing system the implementation is done in both hardware and software. An overview of the communication path is shown in Fig. 6.2.

### 6.4.1 Direct memory access – DMA

There are several ways to move data from the PS to the PL. For a high performance communication link, a DMA is the most suitable. A DMA is a system that allows hardware modules to access the memory of a processing system independently of the CPU. The DMA used in this project is a finished Xilinx IP whose specifications

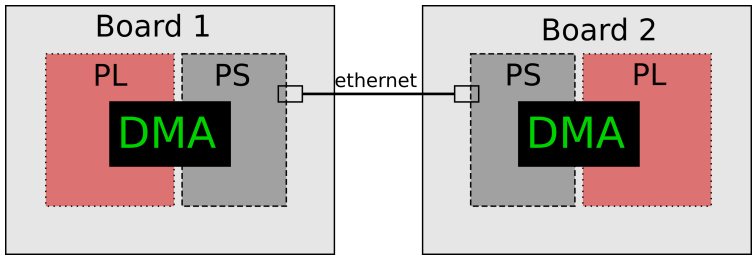



**Figure 6.2:** The communication path between two FPGA boards.

can be found in [47]. Its functionality is that it takes data from an AXI Stream and moves it into the system memory through the AXI4 memory mapped interface. In this project the system memory is the memory of the PS. The DMA has two different channels, *Memory Mapped to stream (MM2S)*, from PS to PL, and *Stream to Memory Mapped (S2MM)*, from PL to PS[47].

The DMA is controlled through several control and status registers. In this project, these registers are written to from the Jupyter notebook through the PYNQ libraries. There are a lot of different features that can be enabled or disabled for this IP, but most of these cannot be used as the PYNQ library is not yet compatible with them. The DMA is therefore implemented in its simplest mode, the Direct Register Mode[48]. In this mode the data transfer does not happen independently of the CPU. This lowers the performance both by increasing set up time and by not allowing the CPU to be utilised by other operations during parts of the transfer.

Since the DMA is a very important IP when working with both the PS and the PL, PYNQ has classes to control it. A class called *allocate*, is used to manage the memory buffers inside of the PS that the DMA interacts with. The DMA demands that the memory buffer in the PS memory has a continuous physical address, which the buffers created through the *allocate* class always have.

The need for allocate buffers for the DMA transfer made the development of the python software somewhat complicated, as the code used for the Ethernet transfer could not work with an allocate buffer. Therefore, it was necessary to decode and encode the data into a bytes object when interfacing with the Ethernet transfer. The encode method loops over every individual sample to translate it from an integer in an allocate array into a bytes object, while the decode method loops over every individual sample to translate it from a bytes object into an integer entered in a pre-created allocate array.

### 6.4.2 Ethernet

To create the Ethernet transfer, a simple set up with the python socket class[49] was used. Socket programming is usually used to connect two nodes which need to communicate with each other. A server-client application on the CPUs of the FPGAs has been established as it was done in [50]. The simple principle behind it is that one board acts as the server, listening to any available ports. The second board acts as the client, trying to form a connection with the listening

port. The client could then continuously send data that would be collected by the server. One of the pitfalls that arised when using this approach was that the send and recv methods within the class were not handling all the bytes which were transmitted[49]. Since the unit of the sent data is made out of 4 bytes, the integrity of the numbers on the *receive* side is of the utmost importance in order to decode them correctly. Since the ethernet directly talks to the DMA on the both sides, an approach of buffering the data on the *receiving* side before sending it further to the programmable logic had to be employed in software.

Another characteristic of the socket class is that it will not stop transmission by itself, when it is not receiving data anymore[49]. This makes it suitable for real-time systems as the time between data transfers might not be always constant. On the other hand, the recv method might wait forever, if the connection is not broken in some way.

The disadvantage of this type of set-up is that both of the ARM cores are occupied and therefore no other action can be performed on them.

# Chapter 7
# Results

This chapter is presenting different performance measurements during different stages of the project. The hardware accelerators and the Ethernet subsystem are treated separately as their implementation has been done on different platforms and with different strategies. The system requirement which needed to be met was a throughput of 0.5ms for 16 simultaneous users. Nevertheless, this could not have been achieved due to the slow nature of the Ethernet module.

## 7.1 Hardware accelerators

Both of the modules have been implemented and tested in VHDL. For the current clock frequency, 100 Mhz, no timing violations were present. To paint a better picture regarding where the bottlenecks lie in hardware, throughput and latency were measured individually for both the neural network and the fusion and they can be seen in Table 7.1. The time for latency was considered between the time when the first valid value that was inputted to the module, and the time when this value would have been computed. To lower these values, one can have a bigger clock frequency which is achievable by pipelining the smaller modules, but at the same time increasing latency.

| **Module** | **Throughput(clock cycles)** | **Latency(clock cycles)** |
|---|---|---|
| Neural network | 200 | 429 |
| Fusion | 160 | 283 |

**Table 7.1:** Performance measurements for the implemented hardware modules

In Table 7.2 the utilisation rate of the *Neural Network* and the *Fusion* is presented for the RFSoC2x2 board. As the constraints for the hardware resources were higher in the beginning, the utilisation is fairly small.

Table 7.3 presents the accuracy achieved at different stages of the system. The average and maximum error have been computed for 100 inputs points. The NN results from the hardware implementation have been integrated with the 32-bit floating-point software model of the fusion. The error for the complete system implemented in hardware is shown as well.

| | **LUT** | **FF** | **DSP** |
|---|---|---|---|
| Available | 425280 | 850560 | 4272 |
| Neural Network | 43289 (10%) | 11377 (1%) | 629 (15%) |
| Fusion | 22569 (5.31%) | 6417 (0.75%) | 156 (3.65%) |

**Table 7.2:** The resource utilisation after implementation for different modules

| | **Average error(cm)** | **Max error(cm)** |
|---|---|---|
| Neural Network | 2.7 | 10.97 |
| Complete system | 2.7 | 10.97 |

**Table 7.3:** The error achieved at different stages of the system

### 7.1.1 Ethernet Communication System

The result for the Ethernet communication system is taken through Jupyter notebook using the PYNQ framework. This is because the processing system cannot be simulated in Vivado. Therefore the measurements have to be taken during run time and through the processing system, and it is no longer possible to check the results on a bit level.

The Ethernet System is made up of two different vital components, the DMA and the socket Ethernet implementation. To test the performance of the DMA, a small design is created that generates new data as soon as the DMA is ready to process it. The FPGA is clocked at 240MHz for all different tests.

In Table 7.4 and 7.5, the results for the DMA are presented. For these tests, the data is generated in the PL by a simple counter IP, and then the data is sent through the DMA's Stream to Memory Channel to the PS as controlled by Jupyter notebook. In Table 7.4 it is shown that the data rate varies greatly depending on the package size of the data transfer, when the package size is doubled, the data rate is almost doubled as well. The reasons for this are explored in the discussion.

| **Package Size** | **Data sent(Mb)** | **Run Time(s)** | **Data rate(Mb/s)** |
|---|---|---|---|
| 30 | 319.68 | 58.07 | 5.51 |
| 100 | 320 | 17.65 | 18.13 |
| 200 | 320 | 9.01 | 35.52 |

**Table 7.4:** Performance of the DMA through jupyters notebook with varying package size.

The performance of the Ethernet transmission together with the encode and decode function is shown in Tables 7.6 and 7.7.

| Package Size | Data sent(Mb) | Run Time(s) | Data rate(Mb/s) |
|---|---|---|---|
| 100 | 32 | 1.77 | 18.08 |
| 100 | 320 | 17.65 | 18.13 |
| 100 | 3200 | 177.38 | 18.04 |

**Table 7.5:** Performance of the DMA through jupyters notebook with varying amount of data sent.

| Package Size | Data sent(Mb) | Run Time(s) | Data rate(Mb/s) |
|---|---|---|---|
| 30 | 319.68 | 57.75 | 5.54 |
| 100 | 320 | 56.67 | 5.65 |
| 200 | 3200 | 57.35 | 5.58 |

**Table 7.6:** Performance of the Ethernet transmission, encode and decode function, with varying package size.

| Package Size | Data sent(Mb) | Run Time(s) | Data rate(Mb/s) |
|---|---|---|---|
| 100 | 32 | 5.84 | 5.48 |
| 100 | 320 | 56.67 | 5.65 |
| 100 | 3200 | 574.03 | 5.57 |

**Table 7.7:** Performance of the Ethernet transmission, encode and decode function, with varying amount of data sent.

The performance achieved by the Ethernet-DMA system is presented in Table 7.8. The performance is very low and only 1% of the available bandwidth, 1Gb/s, is utilised. The reasons for the low performance are explored in the discussion.

It is important to note that the code for the full system is run sequentially, which together with the low performance of the individual parts of the system explains the final result.

| Data sent(Mb) | Run Time(s) | Data rate(Mb/s) |
|---|---|---|
| 30.72 | 30.54 | 1.01 |
| 307.2 | 238.56 | 1.29 |
| 1536 | 1163.26 | 1.32 |

**Table 7.8:** The performance of the Ethernet system with custom DMA transmission

## 7.2 Discussion

### 7.2.1 FPGA utilisation

The final utilisation rate is fairly low for the RFSoC 2x2 board. This is due to the fact that the hardware accelerators were originally designed for the ZCU102 board, which has fewer available resources. This means that there are resources left to utilise for increasing the performance, either in regards to latency, throughput or accuracy when running it on this board.

To truly analyse the amount of resources that is suitable to use in a positioning system it would be good to analyse it together with a full massive MIMO system.

### 7.2.2 Accuracy of Hardware accelerators

The accuracy of the system degrades as it is translated from software into hardware. This is due to simplifications being done to the calculations to increase throughput, lower latency and reduced resources used, but the cost of it is the loss in accuracy. It's interesting to note that all the accuracy is lost in the Neural Network.

In Table 7.3 the accuracy after the Neural Network hardware implementation and the accuracy of the full system is presented. Compared to the accuracy of the software system, as presented in Table 4.3, the average error has doubled and the max error has also increased. This is a fairly large increase and makes it relevant to look at what can be done to minimise that.

The accuracy for the NN as presented in Table 7.3 is different from the accuracy presented for the picked quantization scheme in Table 4.3. This is due to the fact that rounding had to be introduced in a few spots in the hardware architecture to, for example, match the decimal points for the fixed point addition. The numbers are also rounded down into 16 bits just before the division in the batch normalization, to reduce the resource utilization and latency, and this causes most of that difference.

To increase the accuracy of the NN there are two main points of interest. The resolution of the division could be increased. There is also a technique that can be done during the training of the NN to better adapt it to a fixed point quantization. It is briefly mentioned in section 4.3.4, but as stated there it is not done in this thesis as the training of the NN is outside of the scope. It would be interesting to see what impacts that technique would have one the accuracy of the final system, and might be something that could be studied in future works.

No notable accuracy is lost in the fusion, as the resulting average error and maximum error values come very close to the 32-floating point software model. Although some of the approximations of the non-linear functions are introducing notable errors during different steps of the fusion, due to its structure, these differences are not substantial in the final result. This proves that the number representation choice has fulfilled its purpose of catering to the dynamic range of the numbers during different computation stages, thus preserving the initial accuracy. The disadvantages of this implementation is the increased implementation time, as well as the additional required hardware for the floating-point operations. This could have presented somewhat of a bottleneck for a different choice of FPGA.

### 7.2.3 Ethernet Communication System

All the parts of the Ethernet communication system have a lower performance than originally expected. Fig. 7.9 presents the performance of the DMA as written in Xilinx's own documentation. This data cannot be evenly compared to the results in this thesis since they are achieved on a different FPGA at a different clock frequency, but it can be used as a guide to see around what magnitude the performance should be. The performance of the DMA in the current set-up is far less. The reasons for this are most likely that the set-up time for the DMA through PYNQ is very large, pulling the performance down. In Table 7.4 it is shown that the data rate almost doubles when the package size, the amount sent per transfer, doubles, which clearly indicates that the majority of the time is taken up by a constant set-up and not the actual transfer.

| **Channel** | **Clock Frequency(MHz)** | **Throughput (MB/s)** | **Percent of theoretical** |
|---|---|---|---|
| MM2S | 100 | 399.04 | 99.76 |
| S2MM | 100 | 298.59 | 76.64 |

**Table 7.9:** Performance of the DMA as recorded in Xilinx's documentation[9].

The low performance of the socket class is probably due to two factors. The writers of this thesis do not have good background knowledge into this type of python code, which means that the software most likely does not utilise the available functionality in the best way. The second and larger reason is the incompatibility between the socket Ethernet implementation and the DMA transfer. As explained in Chapter 5, the Ethernet class and the DMA had to work with different memory buffers, and to move the data between these formats without dropping information, an encode and decode method had to be constructed. This method has to loop through all the values and translate them one at a time, which is very inefficient.

Table 7.5 shows the performance of the full system, which is lower than both sub modules. The reason for this is that the python code is written in a sequential fashion, which means the sub modules have to wait for each other and can not fully utilise the CPU on their own anymore. The class for controlling the DMA in PYNQ is also implemented using polling. This means that the DMA class is continuously checking whether the transfer is done or not, leaving the CPU unable to process the other tasks while the DMA transfer is happening.

In the Chapter 5 it was explained how this was the non-preferred implementation method, but that it was the only choice left due to licensing issues. These results confirm that thought. To fully utilise the implemented hardware accelerators, the communication system would need to be implemented in a different way, where the achievable data rate is much larger.

# Chapter 8
# Conclusion

With the new emerging technologies, there is obviously the need for efficient hardware acceelerators to support certain computationally-intensive applications. This thesis has managed to achieve a system based on Large Intelligence Surfaces, which is a potential enabler of 6G.

The hardware accelerators implemented in this thesis utilise different optimization techniques to increase throughput with the minimized amount of resources, while preserving accuracy at appeasable values. The current interconnect implementation can not satisfy the bandwidth need of the accelerators though, so to fully utilise them, a different interconnect implementation would still need to be created.

## 8.1 Future work

There are several more interesting ways this project could be expanded on in future works. As mentioned in the previous chapter, it would be interesting to explore the bandwidth achievable with the original Ethernet set-up, which was a hardware implementation, rather than a software application.

The CSI data used in this thesis is created through simulation, and not through measurements taken through a real massive MIMO system. It would be interesting to see how big impact the real data would have on the accuracy of the the training of the NN.

Another important factor in regards to using real data is that a big challenge for this kind of system is gathering the high amount of real-life labeled data with the accuracy demanded for training. If the technology is ever going to be viable to use in actual system outside of research, it will be necessary to decrease the cost of gathering in the live data. This can be done either through developing clever NN that needs less accurate training data or reducing the amount of data needed for the actual training. To reduce the amount of data needed there is, for example, one viable NN training technique called *Transfer Learning*, which uses knowledge gathered during a previous training round with a similar set-up to decrease the data needed for the new training

# References


[1] J. R. Sánchez, O. Edfors, and L. Liu, "Positioning for distributed large intelligent surfaces using neural network with probabilistic layer," in *2021 IEEE Globecom Workshops (GC Wkshps)*, 2021, pp. 1–6.

[2] E. G. Larsson, O. Edfors, F. Tufvesson, and T. L. Marzetta, "Massive mimo for next generation wireless systems," *IEEE Communications Magazine*, vol. 52, no. 2, pp. 186–195, 2014.

[3] Xilinx, "UltraScale Architecture DSP Slice - User Guide," Xilinx User Guide, Tech. Rep. UG579, August 2021.

[4] D. A. Micha, "An introduction to numerical analysis (atkinson, kendall e., ed.)," *Journal of Chemical Education*, vol. 57, no. 4, p. A142, 1980. [Online]. Available: https://doi.org/10.1021/ed057pA142.2

[5] Advanced Micro Devices, "PYNQ: PYTHON PRODUCTIVITY ," http://www.pynq.io/, [Online; accessed 12-Dec-2022].

[6] AMD Xilinx, "Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit," https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html, [Online; accessed 23-May-2023].

[7] ——, "Zynq UltraScale+ RFSoC," https://www.xilinx.com/products/silicon-devices/soc/rfsoc.html#gen1, [Online; accessed 23-May-2023].

[8] ARM, "Amba axi-stream protocol specification," 2021.

[9] Xilinx, "AXI DMA LogiCORE IP Product Guide - Performance ," https://docs.xilinx.com/r/en-US/pg021_axi_dma/Product-Specification, [Online; accessed 14-Dec-2022].

[10] World Economic Forum, "What is the Internet of Things?" https://www.weforum.org/agenda/2021/03/what-is-the-internet-of-things/, 2021, [Online; accessed 4-May-2023].

[11] Investopedia, "What Is the Internet of Things (IoT)? How It Works and Benefits," https://www.investopedia.com/terms/i/internet-things.asp, 2022, [Online; accessed 5-May-2023].

[12] Akhtar, M.W., Hassan, S.A. and Ghaffar, R., "The shift to 6G communications: vision and requirements," *Hum. Cent. Comput. Inf. Sci.*, vol. 10, no. 53, pp. 2031–2043, 2020.

[13] Startus Insights, "What You Need to Know about the Current State & Future of 5G Technology," https://www.startus-insights.com/innovators-guide/what-you-need-to-know-about-the-current-state-of-5g-technology/, 2020, [Online; accessed 5-May-2023].

[14] C. De Lima, D. Belot, R. Berkvens, A. Bourdoux, D. Dardari, M. Guillaud, M. Isomursu, E.-S. Lohan, Y. Miao, A. N. Barreto, M. R. K. Aziz, J. Saloranta, T. Sanguanpuak, H. Sarieddeen, G. Seco-Granados, J. Suutala, T. Svensson, M. Valkama, B. Van Liempd, and H. Wymeersch, "Convergent Communication, Sensing and Localization in 6G Systems: An Overview of Technologies, Opportunities and Challenges," *IEEE Access*, vol. 9, pp. 26 902–26 925, 2021.

[15] F. Boccardi, R. W. Heath, A. Lozano, T. L. Marzetta, and P. Popovski, "Five disruptive technology directions for 5G," *IEEE Communications Magazine*, vol. 52, no. 2, pp. 74–80, 2014.

[16] S. Hu, F. Rusek, and O. Edfors, "Beyond massive mimo: The potential of data transmission with large intelligent surfaces," *IEEE Transactions on Signal Processing*, vol. 66, no. 10, pp. 2746–2758, 2018.

[17] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang, "A high performance fpga-based accelerator for large-scale convolutional neural networks," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016, pp. 1–9.

[18] Rabey,Jan M & Chandrakasan, Anantha & Nikolic, Borivoje, *Digital Integrated Circuits: A Design Perspective, 2nd Edition*, 2003.

[19] The Editors of Encyclopaedia, "Moore's law ," https://www.britannica.com/technology/Moores-law, 2023, [Online; accessed 2-June-2023].

[20] Xilinx, Inc, "What is an FPGA?" https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html, [Online; accessed 16-May-2023].

[21] Y. Fu, E. Wu, A. Sirasao, K. K. Sedny Attia, and R. Wittig, "Deep Learning with INT8 Optimization on Xilinx Devices," Xilinx All Programmable, Tech. Rep. WP486, April 2017.

[22] Prakriti Gupta, "Introduction to Division Algorithm in Computer Architecture," https://www.codingninjas.com/codestudio/library/introduction-to-division-algorithm-in-computer-architecture, 2023, [Online; accessed 18-May-2023].

[23] Sumathy Eswaran, "Fixed Point Arithmetic : Division," https://witscad.com/course/computer-architecture/chapter/fixed-point-arithmetic-division, [Online; accessed 18-May-2023].

[24] *Taylor expansions and applications.* Milano: Springer Milan, 2008, pp. 223–255. [Online]. Available: https://doi.org/10.1007/978-88-470-0876-2_7

[25] Analog Devices, "Fixed-Point vs. Floating-Point Digital Signal Processing," https://www.analog.com/en/technical-articles/fixedpoint-vs-floatingpoint-dsp.html, 2021, [Online; accessed 22-May-2023].

[26] R. Yates, "Fixed-point arithmetic: An introduction," 2013.

[27] J.-M. Muller, N. Brisebarre, F. Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*, 01 2010.

[28] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

[29] ARM, "AMBA," https://developer.arm.com/Architectures/AMBA, [Online; accessed 10-Dec-2022].

[30] ——, "AXI protocol overview," https://developer.arm.com/documentation/102202/0300/AXI-protocol-overview, [Online; accessed 11-Dec-2022].

[31] ——, "Channel transfers and transactions," https://developer.arm.com/documentation/102202/0300/Channel-transfers-and-transactions, [Online; accessed 11-Dec-2022].

[32] P. P. Chu, *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability.* John Wiley & Sons, Ltd, 2006, ch. 9, pp. 257–311. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/0471786411.ch9

[33] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, "Efficient processing of deep neural networks: A tutorial and survey," vol. 105, no. 12, 2017.

[34] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016, http://www.deeplearningbook.org.

[35] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," vol. 15, no. 1, p. 1929–1958, jan 2014.

[36] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," ser. ICML'15. JMLR.org, 2015, p. 448–456.

[37] T. P. Hill, "Conflations of probability distributions," 2009.

[38] "tfp.distributions," https://www.tensorflow.org/probability/api_docs/python/tfp/distributions, [Online; accessed 30-May-2023].

[39] "tfp.distributions.MultivariateNormalTriL," https://www.tensorflow.org/probability/api_docs/python/tfp/distributions/MultivariateNormalTriL, [Online; accessed 30-May-2023].

[40] "Floating Point arithmetic in High Level VHDL," https://hardwaredescriptions.com/floating-point-in-vhdl/, [Online; accessed 30-May-2023].

[41] P. Nilsson, A. U. R. Shaik, R. Gangarajaiah, and E. Hertz, "Hardware implementation of the exponential function using taylor series," in *2014 NORCHIP*, 2014, pp. 1–4.

[42] P. Curtis, "Algorithms for division – part 4 – Using Newton's method," https://blog.segger.com/algorithms-for-division-part-4-using-newtons-method/, 2021, [Online; accessed 30-May-2023].

[43] S. Halder, A. Hasnat, A. Hoque, D. Bhattacharjee, and M. Nasipuri, "Pipelining based floating point division: Architecture and modeling," 2013.

[44] M. Kihl and J. A. Andersson, *Datakommunikation och nätverk*, 2020.

[45] Rita Horner, "Understanding the Ethernet Nomenclature – Data Rates, Interconnect Mediums and Physical Layer," https://www.synopsys.com/designware-ip/technical-bulletin/ethernet-dwtb-q117.html, [Online; accessed 27-May-2023].

[46] Advanced Micro Devices, "PYNQ SD Card image," https://pynq.readthedocs.io/en/latest/pynq_sd_card.html?highlight=sd, 2022, [Online; accessed 12-Dec-2022].

[47] , "AXI DMA v7.1 LogiCORE IP Product Guide," https://docs.xilinx.com/r/en-US/pg021_axi_dma/AXI-DMA-v7.1-LogiCORE-IP-Product-Guide, 2022, [Online; accessed 2-June-2023].

[48] ——, "DMA," https://pynq.readthedocs.io/en/v3.0.0/pynq_libraries/dma.html, [Online; accessed 2-June-2023].

[49] Gordon McMillan, "Socket Programming HOWTO," https://docs.python.org/3/howto/sockets.html, [Online; accessed 2-June-2023].

[50] Kishlay Verma, "Socket Programming in Python," https://www.geeksforgeeks.org/socket-programming-python/, 2023, [Online; accessed 2-June-2023].