**Circuits and Systems**
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ens.ewi.tudelft.nl/

**CAS-2021-4589734**

# M.Sc. Thesis

## Hybrid Posit and Fixed Point Hardware for Quantized DNN Inference

**Zep Kleijweg B.Sc.**

### Abstract

The recently introduced posit number system was designed as a replacement for IEEE 754 floating point, to alleviate some of its shortcomings. As the number distribution of posits is similar to the data distributions in deep neural networks (DNNs), posits offer a good alternative to fixed point numbers in DNNs: using posits can result in high inference accuracy while using low precision numbers. The number accuracy is most important for the first and last network layers to achieve good performance. For this reason, these are often computed using larger precision fixed point numbers compared to the hidden network layers. Instead, these can be computed using low precision posit, to reduce the memory access energy consumption and the required memory bandwidth. The hidden layer computation can still be performed using cheaper fixed point numbers.
An inference accuracy analysis is performed to quantify what the effect of this approach is on the VGG16 network for the ImageNet image classification task. Using 8 bit posit for the first and last network layer instead of 16 bit fixed point is shown to result in a top-5 accuracy degradation of only 0.24%. The hidden layers are computed using 8 bit fixed point in both cases.
The design of a parameterized systolic array accelerator performing exact accumulation is proposed that can be used in a scale-out system along with fixed point systolic array tiles. To increase hardware utilization, a hybrid posit decoder is designed to enable fixed point computation on the posit hardware. Using this hardware, the entire network can be computed using 8 bit data, instead of using 16 bits for some layers. This reduces energy consumption and the complexity of the memory hierarchy.

**TU Delft**

Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

# Hybrid Posit and Fixed Point Hardware for Quantized DNN Inference

Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Embedded Systems

by

Zep Kleijweg B.Sc.
born in Haarlem, The Netherlands

This work was performed in:

Circuits and Systems Group
Department of Microelectronics & Computer Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

**Delft University of Technology**

Copyright © 2021 Circuits and Systems Group
All rights reserved.

Delft University of Technology
Department of
Microelectronics & Computer Engineering

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science for acceptance a thesis entitled **"Hybrid Posit and Fixed Point Hardware for Quantized DNN Inference"** by **Zep Kleijweg B.Sc.** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: September 30, 2021

Chairman: ____________________________
prof.dr.ir. René van Leuken

Advisor: ____________________________
dr. Inayat Ullah

Committee Members: ____________________________
prof.dr. Zaid Al-Ars

# Abstract

The recently introduced posit number system was designed as a replacement for IEEE 754 floating point, to alleviate some of its shortcomings. As the number distribution of posits is similar to the data distributions in deep neural networks (DNNs), posits offer a good alternative to fixed point numbers in DNNs: using posits can result in high inference accuracy while using low precision numbers. The number accuracy is most important for the first and last network layers to achieve good performance. For this reason, these are often computed using larger precision fixed point numbers compared to the hidden network layers. Instead, these can be computed using low precision posit, to reduce the memory access energy consumption and the required memory bandwidth. The hidden layer computation can still be performed using cheaper fixed point numbers. An inference accuracy analysis is performed to quantify what the effect of this approach is on the VGG16 network for the ImageNet image classification task. Using 8 bit posit for the first and last network layer instead of 16 bit fixed point is shown to result in a top-5 accuracy degradation of only 0.24%. The hidden layers are computed using 8 bit fixed point in both cases.
The design of a parameterized systolic array accelerator performing exact accumulation is proposed that can be used in a scale-out system along with fixed point systolic array tiles. To increase hardware utilization, a hybrid posit decoder is designed to enable fixed point computation on the posit hardware. Using this hardware, the entire network can be computed using 8 bit data, instead of using 16 bits for some layers. This reduces energy consumption and the complexity of the memory hierarchy.

# Acknowledgments

With this thesis work, titled "*Hybrid Posit and Fixed Point Hardware for Quantized DNN Inference*", my master study comes to completion.
I would like to thank my supervisor René van Leuken for his assistance during the entire process of writing this thesis, and for providing me with the opportunity to pursue an interesting topic of research.
Special thanks also go to my daily supervisors, David Aledo Ortega, Ercan Kalali, and Inayat Ullah for their continued support during the entire project. You really helped steer my thesis towards interesting topics and ideas. The help you provided with the technical details is also much appreciated.
It goes without saying that I would not have been able to accomplish this work without the strong support from people I care about. I want to thank my mother and father, Annelies & Robert, and brother, Stijn, for their unwavering support during all of my studies. I want to thank my girlfriend, Marit, for always being there for me, either to motivate me or to provide some much needed distraction. I also want to thank my friends, for being understanding of not seeing me as much these past months, yet always being there for a relaxing chat along with a drink.

Thank you.

Zep Kleijweg B.Sc.
Delft, The Netherlands
September 30, 2021

# Contents

**Abstract** **v**

**Acknowledgments** **vii**

**1 Introduction** **1**
1.1 Motivation . . . 2
1.2 Thesis Goals . . . 2
1.3 Approach . . . 3
1.4 Thesis Contributions . . . 3
1.5 Outline . . . 3

**2 Number Systems & Arithmetic** **5**
2.1 Number Systems . . . 5
2.1.1 Fixed Point . . . 5
2.1.2 IEEE 754 Floating Point . . . 10
2.1.3 Posits . . . 12
2.1.4 Rounding . . . 19
2.1.5 Number System Comparison . . . 20
2.2 Arithmetic Operations & Hardware . . . 24
2.2.1 Integer Computation . . . 24
2.2.2 Rounding . . . 30
2.2.3 Floating Point . . . 31
2.2.4 Posits . . . 34
2.2.5 Hardware Comparison . . . 40
2.3 Related Works: Posit Hardware . . . 41

**3 Deep Neural Networks** **45**
3.1 Deep Neural Networks . . . 45
3.1.1 Two Network Architectures . . . 46
3.1.2 Network Components . . . 47
3.2 Inference Computation . . . 51
3.2.1 Compute Architectures . . . 51
3.2.2 Systolic Arrays . . . 53
3.2.3 Energy: Memory vs. Compute . . . 55
3.2.4 Quantization . . . 56
3.3 Related Works: Posit in DNNs . . . 58
3.4 Methodology: Hybrid Hardware . . . 60

**4 Inference Accuracy Analysis** **63**
4.1 Network: VGG16 . . . 63
4.1.1 Weight & Bias Quantization . . . 66
4.1.2 Activation Quantization . . . 67

4.2 Verification Dataset . . . . . . . . . . 68
4.3 Results . . . . . . . . . . 68

**5 Hardware Designs** **71**
5.1 Posit Systolic Array Design . . . . . . . . . . 71
5.1.1 Data Flow . . . . . . . . . . 71
5.1.2 Edge & Distributed Conversion . . . . . . . . . . 72
5.2 Posit Processing Element Design . . . . . . . . . . 74
5.2.1 Processing Element Design . . . . . . . . . . 74
5.2.2 Open-Source Posit Hardware . . . . . . . . . . 79
5.2.3 Encode Fixed Point Numbers to PIF . . . . . . . . . . 83
5.2.4 Quire to Fixed Point . . . . . . . . . . 90
5.2.5 Shorter Quire . . . . . . . . . . 91
5.2.6 Pipelining . . . . . . . . . . 92
5.3 Verification . . . . . . . . . . 94
5.3.1 Generating Verification Data . . . . . . . . . . 94
5.4 Results: Synthesis & Timing Analysis . . . . . . . . . . 96
5.4.1 Using Different Decoders . . . . . . . . . . 97
5.4.2 Quire Conversion . . . . . . . . . . 98
5.4.3 Processing Elements . . . . . . . . . . 98
5.4.4 Systolic Arrays . . . . . . . . . . 103

**6 Conclusions** **107**
6.1 Future Work . . . . . . . . . . 108

# List of Figures


2.1 Unsigned number representations. . . . . 7
2.2 Example of 1's complement number arithmetic, involving end-around carry. . . . . 9
2.3 Example of 2's complement number arithmetic, dropping the carry. . . 9
2.4 32 bit IEEE 754 floating point representation. . . . . 10
2.5 Representation of a posit number. . . . . 12
2.6 A comparison between the number distributions of 16 bit IEEE floating point, posit, and fixed point number systems. . . . . 22
2.7 Unsigned integer ripple-carry adder. . . . . 25
2.8 1's complement signed integer adder. . . . . 26
2.9 2's complement signed integer adder. . . . . 27
2.10 Carry out that is not an overflow. . . . . 27
2.11 2's complement adder with overflow detection, assuming signed input operands. . . . . 28
2.12 Binary multiplication of two 4 bit numbers. . . . . 28
2.13 4 bit array multiplier. . . . . 29
2.14 Fixed point arithmetic with example values, with 2 integer bits and 3 fraction bits. . . . . 30
2.15 Floating point multiplier. . . . . 34
2.16 Posit to PIF decoder, where clog is the ceiling of the log: $\lceil log_2(x) \rceil$. . 35
2.17 8 bit input leading zero counter using 2 modules with 4 bit inputs. . . 37
2.18 An 8 bit barrel shifter design. . . . . 38
2.19 PIF to posit encoder. . . . . 39

3.1 An example of a fully connected layer. . . . . 49
3.2 An example of a convolutional layer. . . . . 50
3.3 An example of the max pooling operation. . . . . 51
3.4 Example of a $5 \times 4$ systolic array. . . . . 54
3.5 A comparison between the distribution of posit numbers and the weight distribution in a DNN. Figures from [1]. . . . . 59
3.6 9 tile systolic array design. . . . . 61

4.1 VGG16 before and after adding quantization layers. . . . . 65

5.1 Output stationary systolic array of size $5 \times 4$. . . . . 72
5.2 Two different approaches to handle posit decoding and quire conversion in a systolic array architecture. . . . . 73
5.3 Posit processing element for exact accumulation. . . . . 75
5.4 Top-level distributed processing element for an output stationary data flow. . . . . 77
5.5 Top-level distributed processing element for an output stationary data flow, that reuses the quire registers for the converted results. . . . . 78
5.6 Top-level edge processing element for an output stationary data flow. . 79

5.7 The Deepfloat decoder design. . . . . . . . . . . . . . . . . . . . . . . . 85
5.8 Hybrid decoder design that can convert both posits and fixed point input numbers to the PIF. Posit and fixed point precision is assumed to be equal. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
5.9 Simple pipelining example. . . . . . . . . . . . . . . . . . . . . . . . . . . 92
5.10 Simplified schematic of a processing element, where the dotted lines indicate potential locations to introduce pipelining registers. . . . . . . 93

# List of Tables

2.1 Effective operations. . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.2 Complement number system modular arithmetic. . . . . . . . . . . . . 8
2.3 Number formats in the IEEE 754-2008 standard, where the +1 in significand length is the implicit bit. . . . . . . . . . . . . . . . . . . . . . 10
2.4 IEEE 754 special case representations. . . . . . . . . . . . . . . . . . . 12
2.5 Regime field examples and their interpreted values, where x is don't care. 13
2.6 Posit formats as described in [2], where the +1 in significand bits is the implicit 1 bit. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.7 Half & full adder truth table. . . . . . . . . . . . . . . . . . . . . . . 25
2.8 Truth table for a 4 input leading zero counter. . . . . . . . . . . . . . . 37

3.1 The architecture of the AlexNet network, where the input propagates from the top to the bottom. . . . . . . . . . . . . . . . . . . . . . . 47
3.2 The architecture of the VGG16 network, where the input propagates from the top to the bottom. . . . . . . . . . . . . . . . . . . . . . . 48

4.1 Simulation results for different number systems, where the hidden layers are quantized to 8 bit fixed point numbers. The single precision baseline uses no quantization in any layer. . . . . . . . . . . . . . . . . . . . 69

5.1 Alignment of a product to the quire. . . . . . . . . . . . . . . . . . . 82
5.2 The functionality of the neighboring bit XOR. . . . . . . . . . . . . . 84
5.3 How the leading zero count of the XORed input signal is interpreted, where LZC() is the leading zero count. The signals in brackets are the 2's complement representation of the signed integers. . . . . . . . . . . 86
5.4 Quire to fixed point example using (4,0)-posit quire and 6 bit fixed point. 91
5.5 Decoder implementation results. . . . . . . . . . . . . . . . . . . . . 98
5.6 Quire to fixed point converter. . . . . . . . . . . . . . . . . . . . . . 98
5.7 Implementation results of processing elements using regular posit decoding. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
5.8 Hybrid processing element results for different fixed point bit widths. . 100
5.9 Hybrid processing element results for different parameters, where the fixed point input precision is set to 8 bits. . . . . . . . . . . . . . . . . 101
5.10 Hybrid PE results using a shorter quire. The fixed point input precision is 8 bits. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
5.11 Pipelined PE implementation results. The fixed point input precision is set to 8 bits. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
5.12 Non-pipelined $9 \times 8$ systolic arrays, with different input fixed point precision. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
5.13 Pipelined $9 \times 8$ systolic array implementation results for different bit widths and with reduced quire sizes. . . . . . . . . . . . . . . . . . . 104

5.14 Pipelined $9 \times 8$ systolic array implementation results for different parameters and with reduced quire sizes, where the fixed point input precision is set to 8 bits. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105

# List of Acronyms

**AI** artificial intelligence.

**ANN** artificial neural network.

**ASIC** application-specific integrated circuit.

**BRAM** Block RAM.

**CNN** convolutional neural network.

**CPU** central processing unit.

**DCNN** deep convolutional neural network.

**DNN** deep neural network.

**DSP** digital signal processing.

**DUT** device under test.

**es** exponent size.

**FF** flip-flop.

**FMA** fused multiply-add.

**FPGA** field programmable gate array.

**FSM** finite-state machine.

**GPU** graphics processing unit.

**HLS** high-level synthesis.

**ILSVRC** ImageNet Large Scale Visual Recognition Challenge.

**IP** intellectual property.

**IS** input stationary.

**LOC** leading one counter.

**LSB** least significant bit.

**LUT** lookup table.

**LZC** leading zero counter.

**MAC** multiply-accumulate.

**ML** machine learning.

**MSB** most significant bit.

**NaN** not a number.

**NaR** not a real.

**NLR** no local reuse.

**OS** output stationary.

**PE** processing element.

**PIF** posit intermediate format.

**ReLU** rectified linear unit.

**RS** row stationary.

**RTL** register-transfer level.

**SA** systolic array.

**sf** scaling factor.

**TPU** tensor processing unit.

**ULP** unit in last position.

**unum** Type III universal number.

**useed** unum seed.

**WS** weight stationary.

# 1 Introduction

Deep learning is a machine learning technique that is used to extract meaningful, high-level information from large sets or streams of data. It has applications in for example image and speech recognition, which makes deep learning a widely studied topic. For image recognition, which is the application that is focused on in this thesis, the deep neural network (DNN) model is currently the most popular, as it has been found to be most effective [3]. This model uses multiple convolutional layers in order to extract more abstract features from the input data [4]. The accuracy that is achieved by DNNs has increased rapidly over the years, thanks to increasing model sizes and a huge increase in the amount of available data to train the networks with [5]. Of course, the progress in the field could not have been achieved without the huge increase in computational resources [5]. The high achievable accuracy has sparked interest for the use of DNNs in consumer electronics and on edge devices like drones [6] and autonomous vehicles [7].

In order to make the networks more powerful and perform more difficult tasks, larger networks can be used [5]. However, this also increases the number of operations that are required to compute accurate results. It also incurs a higher energy consumption, which is often the limiting factor in compute systems [8]. A technique that is often used to decrease energy consumption and memory overhead is to use fewer bits to represent numbers during the computation. This technique is known as quantization, and also helps to increase throughput: smaller precision operations require less hardware area, meaning more compute can be performed within the same area constraint.

The inference accuracy degradation caused by the use of low precision, fixed point data compared to single precision floating point data has been proven to be relatively small [9]. The use of the IEEE 754 floating point number representation [10] has become the standard for real number representation in the computing industry. However, the use of fixed point numbers for deep learning inference has been widely adopted, as the lower accuracy of the numbers does not have a significant impact on the network accuracy and the hardware to compute with fixed point numbers is less expensive in terms of area, power, and delay. A recently introduced number system has some promising characteristics, making it another viable alternative.

Gustafson introduced the posit number system in 2017 [11]. It achieves a larger dynamic range and accuracy than IEEE 754 floating point numbers for the same number of bits, and has a number distribution that is quite similar to typical data distribution in DNNs [1]. This means that low precision numbers can be used, with a less severe accuracy degradation but with the benefit of reduced memory footprint and energy consumption. Unfortunately, this does come at the cost of increased hardware area overhead and power requirement compared to fixed point numbers: posit hardware area is comparable to floating point hardware [12].

A popular approach to increase the performance of DNN computation is to use an

accelerator, rather than performing the computation on a general purpose central processing unit (CPU) and graphics processing unit (GPU). An accelerator implements a compute architecture in hardware that is optimized for a certain application. This makes hardware accelerators less flexible, but also means that they can be faster, and more area and energy efficient, than a general purpose compute platform. A specific architecture that is a good fit for the acceleration of DNN inference is that of the systolic array (SA), where all data passes through multiple compute cores in sequence before the result is again stored in the memory hierarchy [13]. This type of architecture has the benefit that an algorithm that involves multiple operations on each datum sequentially can be computed without the latency and energy consumption of continuously gathering the same data from memory. It is however less general purpose and often needs to be adjusted to work for a specific application, requiring new hardware designs.

## 1.1 Motivation

The desire to deploy high performance DNN applications in edge devices means energy efficient compute systems are required. The use of lower precision number representations has been explored in research, but with the posit number system another option has been introduced that can come with a smaller accuracy penalty. The recent introduction of the number system means that high effort hardware designs to compute with posits are sparse and often not open source. This makes it difficult for designers to make a well informed trade-off as to what suits their application.

## 1.2 Thesis Goals

This thesis studies the posit number system, and specifically its use in a systolic array based hardware accelerator for DNN applications. The posit number system offers a different number distribution compared to IEEE floating point numbers, that is well suited for use in a DNN compute system. The systolic array architecture reuses data, which reduces expensive memory accesses.
The goal of this study is to design a posit SA that can be used in a hybrid compute system that can compute with both posit and fixed point data representations. This way, posits can be used for those network layers that benefit from the increased accuracy posits offer, while the cheaper fixed point number system can be used for the rest of the computation. The posit computations that replace the fixed point computation will also use a smaller bit width, to reduce the memory access energy consumption and the required memory bandwidth.
The aim of this study is to investigate the impact this hybrid computation approach has on the network inference accuracy. Different trade-offs that can be made in the design of the SA will also be evaluated based on their hardware area and delay.

## 1.3 Approach

First the different number systems will be introduced, and their properties compared. Then the hardware structures that can be used to compute with them are presented, identifying their differences. Next, an introduction into DNNs is presented, to discover how the use of the posit number system introduces a benefit. Subsequently, the effect on DNN inference accuracy of using posit numbers for part of the computation will be evaluated, using software to simulate posits on floating point hardware.
Next, a processing element that performs exact accumulation of posit products is designed, that forms the basic module the systolic array is built from. The processing element is designed using parameters, such that it can easily be used to compute with different bit width numbers. Different trade-offs in its design are evaluated. The design of the systolic array is also parameterized to easily change the size of the array. All the hardware designs are implemented in SystemVerilog and verified using reference data.

## 1.4 Thesis Contributions

The following summarizes the main contributions of this thesis:

- An elaborate explanation and comparison of the fixed point, floating point, and posit number systems, as well as the hardware that is used to compute with them.

- An inference accuracy analysis of a post-training quantized deep neural network using posit number system for the first and last network layer, while using fixed point numbers for the hidden layer computation.

- Design and implementation of a reconfigurable hybrid systolic array that can compute exact accumulation with both the posit and fixed point number system. This SA can be used in combination with fixed point SA tiles to form a hybrid compute system.

## 1.5 Outline

The content of this thesis is organized as follows:

- Chapter 2 introduces and compares the different number systems and the basic hardware structures to compute with them. Related work on posit hardware designs is also discussed.

- Chapter 3 provides an introduction to deep neural networks, the hardware architectures that can be used to with compute them, and quantization. Related work about using posit in DNNs is also discussed.

- Chapter 4 explores the effect of using both posit and fixed point numbers in the same DNN computation, to get the best out of both number systems.

- Chapter 5 describes the design of the posit processing element and systolic array architecture, along with the specific optimizations and trade-offs that are implemented and evaluated.
- Chapter 6 concludes this thesis and presents possible future work.

# Number Systems & Arithmetic 2

In this chapter, three relevant number systems are described that can be used in a DNN compute system. First, in Section 2.1, their basic working is discussed, followed by the hardware structures that can be used to compute with the numbers in Section 2.2. Even though there are significant differences between the number systems, the arithmetic operation structures have a lot of overlap. Finally, Section 2.3 discusses some related work regarging posit compute hardware.

## 2.1 Number Systems

In a digital computing system, all information is stored in binary digits, or bits. How these bits are interpreted, determines what information they represent. For example, the same byte can be interpreted as an ASCII character, a negative integer, or a positive fraction. In this section, two major number representations are discussed, that are in use already for a long time in a wide variety of systems: the fixed point number system, and the floating point number system. A third, relatively new number system is also discussed, the Type III universal numbers called posits. This number system attempts to take the useful things from the IEEE 754 floating point standard, and improve upon its weak points.
In the following discussion, the word precision is used when referring to the total bit width of a number. Accuracy is used to say something about the error a number system introduces: a smaller difference in value of subsequent representable numbers introduces a smaller error, so achieves a higher accuracy. Dynamic range describes the ratio between the largest magnitude and the smallest magnitude a certain number representation can assume.

### 2.1.1 Fixed Point

In a fixed point number system, also called Q-point [14], the location of the decimal point is determined beforehand, during the design of a computing system. The decimal point can not dynamically change from one position to another. The simplest example are unsigned integers, where the decimal point is placed after the least significant bit (LSB) and the value is always positive or zero. By moving the decimal point to the left, rational numbers can be represented and real numbers approximated more accurately. For representing signed values, there are numerous different representations that can be used, which will also be discussed.

#### 2.1.1.1 Unsigned Numbers

Unsigned numbers are positive, including zero. This can be useful when the designer knows beforehand that negative numbers will never be used in the system, preventing the need to also encode the sign of the numbers thus saving hardware and enlarging the dynamic range of the representable numbers for the same precision.

**Integers** Integers can be represented as in Fig. 2.1a, where $b_{N-1}$ is the most significant bit (MSB) and $b_0$ the least significant bit (LSB). Converting this number to a decimal number can be done using (2.1). Numbers in the range $[0, 2^N - 1]$ can be represented with N bits of precision.

$$y_{base10} = \sum_{i=0}^{N-1} b_i * 2^i \tag{2.1}$$

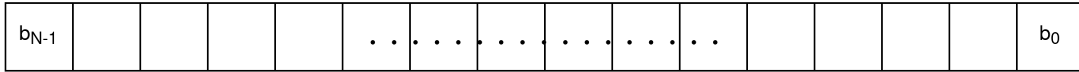
**Rational Numbers** By moving the binary point, the number is divided into an integer part and a fractional part, as shown in Fig. 2.1b. In the figure, the integer part is represented by $N$ bits, and the fractional part by $M$ bits. The conversion to a decimal number can be done using (2.2). The numbers that can be represented are in the range $[0, 2^N - 2^{-M}]$.

$$y_{base10} = \sum_{i=-M}^{N-1} b_i * 2^i \tag{2.2}$$

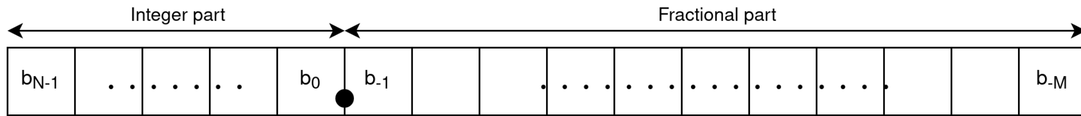
Since only a finite number of bits can be used to represent the fraction, the accuracy of the numbers that can be represented is limited: the numbers that can be represented are always an integer multiple of the value that is assigned to the LSB: $2^{-M}$. This is the often called the unit in last position (ULP), and gives the quantisation step between consecutive numbers in the number system. It also immediately shows that a larger fractional part results in the more accurate representation of real numbers, since the ULP is smaller. However, it also means that there will always be a quatization in the representation of real numbers. When a number needs to be stored that can not be exactly represented in a certain number representation, it needs to be rounded. This process is discussed in more detail in Section 2.1.4.
The precision of a real number is the total number of bits that are used to represent the number: $N + M$. By varying the numbers $N$ and $M$, a trade-off can be made between the accuracy of the numbers that can be represented, and their dynamic range: a larger $M$ will increase the accuracy, but reduce the dynamic range when using the same precision.

#### 2.1.1.2 Signed Numbers

When the numbers that need to be represented can also be negative, a signed representation needs to be used. Different methods exist to encode the sign, three of which are




(a) Unsigned integer representation with N bits.




(b) Unsigned representation with an M bit fractional part and an N bit integer part.

Figure 2.1: Unsigned number representations.

discussed here. Signed numbers also make subtraction possible where the result can be negative. Effective subtraction refers to the cases where the effective mathematical operation is subtraction, including for example the addition of operands of opposite sign (see Table 2.1 for each of the cases). For signed numbers, an extra bit is typically used to indicate whether the number is positive or negative, where the convention is that the MSB of a number indicates the sign, 1 indicates a negative number and 0 a positive number [15].

Table 2.1: Effective operations.

| Operands | Effective operation |
|---|---|
| (+X) + (+Y) | Addition |
| (+X) + (-Y) | Subtraction |
| (-X) + (+Y) | Subtraction |
| (-X) + (-Y) | Addition |
| (+X) - (+Y) | Subtraction |
| (+X) - (-Y) | Addition |
| (-X) - (+Y) | Addition |
| (-X) - (-Y) | Subtraction |

**Sign & Magnitude** Sign & magnitude representation is conceptually quite a simple one: the magnitude is represented in the same way as for the unsigned case, and an extra bit is used to indicate whether the number is positive or negative. Representing a number in this way is quite straightforward and intuitive, other advantages include its symmetric range and easy sign change. For integers, the representable range using sign & magnitude representation is $[-(2^{N-1}-1), 2^{N-1}-1]$. The downside of a symmetric range is that there are two representations for zero ($+0$ and $-0$), requiring separate detection in the case of comparisons. The major disadvantage, however, is that effective subtraction can not be computed using the same hardware as effective addition: subtraction might require borrowing from a more significant digit, while addition will only carry partial results over to more significant digits. This means addition requires

a magnitude comparator to determine the sign of the result in case of effective subtraction, and a separate subtractor circuit next to the adder. Another approach would be to use a complement representation internally, such that the same hardware can be used for addition and subtraction. However, in order to prevent the repeated cost of switching between the different representations it would be more efficient to just use the complement representation throughout the system, which is exactly what is generally being done in computing systems [15]. Two slightly different complement representations are discussed next.

**1's Complement** In a complement number system, a negative value $-x$ is represented as the unsigned value $C - x$ where $C$ is a suitable complementation constant that can be decided by the designer [15]. Addition is computed as the sum of the unsigned representations in modulo $C$, irrespective of the sign of the operands. Table 2.2 shows these calculations, keeping in mind that in modular arithmetic adding $C-1$ is the same as subtracting 1. This makes that the addition is the same for all operands, independent of their sign, as long as their complement $C - x$ can be calculated when necessary for effective subtraction. Depending on the value chosen for $C$, this complement can be quite cheap to calculate. This is the major strength of complement number systems: at the cost of selectively complementing the operands, the rest of the addition process is the same for effective subtraction and addition, preventing the need for separate hardware.

Table 2.2: Complement number system modular arithmetic.

| Desired operation | Operation in mod $C$ |
|---|---|
| (+X) + (+Y) | X + Y |
| (+X) + (-Y) | X + (C - Y) |
| (-X) + (+Y) | (C - X) + Y |
| (-X) + (-Y) | (C - X) + (C - Y) |

An easy operation to obtain the complement of a number is the bitwise inversion in case it is negative, which is exactly what is done in the 1's complement number representation. Since this operation can be done entirely in parallel, only a few inverters are necessary that introduce minimal delay. The range of a 1's complement number system is $[-(2^{N-1} - ULP), 2^{N-1} - ULP]$, with a redundant representation for zero (both all-1's and all-0's). The ULP can be $2^{-M}$ for fixed point, or 1 in case of integer numbers. Bitwise inversion corresponds to subtracting a number from an all-1 vector, or $C = 2^N - ULP$, resulting in $C - x = 2^N - ULP - x$. In order to perform addition modulo this $C$, a carry out that is produced is fed back to the LSB to be added: the dropped carry has a value of $2^N$, the bit added to the LSB has value ULP, resulting in a reduction of the result equal exactly to the modulus.
This principle is known as end-around carry: the carry out of the MSB position is added to the LSB position, effectively doubling the maximum possible length of carry propagation. An example calculation involving end-around carry is shown in Fig. 2.2. What the effect is on the hardware is discussed in Section 2.2.1.2. End-around carry can be quite expensive for high precision numbers, especially because carry propagate

adders are often found in the critical path of a compute system. In order to prevent this overhead, a complement number system with a different modulus can be used.

0 1 1 1 ($=7_{10}$)
1 0 1 0 ($=-5_{10}$) +
1 0 0 0 1
end-around carry → 1 +
0 0 1 0 ($=2_{10}$)

Figure 2.2: Example of 1's complement number arithmetic, involving end-around carry.

**2's Complement** Choosing $C = 2^N$ results in the 2's complement number system. The complementation of a number is slightly more involved than in the case of 1's complement: it requires bitwise inversion and the addition of ULP. This can be seen from the equation that calculates the complement in the case of 1's complement: $C - x = (2^N - ULP - x)_{1's\ complement} + ULP = 2^N - x$. The range of the 2's complement system is asymmetric: $[-2^{N-1}, 2^{N-1} - ULP]$, which has the benefit of only having one representation for zero, all-0's. The addition of two numbers can be done by simply dropping the carry-out, since this has the same value as the modulus: $2^N$. An example calculation in the 2's complement representation can be seen in Fig. 2.3.
At first sight it seems like the complementation needed for subtraction makes this operation significantly more expensive than bitwise inversion, since it requires an addition and potentially full carry propagation. However, because complementation is only needed for computing the subtraction of two numbers, the addition of ULP can be incorporated into this computation at marginal cost. How this is done exactly is again discussed in Section 2.2.1.2 in more detail. The fact is that it is a lot cheaper than computation in 1's complement notation, which has made 2's complement the most widely used method for signed number representation in modern computing systems.

0 1 1 1 ($=7_{10}$)
1 0 1 1 ($=-5_{10}$) +
1 0 0 1 0 ($=2_{10}$)
↓
drop carry

Figure 2.3: Example of 2's complement number arithmetic, dropping the carry.

### 2.1.2 IEEE 754 Floating Point

As opposed to a fixed point number system, the decimal point in a floating point number system is not placed at a fixed position. The most widely used floating point representation is the IEEE 754 standard [10], the 2008 revision of which is discussed in detail here. The decimal representation definitions that are also in the standard are not discussed here, since these are not of interest. When using the word float, it refers to a floating point number adhering to the IEEE 754 standard.
The representation of the number is split into three different parts, much like signed fixed point numbers. In this case however, the integer part is replaced by an exponent to increase the dynamic range of the representable numbers by effectively moving the position of the decimal point depending on the value that is represented. Fig. 2.4 shows the general representation standard, and (2.3) can be used to calculate the decimal value of a floating point number when it is not one of the special representations that are discussed later. Table 2.3 shows the different (binary) format lengths defined in the standard, where the significand length includes the implied bit in front of the fraction corresponding to the +1 in (2.3).

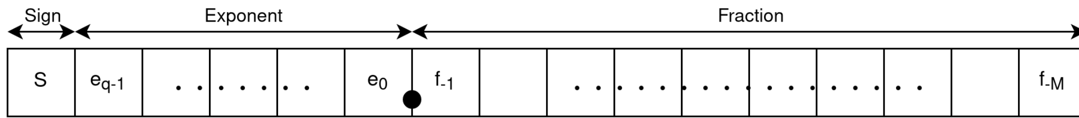



Figure 2.4: 32 bit IEEE 754 floating point representation.

$$y_{base10} = (-1)^{MSB} * 2^{exp-bias} * (1 + fraction) \tag{2.3}$$

Table 2.3: Number formats in the IEEE 754-2008 standard, where the +1 in significand length is the implicit bit.

| Bits | Common name | Significand bits | Exponent bits | Exponent bias |
|---|---|---|---|---|
| 16 | Half precision | 1 + 10 | 5 | 15 |
| 32 | Single precision | 1 + 23 | 8 | 127 |
| 64 | Double precision | 1 + 52 | 11 | 1023 |
| 128 | Quadruple precision | 1 + 112 | 15 | 16383 |

The first term of (2.3) represents the sign of the numbers, that is encoded in the MSB of the number according to the same convention as for signed integer and fixed point numbers. For the exponents, a biased representation is used. This allows representing integers in the range $[-bias, 2^q - bias]$, where $q$ is the length of the exponent, as unsigned integers in the range $[0, 2^q - 1]$ when the bias is chosen to be $2^{q-1}$. The reason this biased representation is rarely seen anywhere else than in the exponents of floating points is because they require correction of the bias even after simple arithmetic, as is shown in (2.4). Multiplication and division become even more complicated. However, using a

biased representation for the exponents allows for easy zero detection and magnitude comparison for floating point numbers.
In the floating point standard a slightly different bias value of $2^{q-1}-1$ is used. This is because the all-0's and all-1's exponents are used to represent exception cases, and with this bias the smallest representable exponent has a value of 1 after biasing. Because the exponents are represented as unsigned values, magnitude comparison can be done without having to decode the signs, simplifying the operation. Since the exponents of floating point numbers are only added and subtracted, for multiplication and division of floating point numbers respectively as shown in (2.5), these benefits outweigh the downside of the necessary bias correction. On top of that, the delay of computation on the significands is typically longer than the delay of the exponent computation, effectively hiding the delay caused by the bias representation from the critical path.
The third term of (2.3) is called the significand and is in the range $[1, 2)$, as the fraction is in the range $[0, 1)$. The addition of 1 to the fraction is implicit, and not represented by a bit in the binary number representation in order to save space. The fraction is calculated in the same way as the fraction of a fixed point number.

$$\begin{aligned} x + y + bias &= (x + bias) + (y + bias) - bias \\ x - y + bias &= (x + bias) - (y + bias) + bias \end{aligned} \tag{2.4}$$

$$\begin{aligned} 2^x * 2^y &= 2^{x+y} \\ \frac{2^x}{2^y} &= 2^{x-y} \end{aligned} \tag{2.5}$$

#### 2.1.2.1 Special Representations

The IEEE 754 standard was designed in such a way that each operation has a well-defined result, for example also division by zero. This means that certain representations are reserved in order to represent special cases: not a number (NaN) is used to represent undefined numbers, like $0/0$, $0 * \infty$, and $\sqrt{-1}$. There are also representations for zero and infinity, which are both signed numbers in the standard, that do no follow the general rule of interpretation as given in (2.3). The different cases are shown in Table 2.4, where the value of subnormals can be calculated using (2.6): instead of an implicit 1 in the significand, a 0 is used. This is done to accurately represent smaller magnitude values close to zero, with a magnitude smaller than the smallest normal magnitude, hence their name. Effectively, these numbers are represented with less bits for the fraction, not utilizing the most significant bits. Using subnormal numbers results in gradual underflow, rather than replacing a value that is smaller than the smallest normal number by zero.

$$y_{base10} = (-1)^{MSB} * 2^{exp-bias} * (0 + fraction) \tag{2.6}$$

From the way NaNs are represented can be seen that there is a lot of redundancy: there are $2^{fraction\ bits+1}$ different ways to represent a NaN, where the +1 originates from the sign bit. As can be seen from Table 2.3, most bits are used to represent the fraction, making this redundancy rather expensive: these representations could also be used to represent actual values. There also appears to be slight redundancy for representing

Table 2.4: IEEE 754 special case representations.

| Value | Exponent | Fraction |
|---|---|---|
| Subnormal | all-0's | non-zero |
| NaN | all-1's | non-zero |
| $\pm\infty$ | all-1's | all-0's |
| $\pm$ 0 | all-0's | all-0's |

signed 0, yet the sign still carries some information: division by zero will for example result in a signed infinity, which can be important depending on the application. However, the signed zero representation and all the different NaN representations make comparison between numbers difficult, for example for $x < y$ and $x = y$, complicating the hardware to compute the result.

### 2.1.3 Posits

In 2017 John L. Gustafson introduced the Type III universal number (unum) [11], that contains two different modes: valid mode and posit mode. In valid mode, a unum represents a range of real numbers much like interval arithmetic. This mode is not really relevant to the contents of this research, so the focus is on posit mode. In posit mode, a unum represents a real number of fixed size and behaves quite similar to floats. A unum in posit mode is referred to as a posit, for brevity and in accordance with the literature.
The posit representation consists of four different fields, as can be seen in Fig. 2.5: the sign, regime, exponent, and fraction. The regime field is of variable length and consists of a sequence of the same bits terminated by one bit of opposite value, as shown in Table 2.5 for a maximum length of 4 bits. The zero sequence that occupies the entire length of the number is not assigned a regime value, as it indicates either of the two exception values discussed later. To calculate the value of a posit number (2.7) can be used, where $k$ is the value of the regime, $e$ the value of the exponent, and es the maximum exponent size. This equation is also often written as (2.8), where $useed = 2^{2^{es}}$ which is short for unum seed. The term $2^{k*2^{es}+e}$ is often called the scaling factor (sf), as it determines the effective exponent of the number.

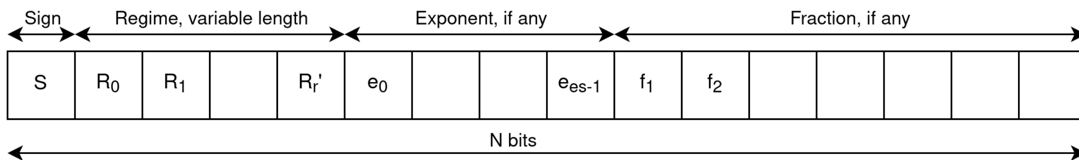



Figure 2.5: Representation of a posit number.

$$y_{base10} = (-1)^S * 2^{k*2^{es}+e} * (1 + fraction) \tag{2.7}$$

Table 2.5: Regime field examples and their interpreted values, where x is don't care.

| Regime field | 0000 | 0001 | 001x | 01xx | 10xx | 110x | 1110 | 1111 |
|---|---|---|---|---|---|---|---|---|
| Value (k) | x | -3 | -2 | -1 | 0 | 1 | 2 | 3 |

$$y_{base10} = (-1)^S * useed^k * 2^e * (1 + fraction) \tag{2.8}$$

The precision of a number, N, and the maximum exponent length, es, are the only two parameters that are determined during the design of a system using posits, and can be chosen with complete freedom according to the designers needs. If 19 bits are sufficient to achieve the desired dynamic range and accuracy, numbers of 19 bits can be used. The notation (N, es)-posit is often used to indicate the parameters of the posit number that is being used. Nevertheless, the posit standard draft [2] does speak of four standard formats with set parameters. These are summarized in Table 2.6, where the quire and PIF length are explained in the next sections.

Table 2.6: Posit formats as described in [2], where the +1 in significand bits is the implicit 1 bit.

| Bits | Common name | Max. significand bits | Max. exponent bits (es) | Quire bits | PIF length |
|---|---|---|---|---|---|
| 8 | Posit8 | 1 + 5 | 0 | 32 | 12 |
| 16 | Posit16 | 1 + 12 | 1 | 128 | 21 |
| 32 | Posit32 | 1 + 27 | 2 | 512 | 38 |
| 64 | Posit64 | 1 + 58 | 3 | 2048 | 71 |

The regime field is run-length encoded, meaning that its length determines the value $k$: $k = -r$ if $R_0 = 0$, and $k = r - 1$ if $R_0 = 1$, with $R_0$ and $r$ as in Fig. 2.5. This is also summarized in Table 2.5. Since the length of the regime is variable it is different for different numbers in the same number system. It can even occupy all of the $N - 1$ bits that are not used by the sign. In this case it also does not need to be terminated by a bit of the opposite value. This also means that even though es is defined to be a certain length, this indicates the maximum length of the exponent and it can be shorter as well, or not present at all. If the regime is short and the exponent also fits into the number representation, the remaining bits are used for the fractional part. If the fraction and/or (part of) the exponent bits are not present in the representation of a number, they are assumed to be zero when decoding the value of the number. Since the number of fraction bits also changes between numbers, the accuracy of a number depends on its value: numbers that have a large magnitude exponent have a longer regime, so less fraction bits and accuracy. Numbers that are close to 1.0 in magnitude have the highest accuracy, as most bits are being used for the fraction. The accuracy reduces both when a number is closer to zero in magnitude, and for larger magnitude numbers: the number of regime bits increases which reduces the number of fraction bits. This is known as tapered accuracy, and is explained in more detail in Section 2.1.3.2.
The exponent is represented by an unsigned integer without a bias. As the value of

the regime can be negative, there is no need for the exponent to also be negative to make the exponent of the scaling factor negative. Again, the scaling factor is given by $2^{k*2^{es}+e}$. Because the contribution of the regime value $k$ to the scaling factor is multiplied by $2^{es}$, the exponent of the scaling factor can express any signed integer in its range: the regime scales the value of the exponent with signed integer steps of $2^{es}$, and the exponent field with any power of two in the range $[2^0, 2^{es-1}]$. This makes the exponent field easy to interpret.
Whenever the sign bit of a posit is 1, the 2's complement of the number needs to be computed before decoding the rest of the fields. This prevents the need for a representation of negative zero, along with the complications of detecting different bit patterns that represent the same real value [11]. This also makes that integer comparison hardware can be used to compare posit numbers.

#### 2.1.3.1 Special Representations

There are only two bit patterns that are interpreted differently in the posit number system: the all-0's vector that means zero, and the sign being 1 with the remainder of the bits being zero. This represents not a real (NaR), which is $\pm\infty$ or "the point at infinity". Unlike in IEEE 754, it is not signed and the need for a developer to represent a NaR is considered a clear indication that an application is still in development [11]. For the same reason, if the result of a calculation exceeds the maximum representable number it should not overflow to infinity according to the posit arithmetic rules: doing so would turn a finite error into an infinite error. The same goes for underflow to zero, which means losing all the information. Instead, the result should be the minimum representable number.
For convenience, these two numbers get their own name: *minpos* is the smallest magnitude, non-zero value that is expressible in a given posit format, and *maxpos* is the largest value expressible in a given posit format. *Minpos* is the posit number with only the LSB set to 1 and the rest to 0, and *maxpos* is the posit number with only the sign set to 0 and the rest set to 1. Their values can be computed according to (2.9), where $-(N-2)$ and $N-2$ are the minimum and maximum regime value for an $N$ bit posit respectively.

$$\begin{aligned} minpos &= 2^{-(N-2)*2^{es}} \\ maxpos &= 2^{(N-2)*2^{es}} \end{aligned} \tag{2.9}$$

Handling the result of potential overflow or underflow is left up to the programmer, making it visible in the source code. The same goes for possible NaR values, which should never occur as a result from calculations with posits since rounding to infinity does not occur. This comes from the different approach that is used in the posit system, which makes the programmer or computer language responsible for handling illegal computations like division by zero. While posit hardware should still detect and handle NaR values, it does so completely silently: if something goes wrong, it can be dumped to NaR without alerting the user or setting any flags. This can for example be useful when processing an array of data where exceptions can be discarded. However,

the programmer should be aware of this and handle it accordingly in places where it might be a problem.
This is illustrated by Gustafson in [11] with an example: the computation of the square root of a negative number. While a programmer might object that this is an exception for which specific behavior is required, the posit standpoint is that this is a software bug and not something that the hardware should be taking care of. The programmer should figure out where it might be a problem and take care of it in those cases, for example by checking input operands to the square root function. This way the hardware is not paying a penalty in maximum performance for every operation, instead exceptions are handled only when required.
Therefore, posits are developed not to handle all exceptions in hardware but instead target speed, simplicity, and economy. This is most important for many computer users, according to Gustafson [11]. For those concerned with rigorous computing, bounding results and tracking all accuracy loss, or those still checking the validity and behavior of a numerical algorithm, using valid mode instead of posit mode is the answer.

#### 2.1.3.2 Tapered Accuracy

The number of bits that are being used for the fraction changes with the magnitude of the number that is represented in the posit number system. When the scaling factor has a large exponent magnitude, the regime uses more bits and less bits remain to represent the fraction. This means that numbers that can be represented with the minimum regime length of 2 bits are represented with most fraction bits. In an N bit posit, 1 bit is used for the sign, making the maximum number of fraction bits equal to $N - 1 - 2 - es$. The range of numbers that can be represented with this number of fraction bits is $\pm[2^{-2^{es}}, 2^{2^{es}})$. As posits have a symmetric number distribution around zero the same range applies to both positive and negative numbers.
The given range is around 1.0, and there are as many values that can be represented in the range $[0, 2^{-2^{es}})$ as in $[2^{2^{es}}, maxpos]$. As an example, for (8,1)-posit the range that uses the maximum number of fraction bits is $[\frac{1}{4}, 4)$, or $[\frac{1}{4}, 4 - \frac{1}{8}]$, while $minpos = \frac{1}{4096}$ and $maxpos = 4096$. This shows the impact of tapered accuracy: the posit number system has a large dynamic range with a high accuracy for a smaller range of numbers. In Section 2.1.5, a figure that shows the number distribution of posit numbers is shown and compared to the fixed and floating point systems.

#### 2.1.3.3 Posit Intermediate Format

The length of all posit fields, except the sign, is variable, meaning it is difficult to compute with posits directly. Therefore, posits have to be decoded before computation. The posit intermediate format (PIF) can be used as an intermediate representation in which a posit number is represented with fixed field sizes [16, 17]. The fields are as follows:

- A NaR flag of 1 bit,
- A flag to indicate zero of 1 bit,

- A sign of 1 bit,
- An exponent of $w_E$ bits,
- A fraction of $w_F$ bits.

The NaR flag is used to prevent the necessity of checking for NaR in each arithmetic operator, instead just the flag can be checked which reduces the complexity. For the same reason a flag is used to indicate a posit of value zero: it is interpreted differently to other posits and needs to be detected separately. The sign just contains the sign bit of the original posit number.
The exponent is the combination of the regime and exponent field, so it represents the scaling factor. The value of the regime is multiplied by $2^{es}$ for the scaling factor exponent, which is equivalent to shifting it to the left by es bits. The value of the exponent can then be represented in these shifted in bits. The maximum regime value $k$ is achieved when all of the bits after the sign are 1, in which case $k_{max} = N - 2$ that can be represented using $\lceil log_2(N-2+1) \rceil$ bits: say $N = 6$, then $k_{max} = 4$ which requires $\lceil log_2(4+1) \rceil = 3$ bits. Since the exponent can also be negative, an extra bit is required for the sign (which is unrelated to the sign of the complete number). The width of the exponent in the PIF can be computed using (2.10). The exponent is represented in a biased representation, similar to floats, with a value $bias = (N - 2) * 2^{es} + 1$. This bias value is chosen to represent the smallest representable exponent with a value of 1 in the biased representation, since the maximum exponent magnitude of the scaling factor $E_{max} = (N - 2) * 2^{es}$.

$$w_E = 1 + es + \lceil log_2(N - 1) \rceil \tag{2.10}$$

The maximum fraction width $w_F$ occurs whenever the regime is the shortest possible: a length of 2 bits. As 1 bit is used for the sign, (2.11) can be used to calculate the width of the fraction in the PIF. This makes the total width of the PIF representation $w_{PIF} = w_E + w_F + 3$, as also shown in Table 2.6.

$$w_F = N - (3 + es) \tag{2.11}$$

In [16] it is suggested to include two more bits in the PIF, namely the round and sticky bits that are used for correct rounding after a computation. However, it makes more sense to only represent these when they are required, to prevent unnecessary interconnect and registers. For example, after decoding a posit these two bits will always be zero. If the decoded posit is then stored in registers to be used for computation in the next clock cycle, it would be a waste to also store these two bits that are known to be zero anyway. Therefore, these bits are not included in the PIF here, but handled separately when required during computation.
After a posit number has been decoded into its corresponding PIF representation, all the fields have a constant width. While this format requires more wiring, the computational hardware would need to be wide enough to calculate with the widest possible input fields anyway. After the calculation is done, the result that is represented in the intermediate format needs to be encoded back to a regular posit format. The

hardware structures used for decoding and encoding posit numbers are discussed in Sections 2.2.4.1 & 2.2.4.3, respectively.

#### 2.1.3.4 Quire

In posit arithmetic, there is a quire that can be used to exactly compute the dot product of two vectors [2], along with the operations that can be constructed using a dot product. Exact addition is an example: if the second input vector contains only the value 1, the elements of the first input vector will be exactly added. Exact computation in this case means there is no intermediate rounding of the products or the accumulated sum, only the final result is rounded once. This increases the accuracy of the final result.
The quire is based on the Kulisch accumulator [18, 19], which can be used for the exact accumulation of floating point numbers and products. However, the IEEE 754 standard does not require an exact accumulator to be present [10]. Even so, exact accumulation has also been researched for floating point numbers [20]. Floating point arithmetic does require the fused multiply-add (FMA) to be present, which combines the multiplication of two numbers and the addition to a third into a single operation with a single rounding. While this does increase the accuracy of the result of a dot product compared to separate multiplication and addition, it does not offer the same benefits as exact accumulation.
One of the benefits of exact accumulation is that it makes the multiply-accumulate (MAC) operation associative, and with that provides the same results between different compilers and hardware implementations. While regular addition should be associative, rounding intermediate results breaks this property [21] as displayed in (2.12), where $round()$ is any rounding function and the symbol $\not\equiv$ means that the two expressions are not equivalent, even though their value may be equal in some cases. Computing the dot product, or MAC operation, using the quire can be done with a single rounding as in (2.13), increasing the accuracy of the final result as well as keeping the associative property of addition intact. This also means the same results are achieved when using different compilers or compute systems for the same computation, which is not necessarily the case when exact accumulation is not used [21].

$$round(round(a+b)+round(c+d)) \not\equiv round(round(round(a+b)+c)+d) \quad (2.12)$$

$$round(\sum_{i=0}^{i=N-1} a_i * b_i) \quad (2.13)$$

The quire is basically a fixed point number with enough integer and fraction bits to exactly represent posit products. To make sure products can be added to the quire exactly, it needs to be able to represent both the smallest and largest possible products of a certain posit representation. This means that it should consist of enough integer bits to represent $maxpos^2$, and enough fraction bits to represent $minpos^2$. An extra bit is required for the sign. The quire is used to store intermediate results during accumulation, so extra integer bits should be used to protect the quire from overflow. How

many integer bits should be used however, depends on which of the posit documentation you are reading.
According to [11], the quire should accommodate at least a billion products to be accumulated, requiring an additional 30 integer bits to prevent overflow independent of the posit parameters the quire will be used with. The total number of quire bits should then be rounded up to the nearest power of 2, to make the hardware cleaner and to allow for easy reading and writing of the quire to memory. In the posit standard draft [22] it is mentioned to use $N-1$ bits to prevent quire overflow, allowing at least $2^{N-1}-1$ sums of products before overflow. This assumes that longer accumulations are used when a larger number precision is required, which may not be true for all applications. However, it does keep the cost of the quire down for smaller precision posits.
With $minpos^2 = 2^{-(N-2)*2^{es}*2}$ and $maxpos^2 = 2^{(N-2)*2^{es}*2}$, the number of fraction and integer bits for the quire can be computed using (2.14) & (2.15) respectively. In (2.15) $N-1$ bits are assumed for preventing overflow, and the +1 is because $X+1$ bits are required to represent the integer $2^X$.

$$Quire\ fraction\ bits = \lceil |log_2(minpos^2)| \rceil = 2*(N-2)*2^{es} \tag{2.14}$$

$$Quire\ integer\ bits = N-1+\lceil log_2(maxpos^2) \rceil + 1 = N+2*(N-2)*2^{es} \tag{2.15}$$

$$Quire\ bits = sign + quire\ integer\ bits + quire\ fraction\ bits = 1+N+4*(N-2)*2^{es} \tag{2.16}$$

Another benefit of an exact accumulation is that it solves the magnitude difference problem. Since large magnitude numbers are spread further apart in number systems that use an exponent for moving the binary or decimal point, the addition of a small magnitude number might have no effect on the sum after rounding, resulting in the situation in (2.17) where the product results in a small magnitude number and $a$ has a large magnitude [23]. Posits are specifically vulnerable to this, because they use tapered accuracy: large magnitude numbers are further apart than what would be the case without tapered accuracy.

$$round(a+b*c) = a \tag{2.17}$$

Even though the situation in (2.17) is the expected behavior for a single operation, it can become problematic when computing the dot product of two vectors. The continued addition of small products can still have a significant effect on the accumulated value, resulting in a large final error. Using (4,0)-posit as an example this problem can become more clear.
All representable values for (4,0)-posit are $\{0, \pm 0.25, \pm 0.5, \pm 0.75, \pm 1, \pm 1.5, \pm 2, \pm 4\}$. For simplicity, consider the continued addition of 0.5 with intermediate rounding:

$$round(round(round(round(round(round(0.5+0.5)+0.5)+0.5)+0.5)+0.5)+0.5) = 2.0$$

Performing the same accumulation using the quire will result in a different result, because only one rounding operation will be done:

$$round(0.5 + 0.5 + 0.5 + 0.5 + 0.5 + 0.5 + 0.5) = round(3.5) = 4.0$$

This result is a lot closer to the result of an unrounded computation. The impact of rounding intermediate results becomes larger when the data does not only consist of positive numbers, but also negative numbers:

$$round(round(round(round(round(round(0.5 + 0.5) + 0.5) + 0.5) + 0.5) + 0.5) - 0.5) = round(2.0 - 0.5) = 1.5$$

$$round(0.5 + 0.5 + 0.5 + 0.5 + 0.5 + 0.5 - 0.5) = round(2.5) = 2.0$$

As displayed in the case above, a single negative number can throw off the result of the entire accumulation when each intermediate result is rounded. In the case of exact accumulation the error is due only to a single rounding operation, providing a more accurate result. For accumulations with a larger number of terms, more intermediate rounding occurs, possibly degrading the quality of the result. Of course performing exact accumulation using the quire does come at a cost.
The downside of using the quire is that quite a wide register needs to be used, as can also be seen from Table 2.6. Especially for high precision numbers, the cost of the quire becomes quite large. In order to add numbers or products to the quire a wide shifter and adder are required: to shift the summand to the correct (fixed point) position according to the scaling factor, and then add it to the quire, respectively. The hardware to compute with the quire is explained in more detail in Section 5.2.1.

### 2.1.4 Rounding

When a number can not be exactly represented in a given representation, it needs to be rounded. This can for example happen after a multiplication operation, which increases the precision of a real number: the example $0.1 * 0.1 = 0.01$ shows that the result requires a wider fraction to be exactly represented. Rounding can be done in any arbitrary manner, but the IEEE 754 standard defines a few techniques that can be used.
The different rounding algorithms are the following, where the first is the default method for IEEE 754 floating point numbers, and the only one that should be used according to the posit standard:

1. Round to nearest, ties to even
2. Round to nearest, ties away from zero
3. Round towards zero
4. Round towards +infinity
5. Round towards -infinity

The simplest rounding method is to simply ignore the extra, least significant bits that do not fit into the number representation, which is often called truncation. For positive numbers in a sign & magnitude representation this means always rounding down even when all the bits are 1, while for a negative number this equals always rounding up: in short, the number is rounded towards zero as in rounding method 3. In a 2's complement representation however, the result is always rounded down when truncated: or towards -infinity as in rounding method 5. The benefit of truncation is that it is free: no hardware is required to compute the correct rounding of a number. The downside is that all of the information in the bits that are truncated is lost.
For the default rounding method of rounding to the nearest representable number, with a tie rounded to the nearest even number, some more work is necessary. Since rounding to a larger magnitude number requires incrementing it, this might cause full carry propagation through the fraction and require renormalization of the number, possibly making rounding quite an expensive operation. In this rounding method, numbers are rounded to the nearest number that can be accurately represented in the number system: for integer numbers this means, for example, that all the numbers in the range $(2.5, 3.5)$ will be rounded to the value 3. Numbers that are exactly in the middle of two representable numbers, or "ties", are rounded to even numbers. In the same example, this means 2.5 is rounded to 2 while 3.5 is rounded to 4. Even numbers have the property that the final bit is set to 0.
How this rounding operation can be achieved is easier to explain clearly after having explained multiplication, and is therefore explained in Section 2.2.2. It is however not required to store all of the bits during computation that can not be represented in the final result: three bits suffice to ensure correct rounding. These are called the guard, round, and sticky bits.
When rounding posits, it is important to keep in mind that bits that do not fit into the representation are not necessarily all fraction bits: they can be exponent bits as well. In this case, "nearest" means "nearest exponent", and the tie point is not the arithmetic mean of the two adjacent numbers, as in (2.18) that is used for rounding fractional parts, but the geometric mean as in (2.19). While it may make more sense from a mathematical point of view to round to the nearest posit using the arithmetic mean as a tie point in every case [24], this would mean the hardware needs to detect whether the rounded bits are part of either the exponent or fraction. This increases the hardware cost, which is probably the reason posits are not rounded in this manner.

$$mean_{arithmetic} = \frac{x+y}{2} \tag{2.18}$$

$$mean_{geometric} = \sqrt{x*y} \tag{2.19}$$

### 2.1.5 Number System Comparison

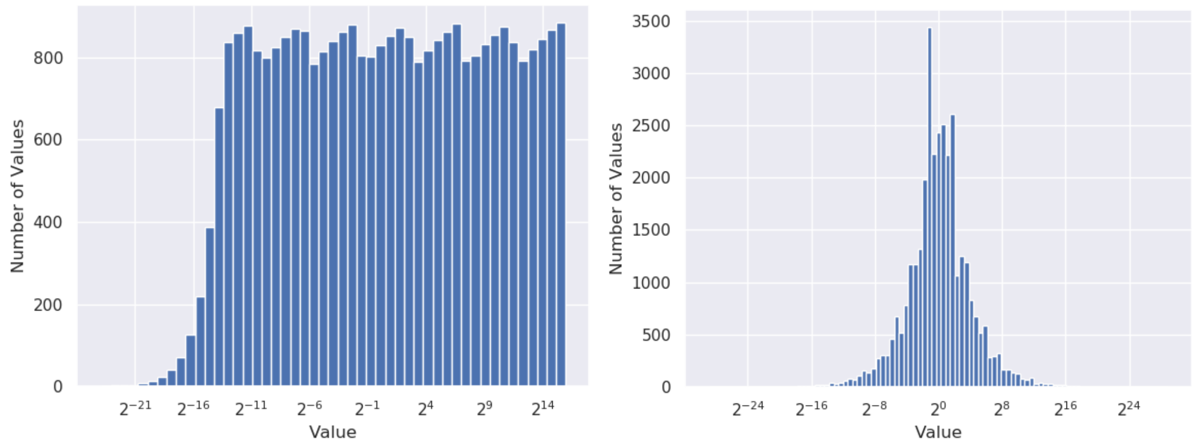
Each of the different number systems has its own benefits and disadvantages, for example computational simplicity or high accuracy. Here, a comparison is made between some key characteristics of the different systems. A comparison of the hardware to compute with the numbers is made in Section 2.2.5. The benefits of each of the number

systems for use in DNN application is discussed in Section 3.2.4.

#### 2.1.5.1 Number Distribution

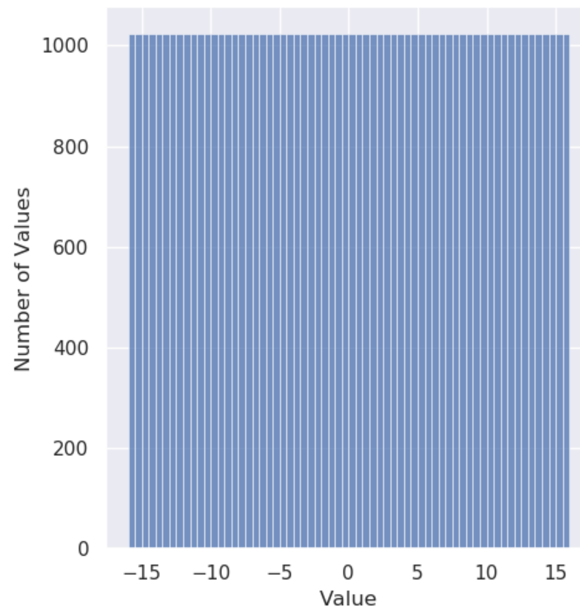
The different number systems have significantly different number distribution, which can play an important role in choosing the right number system for a specific application. If a number system is chosen that has a similar number distribution to the data that it needs to represent, less bits are probably required to achieve the required accuracy. The number distributions of 16 bit floating point, posit, and fixed point number systems can be seen in Fig. 2.6, and is discussed in this section.
From the number distribution of half precision floating point numbers in Fig. 2.6a can be seen that floats are sort of uniformly distributed when plotted on a logarithmic scale. This is because the number of fraction bits is constant for each number, independent of the value of the exponent. Therefore there are equally many numbers that can be represented for each exponent value. The decreasing number of representable small magnitude numbers on the left of the graph are caused by the gradual underflow that is introduced by subnormal numbers. For subnormal numbers, the effective number of fraction bits decreases for smaller magnitude numbers.
The distribution of (16,1)-posit is shown in Fig. 2.6b. This shows a distribution where the number of representable values decreases with the magnitude of the numbers, even on a logarithmic axis. This is because posit uses tapered accuracy: when the magnitude of the exponent increases, the regime becomes longer at the cost of fraction bits. This also displays that it can be a risk to use posits near overflow: their accuracy is significantly reduced, *maxpos* for (16,1)-posit is $2^{24}$. The same is actually true near underflow, even though it is not necessarily clear from the figure. If it is required to represent data near overflow or underflow, it might be safer to extend the precision of the number representation by 1 or 2 bits to ensure higher accuracy.
The fixed point number distribution in Fig. 2.6c is plotted on a linear axis, which clearly shows the limited dynamic range of the fixed point number system. All the numbers are uniformly distributed though, which can be beneficial when the data is known to be uniformly distributed over a certain range as well.

When comparing the dynamic range of float and posit using Fig. 2.6 it can be seen that posit can represent numbers over a wider range. A formal definition of dynamic range is given in (2.20), where *max* and *min* refer to the largest and smallest magnitude numbers that can be represented in the number system. For half precision floats this gives $log_{10}(\frac{2^{15}*(2-2^{-10})}{2^{-24}}) = 12.0$, while for (16,1)-posit it is $log_{10}(\frac{2^{28}}{2^{-28}}) = 16.9$ decades. At the same time, the number of values that can be represented in a range around $2^0$ is using posit is larger than for floating point. This means that in this range, posits achieve a higher accuracy than floats. For numbers with a larger magnitude exponent however, floats are represented more accurately. Fixed point numbers can only represent numbers over a small range, but can do so with large accuracy. The dynamic range and accuracy of fixed point numbers is a trade-off that can be made by changing the number of integer and fraction bits that are used to represent the




(a) Number distribution of 16 bit, half precision floating point with a $log_2$ x-axis.

(b) Number distribution of (16,1)-posit with a $log_2$ x-axis.




(c) Number distribution of 16 bit fixed point with 4 integer bits with a linear x-axis.

Figure 2.6: A comparison between the number distributions of 16 bit IEEE floating point, posit, and fixed point number systems.

numbers.

$$Dynamic\ range = log_{10}(\frac{max}{min}) \tag{2.20}$$

In 1971 Robert Morris suggested a tapered floating point number system that uses an extra field that indicates the length of the exponent field, and with that the length of the fraction field [25]. In that paper he wrote: ”It seems likely that most applications

which require the greatest accuracy of representation also are likely not to generate numbers of extremely large or extremely small magnitude. In other words, users of floating-point numbers are seldom, if ever, concerned simultaneously with loss of accuracy and with overflow. If this is so, then the range of possible representation can be extended to an extreme degree and the slight loss of accuracy will be unnoticed."

Even though the quote is quite old, it probably remains true: in most cases the high accuracy that can be achieved by large magnitude floating point numbers is not required. The posit number system implements the same idea of using tapered accuracy, but does so using run length encoding instead of a separate field. The result is similar though: the accuracy of the representable numbers drops with the increasing magnitude of the exponent. Depending on the application this can come at only a slight loss in accuracy, as is shown in more detail in Section 3.3.

While the posit documentation of course hurries to show examples where posits perform better than floating points [26], there are also examples where this is not the case. Some are provided in [24], for example the fact that most physics constants are represented less accurately in (64,3)-posit than they are in double precision floating point. This implies computations using those constants will also be less accurate when using posits instead of floats. It is therefore important to make a proper trade-off when selecting between the use of posits or floats for a specific application, and to choose the number system that is most suited to represent the required data.

#### 2.1.5.2 Redundancy

In the IEEE 754 number representation there are a lot of different bit patterns that represent NaN, as can be seen from Table 2.4. Whenever the exponent is all-1's, the number is either infinity or NaN. The original idea was that the fraction bits could be used to provide useful information about what sort of exception has occurred [10]. In practice, however, it seems this feature is often not supported [21]. Therefore, a lot of representations are reserved to represent the same thing. For single precision floats, with a length of 32 bits, 23 bits are used for the fraction. This means that, including the sign and excluding infinities, there are $2^{23+1} - 2 = 16777214$ different representations for NaN, which is 0.39% of the total number of representations. While this percentage is quite small, it becomes larger for smaller precision numbers: for 16 bit, half precision floats it becomes 3.12%.

Posits, on the other hand, have been designed to include no redundancy, not even for signed zero and infinity. The value zero does not follow the general rule of how to decode the posit number, and needs to be detected separately. The same goes for NaR, but since the bit vector is very similar to zero most of the hardware can be reused for detection. The remaining bit patterns have a distinct meaning, all following the same rule for interpretation. Fixed point representation does not use any exceptions, and as such there is no redundancy at all.

The downside of redundancy is that multiple bit representations are being used to represent the same value, instead of representing distinct values. These representations could also have been used to extend the dynamic range or accuracy of the numbers. On top of that, redundancy also complicates the hardware to compute with them as discussed in Section 2.2.5.

#### 2.1.5.3 Interpretation

As fixed point and floating point numbers have fixed size fields, they can be more easily computed with than posits. Due to the run length encoding of the regime, the location and length of the exponent and fraction field changes between numbers in the same representation. The need to decode posits complicates their computation and also comes with a hardware and delay overhead.

## 2.2 Arithmetic Operations & Hardware

Each of the number systems require different hardware to correctly compute the desired results. However, the devil is in the details here and a lot of the major components that are used for the different number systems are quite similar and can be reused. Division will not be discussed, since it is more complicated than addition and multiplication and not an important operation in the targeted application. First, integer and fixed point hardware is discussed since it is simplest. After that, floating point and posit hardware is described. Only basic designs are discussed, to get some insight into the basic differences between the different operations and number systems. Depending on the designers needs there are a lot of different optimizations that can be done to make a trade-off between hardware area, delay, latency and energy. These optimizations are not the focus of this discussion and are therefore not described.

### 2.2.1 Integer Computation

The hardware to compute with integers does not require much control, making it the most straightforward to implement. However, it is important to note that arithmetic operations increase the precision of the result compared to the precision of the input operands. For $x$-bit operands the results after addition will be $x + 1$-bits, while the result after multiplication will require $2 * x$-bits. If the width of the data path remains the same width after the computation, the result may need to be rounded and overflow detected.

#### 2.2.1.1 Addition

The basic module out of which adders are build are called full adders. These have three input bits, one for both input bits of the corresponding index, and a carry input from the previous index. It has two output bits, a carry that propagates to the next index and the sum result of the current index bits. The truth table for the full adder module can be found in Table 2.7. There are many ways to implement a full adder module, depending on the speed and area requirements as well as the used technology.
Essentially, a full adder is a single bit adder. By using multiple of them, an adder can be designed that adds larger precision numbers. Fig. 2.7 shows how to use full adder modules to construct a basic ripple-carry adder for unsigned integer addition. It is called a ripple-carry adder because the carry can propagate from the LSB position all the way through the adder to the MSB position. As can be seen in the figure, a half adder is used for the LSBs of the input, since there is no carry input for this position.

A half adder is a simpler version of the full adder, and its truth table corresponds to the first four rows of Table 2.7.
The carry output of the final full adder can be used as overflow detection: if a carry is produced from the most significant position, the result of the addition does not fit into the same number of bits as the input operands. Typically the datapath after the addition is only as wide as the input operands, meaning the larger result can not be represented. In this case, the overflow signal can be used to detect that a result has been produced that is too large in magnitude to be correctly represented. How this should be handled depends on the application, and the overflow signal can be used as a control signal to trigger handling the exception.

Table 2.7: Half & full adder truth table.

| A | B | $C_{in}$ | S | $C_{out}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

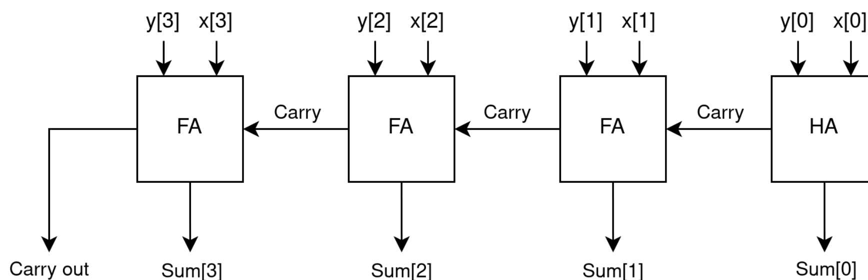



Figure 2.7: Unsigned integer ripple-carry adder.

#### 2.2.1.2 Subtraction

In order to enable subtraction, a signed number representation needs to be used. As discussed before in Section 2.1.1.2, using a complement number system has the benefit that the same hardware can be used for addition and subtraction. First, 1's complement hardware is described in order to show its major drawback. After that, 2's complement is discussed.

**1's Complement** Typically, for a fixed point or integer carry propagate adder, the LSBs are added using half adder cells, whereas all the other bits are added using full

adder cells as shown in Fig. 2.7. Half adders do not have a carry input, since there can be no carry in at the lowest index in unsigned addition. However, when using a 1's complement number system a "carry in" at the LSB position can be used for the end-around carry.

An adder/subtractor circuit for computation with 1's complement numbers is shown in Fig. 2.8. The control signal that indicates whether the addition or subtraction operation is required is used to selectively complement the second input operand. Alternatively, an inverter and multiplexer can be used for each input bit instead of XOR gates, where the mux select is again the control signal for addition or subtraction.

The feedback from the MSB carry bit to the LSB position is the end-around carry that is necessary for 1's complement addition. This figure also illustrates why this is quite undesirable in a computation: the end-around carry can cause another carry being created in the LSB position, propagating through the rest of the adder bits. This increases the length of the critical path, reducing the maximum operating frequency.

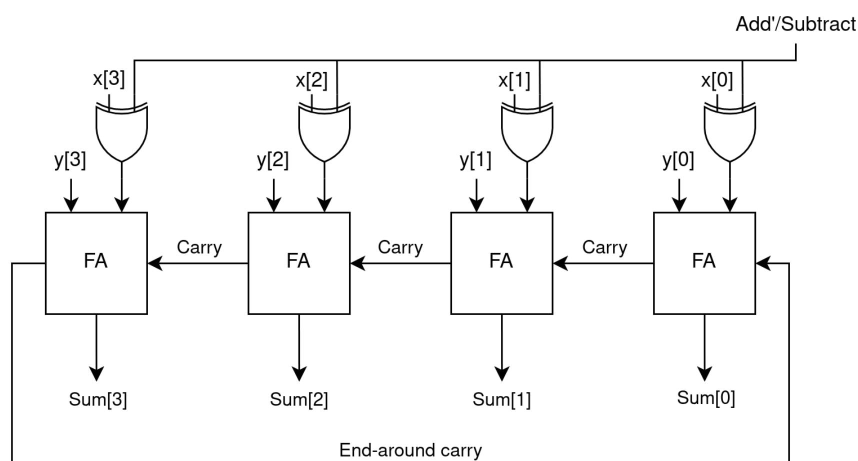



Figure 2.8: 1's complement signed integer adder.

**2's Complement** In a 2's complement adder, the addition of a carry in to the LSB can be used in a different manner: for the complementation of the operand in case of subtraction. Complementing the input operand in the case of subtraction requires an addition to the LSB. Instead of performing this addition only on the operand that is being subtracted, the addition of ULP can be performed during the computation by using the carry in of the adder. This prevents the need to use two separate adders in case of subtraction, saving hardware and delay. An adder/subtractor circuit for computation with 2's complement numbers is shown in Fig. 2.9, where the XOR gates are used to selectively invert the second operand in case of subtraction.

**Overflow Detection** In order to handle overflows in the 2's complement adder, some more hardware is required since the carry out can not be used on its own to detect

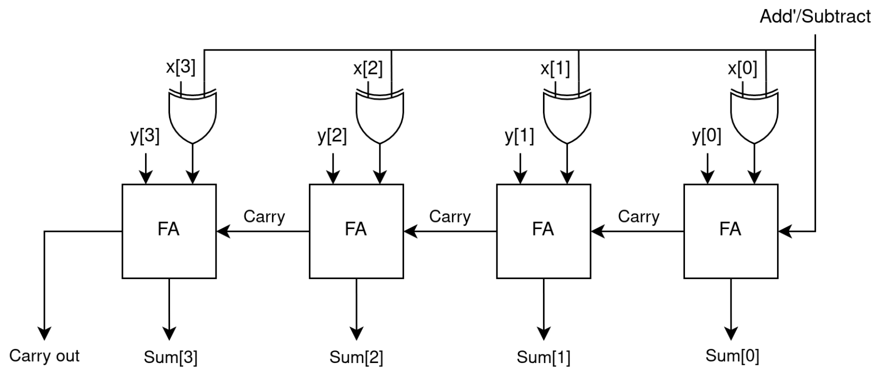



Figure 2.9: 2's complement signed integer adder.

overflow. Since the MSBs indicate the sign, a carry out does not always indicate an overflow as shown in Fig. 2.10. The adder circuit with overflow detection is shown in Fig. 2.11, where the assumption is that the input operands are already in 2's complement representation and that the operation is always addition: the circuit is not a subtractor, even though effective subtraction can occur due to negative input operands. The only difference in hardware compared to the unsigned case is the determination of the overflow signal. The reasoning behind this overflow detection scheme is that a carry into the sign position should also produce a carry output if there is no overflow. Otherwise, the sign of the result is not correct with regard to the input operands indicating an overflow.

0 1 1 1 (=$7_{10}$)
1 0 1 1 (=-$5_{10}$) +

1 0 0 1 0 (=$2_{10}$)
↓
drop carry

Figure 2.10: Carry out that is not an overflow.

#### 2.2.1.3 Multiplication

There are multiple different algorithms that can be used for multiplication, each requiring a specific hardware architecture as well. From Fig. 2.12a a first approach becomes clear: using the bits of the second operand to selectively add the first operand to the correct indices of the result. This can be implemented as a sequential algorithm [15],

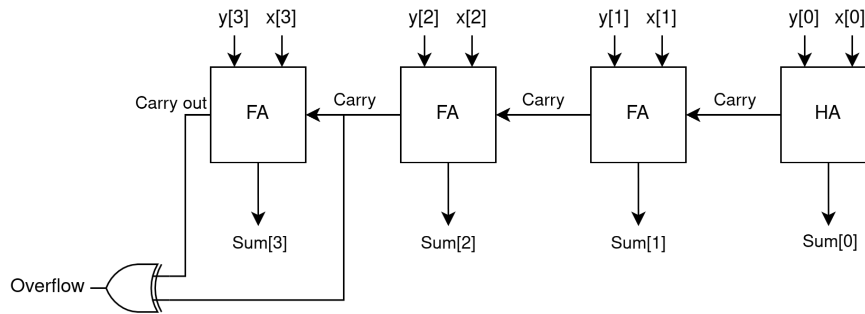



Figure 2.11: 2's complement adder with overflow detection, assuming signed input operands.

where the cumulative partial product is initialized to zero and each cycle the correctly shifted term $b_i * a * 2^i$ is added until all the partial products are added to the cumulative result, where the power of 2 corresponds to the required shift for each index. Because this approach is sequential, it can take many clock cycles before the result of the computation is complete and the next computation can begin.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | $a_3$ | $a_2$ | $a_1$ | $a_0$ | |
| | | | | $b_3$ | $b_2$ | $b_1$ | $b_0$ | * |
| | | | $b_0$*( | $a_3$ | $a_2$ | $a_1$ | $a_0$ | ) |
| | | $b_1$*( | $a_3$ | $a_2$ | $a_1$ | $a_0$ | ) | |
| | $b_2$*( | $a_3$ | $a_2$ | $a_1$ | $a_0$ | ) | | |
| $b_3$*( | $a_3$ | $a_2$ | $a_1$ | $a_0$ | ) | | | + |
| $c_7$ | $c_6$ | $c_5$ | $c_4$ | $c_3$ | $c_2$ | $c_1$ | $c_0$ | |

(a) Partial products for multiplication.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | $a_3$ | $a_2$ | $a_1$ | $a_0$ | |
| | | | | $b_3$ | $b_2$ | $b_1$ | $b_0$ | * |
| | | | | $b_0a_3$ | $b_0a_2$ | $b_0a_1$ | $b_0a_0$ | |
| | | | $b_1a_3$ | $b_1a_2$ | $b_1a_1$ | $b_1a_0$ | | |
| | | $b_2a_3$ | $b_2a_2$ | $b_2a_1$ | $b_2a_0$ | | | |
| | $b_3a_3$ | $b_3a_2$ | $b_3a_1$ | $b_3a_0$ | | | | + |
| $c_7$ | $c_6$ | $c_5$ | $c_4$ | $c_3$ | $c_2$ | $c_1$ | $c_0$ | |

(b) Expanded multiplication terms.

Figure 2.12: Binary multiplication of two 4 bit numbers.

For this reason a different approach is more popular for high speed designs, that is based more on the expanded multiplication terms as in Fig. 2.12b. This multiplier design is known as an array multiplier, because it uses an array of full adder cells to compute the multiplication result. The hardware structure of such a multiplier can be seen in Fig. 2.13, where each of the bitwise product terms $a_ib_j$ is computed using a two input AND gate as shown explicitly for $c_0$. The benefit of this design is that it can be easily pipelined to increase the operating frequency and the utilization of each of the components. Even though this increases latency just as the sequential algorithm described before, new operands can actually be applied to the array multiplier in each clock cycle rather than having to wait for the previous computation to be completed. This increases the throughput of the multiplier.
As can be seen from Fig. 2.13 the structure of an array multiplier is very regular, and only short wires are required to connect the components. This leads to a simple and efficient hardware implementation [15]. As can be seen from the figure, zeroes are added in multiple places to maintain the structure of the multiplier. Instead of adding zeros these could be used to perform an extra addition during the multiplication, resulting

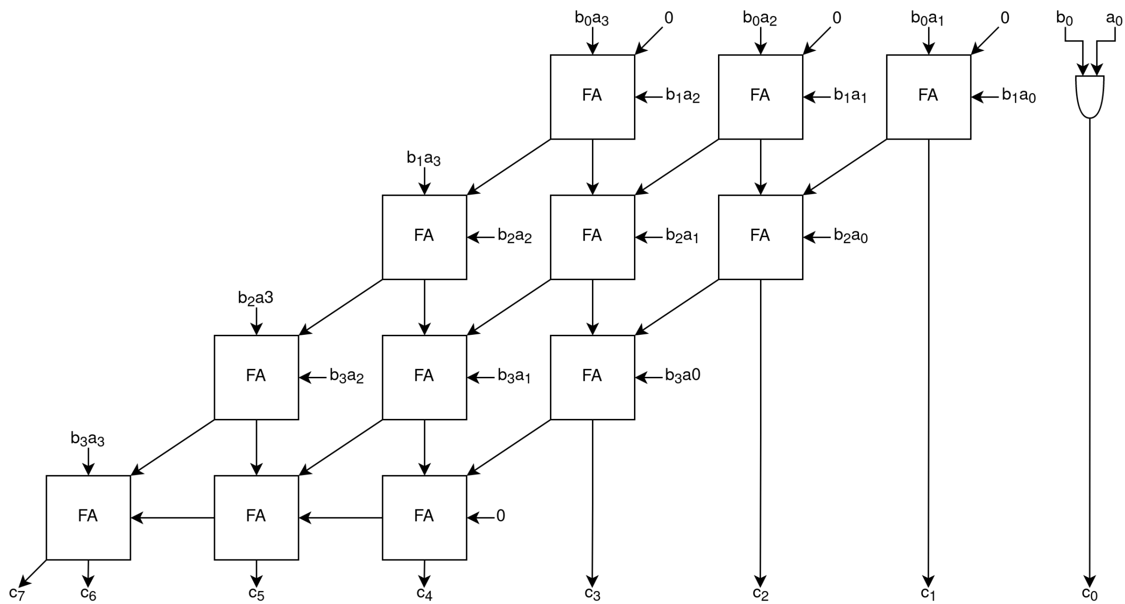



Figure 2.13: 4 bit array multiplier.

in a fused multiply-add operation. If this addition is not required and it is acceptable to reduce the regularity of the multiplier, the full adder cells that have one of its inputs being zero in Fig. 2.13 can also be replaced by half adders to reduce the hardware utilization.
The last row of full adders are basically a ripple carry adder, to compute the final result of the partial product addition. By replacing this adder with a faster type, the delay of the multiplier can be reduced. The width of the result has been doubled compared to the number of input bits of each operand and how this is handled depends on the application: the data path may be wide enough to accommodate the extra bits, or it may need to be shortened. If the application uses modular arithmetic, the MSBs can simply be dropped.

#### 2.2.1.4 Fixed Point Computation

Computation with fixed point numbers is quite similar to integer computation, but care must be taken to align the decimal points of the operands [14]. Addition, as well as subtraction for signed numbers, can be handled by the same hardware as is used for integers: fractional addition is the same as integer addition, and carry propagation over the binary point still gives the correct result. An example of addition is shown in Fig. 2.14a.
Multiplication and division, however, require an extra shift operation to correct the location of the binary point after computation, as is shown in Fig. 2.14b for multiplication: after the computation, the precision of the result has been doubled. In order to store the number in the original format, the result needs to be shifted to the right by the number of fraction bits that are used in the representation. If any of the bits that are

shifted out of the number are 1 accuracy of the result is lost, which can be minimized by using a rounding algorithm as described in Section 2.2.2. It is also possible not to perform this shift, if the datapath after the operation allows for the extra precision of the number. This is done for exact accumulation, for example.
The fact that fixed point arithmetic is so similar to integer arithmetic makes that a fixed point number system can be implemented entirely in software, making use of integer hardware. This saves the need to implement separate hardware for fixed point computation.

```
 01.110 (=1.75_10)
 00.011 (=0.375_10)   +
 ------------------------
 10.001 (=2.125_10)
```

(a) Fixed point addition.

```
     01.110 (=1.75_10)
     10.111 (=2.875_10)   x
     ----------------------
       . 01110
       .01110
      0.1110
     00.000
    011.10                  +
   -------------------------
   0101.000010 (=5.03125_10)
```

(b) Fixed point multiplication.

Figure 2.14: Fixed point arithmetic with example values, with 2 integer bits and 3 fraction bits.

### 2.2.2 Rounding

As round to nearest, ties to even is the default rounding mode for IEEE floating points and the only one that should be used with the posit number system, that rounding scheme is discussed here. Rounding can be required after an integer multiplication as was already shown, but can also be required in other cases like floating point addition. The principle to rounding a number remains the same.
Assume a 4 bit fixed point number with 3 fraction bits: $i_0.f_{-1}f_{-2}f_{-3}$. After multiplication the number will have the following configuration: $i_1i_0.f_{-1}f_{-2}f_{-3}f_{-4}f_{-5}f_{-6}$, where $i_1$ can be used as an overflow detection. When using the word "result" in the following discussion, the result after rounding is meant, again containing the same number of fraction bits as the input operands.
The fraction bits $f_{-4}f_{-5}f_{-6}$ can not be represented in the original number representation, and as such the multiplication result needs to be rounded. For proper rounding, the value of the bits that can not be represented needs to be determined: if their value is less than $ulp/2$, the result needs to be rounded down and the bits that do not fit can be truncated. If their value is larger than $ulp/2$, the number needs to be rounded up which requires incrementing the result. A tie occurs when the bits that can not fit in the result have a value exactly equal to $ulp/2$: it is exactly in between two representable values. In case of a tie, the result must be rounded to an even number: the LSB of the result must be 0. Achieving this may either require no action or incrementing the

result, depending on the LSB of the result. Selectively incrementing the result can be achieved by adding the least significant result bit to the result in case of a tie.
The most significant bit that can not be represented in the final result has a value of $ulp/2$ and therefore plays an important role in rounding. This bit is often called the round bit for that reason. In the example above, the round bit is $f_{-4}$: if it is unset, the result needs to be rounded down and the fraction bits that can not be represented can be truncated. If it is set, a tie needs to be detected: this occurs when all the bits less significant than the round bit are zero. This can be determined using an OR operation: if the result of the OR of all of the bits after the round bit is zero, it is a tie. Otherwise, the result needs to be rounded up by incrementing it. The result of this OR operation is called the sticky bit and can often be obtained dynamically during computation, instead of in a single step at the end of the computation: this reduces the latency of the rounding. For the example, the sticky bit is equal to $f_{-5}\ OR\ f_{-6}$.

### 2.2.3 Floating Point

Floating point operations are more complex than integer and fixed point operations because the exponent and fraction fields need to be handled separately. In this section the basic algorithms for floating point addition and multiplication are discussed, as well as the hardware structure for multiplication. The many optimizations that exist to meet certain design goals are not explored here. Similarly, the additional complexity of supporting subnormal numbers is not explained. For the computations on the exponent and fraction fields integer hardware is being used, with the direct result that floating point hardware will be larger and more complex than integer hardware.

#### 2.2.3.1 Addition

In this section the algorithm that can be used for floating point addition is explained, and summarized in Algorithm 1. For the correct addition of two floating point numbers the significands need to be aligned such that their exponents have the same value. This can be achieved by shifting the operand with the smaller exponent to the right. A decimal example will be used to make this more clear:

$$9.943 * 10^3 + 7.435 * 10^1 = 9.943 * 10^3 + 0.07435 * 10^3 = 10.01735 * 10^3$$

What also becomes clear from this example is that the result of the addition is not necessarily normalized: in this case there are 2 digits before the decimal point. For the result to be correctly represented as an IEEE float again, it should be normalized. In case of subtraction, the first non-zero digit can also be after the decimal point. When subtracting numbers of similar magnitude, specifically numbers with exponents that differ not more than 1 in value, the first set bit can be many bits after the binary point. This means the addition of the significands may need to be normalized by either a right shift of 1 bit, or a left shift that can be larger. The exponent also needs to be adjusted according to the required shift, and should be checked for overflow and underflow as well: it might not be possible to represent the exponent of the result in the given number representation. For the previous example, the normalized result is $1.001735 * 10^4$.

The significand of this result consists of more digits than the input operands, so it needs to be rounded for the significand to comply to the number system parameters. Rounding was discussed in Section 2.2.2, and can mostly be used for rounding the addition result. However, because left shifting the significand may be required for normalization, an extra bit for rounding needs to be used: the guard bit. The guard bit is the most significant rounding bit, followed by the round and sticky bit. If the addition result needs to be left shifted for normalization the guard bit is shifted in the LSB position, and the round and sticky bits can be used for correct rounding. A single extra bit is sufficient to ensure proper rounding, even though a larger left shift may be required for normalization: this larger shift is only required for input exponents with a difference in value of at most 1, in which case only a single bit right shift is required. This means no more than 1 bit is shifted out of the representable fraction field in that case.

Notice how the rounding of the significand may cause the result not to be normalized again, requiring another normalization. This normalization results in the significand to be a digit too long, requiring a second rounding before the output is finally ready. The addition algorithm is summarized in Algorithm 1.

For this research the addition of floating point numbers is not actually used, so the exact details on how to implement the hardware will not be explained. However, the general steps in the algorithm are quite similar to floating point multiplication and the discussion about the multiplication hardware should give an idea about how the addition can be implemented as well.

**Algorithm 1** Algorithm for floating point addition.

1: Compare exponents of the two input numbers,
2: Shift the significand of the number with the smaller exponent to the right until the exponents of the numbers match,
3: Add the significands,
4: Normalize the sum if required: either a right shift by 1 digit, or a left shift of possibly multiple bits,
5: Adjust the exponent of the result according to the normalization shift,
6: Check the exponent for overflow and underflow, trigger an exception if this is necessary,
7: Round the significand to the required number of bits,
8: Check if the result is still normalized after rounding. If not, repeat from step 4.

#### 2.2.3.2 Multiplication

Before looking at the hardware for floating point multiplication, it is again important to understand the algorithm. The same decimal example will be used to help explain it:

$$9.943 * 10^3 * 7.435 * 10^1 = 9.943 * 7.435 * 10^{3+1} = 73.926205 * 10^4$$

As can be seen from this example the exponents can just be added, and the significands multiplied. For both of these operations integer hardware is used. What is important to note here is that floats use a biased exponent, as discussed in Section 2.1.2. After

the addition of the exponents the bias will be in the result twice, and needs to be compensated by subtracting the bias: $(x + bias) + (y + bias) - bias = x + y + bias$. The significands do not need to be aligned as is the case for addition, so their multiplication can be handled in parallel of the exponent computation. As seen in Section 2.2.1, an integer multiplier has a longer delay than an integer adder. This means the delay of compensating the bias is hidden from the critical path of the floating point multiplier: the multiplication of the significands will take longer.
As for addition, the result of significand multiplication is not necessarily normalized. In this case however, normalization can only require a single digit right shift: the significands are in the range $[1, 2)$, so their product is in the range $[1, 4)$ in the binary case. If normalization is required the exponent needs to be incremented, before it is checked for overflow and underflow. Then, the significand needs to be rounded to the appropriate number of bits, after which another normalization may be required.
Finally, the sign of the result needs to be set. This can be handled separately from the other computations. The sign will be positive if the signs of the input operands are the same and negative otherwise. This can be computed using a single XOR gate. The algorithm is summarized in Algorithm 2.

**Algorithm 2** Algorithm for floating point multiplication.

1: Add the exponents and subtract the bias value,
2: Simultaneously, multiply the significands,
3: Normalize the significand if required: shift to the right by 1 digit and adjust the exponent,
4: Check the exponent for overflow and underflow, trigger an exception if this is necessary,
5: Round the significand to the required number of bits,
6: Check if the result is still normalized after rounding. If not, repeat from step 3.
7: Set the sign of the result to the XOR of the input signs.

Using this algorithm the hardware for the floating point multiplier can be designed. The high level design can be seen in Fig. 2.15, where the addition, subtraction, and multiplier are integer components. An incrementer is a dedicated adder where the input is increased by 1, which can be smaller in hardware utilization than a regular adder. The input from the normalization is used to determine whether the exponent should be incremented or not. To reduce the delay, it is also possible to precompute the incremented exponent and use the signal from normalization to select either the incremented exponent or the exponent that has not been incremented.

Rounding can be done using the round and sticky bits as discussed in Section 2.2.2, the guard bit that is used during addition is not required because the result is never left shifted. Instead of performing rounding as a separate step, it can also be incorporated into the multiplication [15]. As seen in Section 2.2.1.3, the least significant bits of the result are available first. These are also the bits that need to be rounded, so instead of waiting for the entire multiplication to be finished they can already be rounded. Because a normalization shift may still be required, the final rounding step only becomes known after the entire significand multiplication is done.

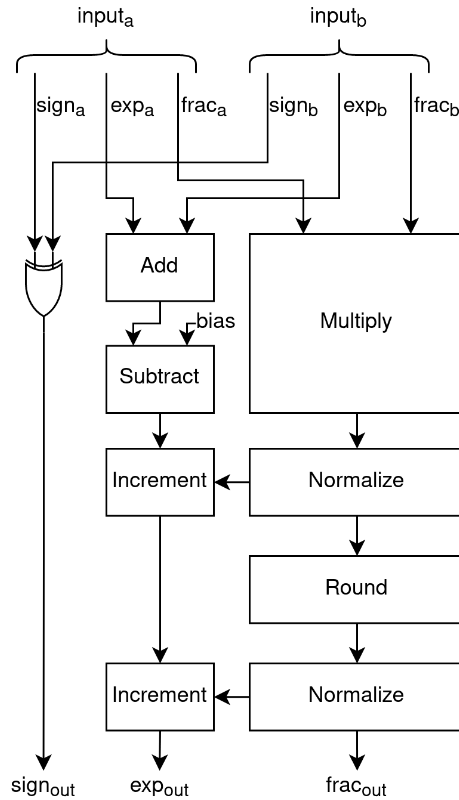



Figure 2.15: Floating point multiplier.

### 2.2.4 Posits

Calculation with posits is more complicated than with floating point, primarily because of the variable length of each of the fields. However, the posit intermediate format (PIF) is really quite similar to floating point: it is basically an exponent and a fraction that determine the value of the number, as discussed in Section 2.1.3.3. The first step in computations is to decode the posit number into the intermediate representation. After the computation is done, the result can again be encoded into the standard posit format.

#### 2.2.4.1 Decoding

The most important field to decode posit numbers is the regime, since it is run length encoded and with that also determines the position and length of the exponent and fraction fields. This also means that the exponent and fraction can only be handled after the value of the regime is determined, rather than decoding all the fields in parallel. The schematic of a decoder can be seen in Fig. 2.16, and is explained in this section. Posit numbers adhere to the 2's complement standard [11], meaning the 2's complement should be taken if the sign is negative. This is done to prevent a representation for

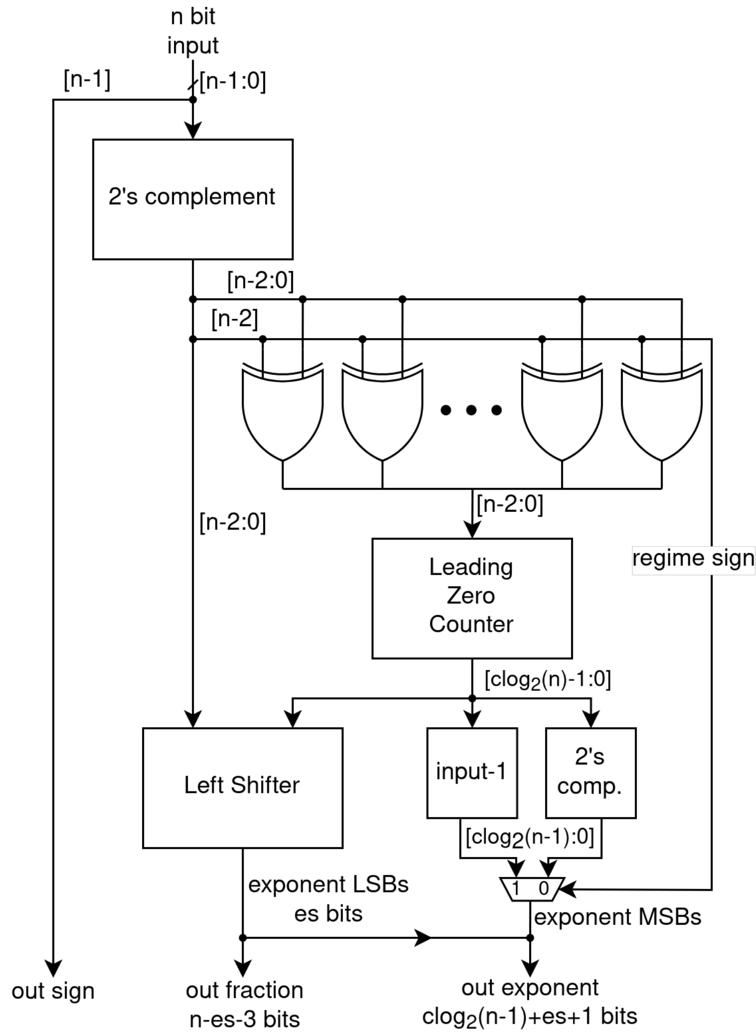



Figure 2.16: Posit to PIF decoder, where clog is the ceiling of the log: $\lceil log_2(x) \rceil$.

negative zero as well as the complications that come with having two representations for the same real value. The 2's complement can be calculated by using a XOR gate on each input bit with the sign, and an adder to add the sign to the ULP.
The next step is to determine the length of the regime field. It has a minimum length of two bits, and a maximum length of $N-1$ bits. The first bit after the sign determines whether the regime is a sequence of unset bits terminated by a set bit, or the other way around. Conceptually the easiest method to determine the regime value is to use both a leading one counter (LOC) and a leading zero counter (LZC), and use the result of one of them according to the first bit of the regime. However, these modules are quite expensive in area so there is an alternative that is cheaper to implement, as also shown in Fig. 2.16.
Instead of using a separate LOC, a single LZC can be used that counts the number of

zeros in sequence at the most significant end of the input bits. If the first bit of the regime is set, all the bits are negated before being input to the LZC, such that the first bit is again zero. This negation is selectively done using the XOR gates in Fig. 2.16 and the first regime bit. The final value of the regime is then determined using (2.21), where $ZC$ is the zero count. Because the regime can be negative, it requires an extra bit compared to the zero count.
Note that the bit length of $ZC$ is indicated in the figure to be $\lceil log_2(N) \rceil$, while the final length of the regime has a bit length of $\lceil log_2(N-1)+1 \rceil$. This is because the value of $ZC$ can only be $N$ in case it is all-1's: the all-0's regime indicates either zero or NaR and is detected separately. In case the regime starts with a set bit, a 1 is subtracted from the zero count value. After this subtraction the its maximum value is $N-1$ which always fits in $\lceil log_2(N-1) \rceil$ bits, but an extra bit is added to represent the sign. This number of bits also corresponds to the value found in Section 2.1.3.3.

$$regime = \begin{cases} ZC - 1, & \text{if first bit set (1)} \\ -ZC, & \text{if first bit unset (0)} \end{cases} \tag{2.21}$$

Now that the regime is known, it can be shifted out of the 2's complemented input by shifting it to the left. The exact number of bits to shift is slightly flexible, since the minimum length of the regime is known to be two bits. Say that the regime is 0001, in which case the zero count is 3. If the input is shifted by 3 bits, the first bit of the output needs to be skipped since it is the set bit that is still part of the regime. Instead, the input could also be shifted by 4 bits, in which case the exponent bits are in the most significant positions. The third option is to shift the input only 2 bits, and skip the 2 most significant bits when determining the exponent and fraction bits.
Which of these options is the best choice depends on the implementation trade-offs, and the way the other components are implemented. Since the hardware to compute $ZC-1$ is already present to compute the final regime value, this seems like a valid option to use for the shift as well. It has the additional benefit that the left shifter can be smaller because it has less input bits. Dependent on which shift is chosen, the exponent bits can be determined, which are followed by the fraction bits. By shifting in zeros from the right during the left shift, it is ensured that exponent and fraction bits that are not present in the input number are interpreted as being zero.
Generally speaking the exponent in PIF is biased, just like the exponent in floats and for the same reasons: easy zero detection and magnitude comparison. This means the exponent should still be biased after the circuit shown in Fig. 2.16, implying an addition to the exponent.

**Leading Zero Counter** There are different ways to implement a leading zero counter (LZC) [21], but a basic circuit is briefly discussed here for completeness. Optimizing the implementation towards a certain design goal can be important to achieve maximum performance in a specific metric. An often used method is to make use of a tree structure, as described in [27]. The idea is to design a small module that counts the number of leading zeros for a small input vector of for example 2 or 4 bits. A truth table for a 4 bit input module is given in Table 2.8, where position indicates the position of the first set bit, which is equivalent to the leading zero count, and valid indicates

whether or not the first set bit has already been encountered. Two of these modules can then be combined to count the leading zeros for an 8 bit input vector, and so on. This is shown in Fig. 2.17. This approach results in a modular and structured design.

Table 2.8: Truth table for a 4 input leading zero counter.

| Input | Position | Valid |
|---|---|---|
| 1XXX | 00 | 1 |
| 01XX | 01 | 1 |
| 001X | 10 | 1 |
| 0001 | 11 | 1 |
| 0000 | XX | 0 |

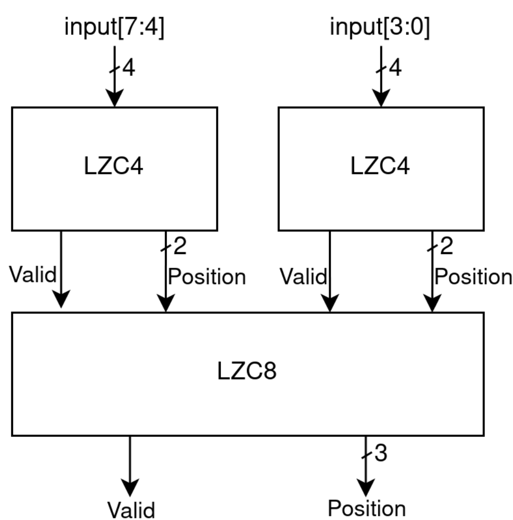



Figure 2.17: 8 bit input leading zero counter using 2 modules with 4 bit inputs.

**Barrel Shifter** A constant shift can be achieved in hardware by rewiring the bits to their shifted position. In the decoder however, the shift amount depends on the length of the regime and is therefore different for each number that is being decoded. This means a variable shift is required, which requires more hardware than just a rewiring. A common design for such a shifter is called a barrel shifter, and is composed of 2 input multiplexers in multiple stages [28]. An example 8 bit input, three stage barrel shifter is shown in Fig. 2.18. The number of input bits can be doubled by adding another stage.

#### 2.2.4.2 Addition & Multiplication

The PIF is similar to floating point in that it has an exponent and a fraction. As such, the basic hardware to compute addition and multiplication is similar to floating point

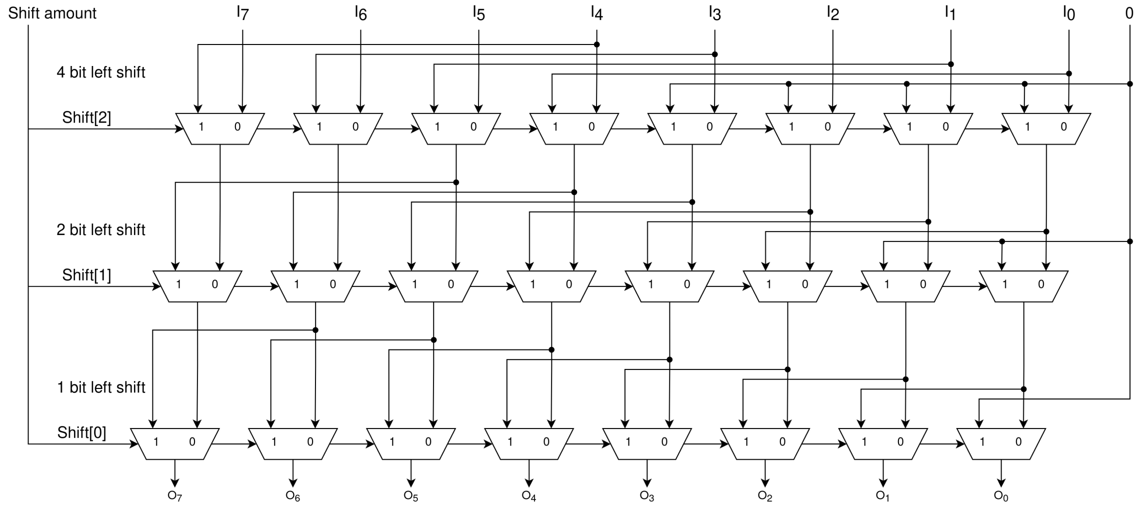



Figure 2.18: An 8 bit barrel shifter design.

hardware for posits, with the exception that posits first need to be decoded before the computation. This decoding adds delay and area overhead to the computation. Of course, the field lengths of the PIF are different to the field lengths of floating point numbers, so the width of the computational elements need to be adjusted for posit computation.
Of course, the exception cases for posit and float are quite different: in posit there are only two bit sequences that are interpreted differently than the rest: zero and NaR. This means that exception handling is simpler for posit than it is for floating point. The exact differences between posit and floating point computation are explained and compared in more detail in Section 2.2.5. Nevertheless, the algorithms and hardware to compute addition and multiplication is very similar.
After the computation using posits is done, the PIF result can be encoded back to regular posit encoding. This encoder is described in Section 2.2.4.3. When using the quire for exact accumulation though, it is not necessary to perform any normalization, rounding or encoding on the results of addition and multiplication. All of the computed bits need to be stored to ensure the accumulation can happen exactly, with only a single rounding after the entire computation is done. How this is done is explained in Section 5.2 in more detail. Essentially, the multiplied significands are shifted to be aligned with the fixed point quire value according to the value of the exponent. Then they can be added to the quire value using a large adder.

#### 2.2.4.3 Encoding

After the computation, the result is still in the PIF. In order to store the result as a posit, which requires less memory and bandwidth than the intermediate representation, the number needs to be encoded again. The schematic of an encoder can be seen in Fig. 2.19, and is explained in this section. The assumption in the figure is that the bias that is used for the exponent representation in the PIF is already subtracted again

before the input.

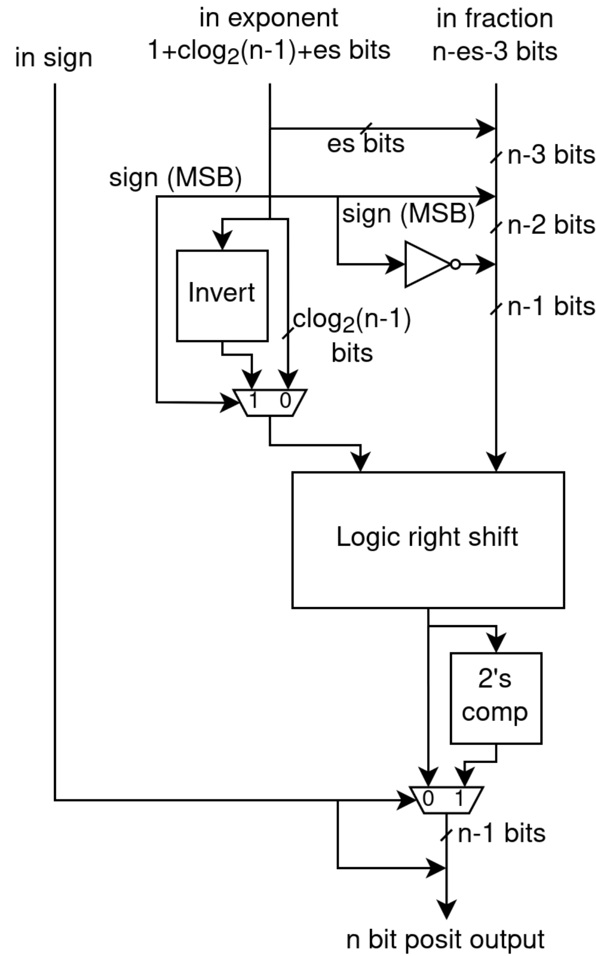



Figure 2.19: PIF to posit encoder.

The first step is to prepend the exponent bits to the fraction bits. Then, either 01 or 10 is prepended to the exponent and fraction bits, depending on the sign of the regime being negative or positive respectively. These form the last two bits of the final regime. This is achieved in Fig. 2.19 by prepending the regime sign to the exponent and fraction, and then prepending the inverse of the regime sign to this value. Adding these two bits as a separate step simplifies encoding the rest of the regime [29].
If the regime is positive it starts with a set bit, and its value $k$ is equal to the number of consecutive set bits minus 1. This means a logic right shift equal to the magnitude of the regime can be used to correctly extend positive regimes, as the first set bit has already been added. A logic right shift implements sign extension, meaning it will duplicate the most significant bit and prepend it.
If the regime is negative it starts with an unset bit, and its value $k$ is equal to the num-

ber of consecutive unset bits. Because the first unset bit has already been prepended, a logic right shift of $|k|-1$ is required. As the regime is represented in 2's complement representation, this value can be cheaply computed by bitwise inversion in case the regime is negative: say the regime value is $-3$. $|-3|-1=2$ unset bits need to be prepended to obtain the correct regime for the encoded posit number. In 2's complement representation, $-3_{10}=101_{2's\ complement}$. Performing a bitwise inversion the right shift will be $010_2=2_{10}$ bits, where the subscripts indicate the base of the number.
Finally, posits are represented in 2's complement representation. This is computed using the input sign, and the sign prepended to complete the encoding from PIF to posit.

### 2.2.5 Hardware Comparison

In this section a comparison is made between the different hardware that is required to compute with the different number systems.
From the previous description of the hardware should be obvious that integer, and with that also fixed point, computation is the easiest. It can also be implemented using the smallest hardware: in fact, integer addition and multiplication are both required operations to implement floating point and posit arithmetic units. This also means that fixed point computation is faster compared to the other number systems. On top of that, fixed point hardware is also simplest, meaning there is a smaller possibility for implementation errors in the hardware. There are examples where floating point hardware that computed incorrect results for some corner case made it into production [30], which is difficult and expensive to correct. Of course the downside of the fixed point number system is the limited accuracy and dynamic range, offering a clear design trade-off.
If either the dynamic range or accuracy of the fixed point number system is insufficient for an application, one of the more complicated number systems can be used. Floating point hardware has been around for a long time and is readily available: it has become the standard for real number computation. Many different hardware designs exist that focus on high speed, low energy, or small hardware area. This makes it easy to use floating point number computation in a design, without having to worry about the implementation details.
A downside of the IEEE 754 floating point number system is that there are relatively many exceptions, as well as different representations for the same thing. Each of the different exceptions needs to be detected and handled by the hardware. Signed zero and infinity do not follow the general rule for interpreting the value of a number, so these also fall into this category along with NaNs. Detecting each of the different cases can be quite expensive in terms of hardware area. Handling them during computation can also be complicated: when two NaNs are checked for equality the result should be *false*, even if the bit patterns are exactly identical. This is just an example, a lot of specific rules exist regarding the results of computation using NaNs, signed zero and signed infinity [10]. Correctly handling subnormal values can also come at a significant cost and is sometimes not supported by hardware to make it simpler and reduce the hardware area, reducing the accuracy of the smallest magnitude numbers [21]. Of course, this is a trade-off that depends on the application of the hardware.

The recent introduction of the posit number system means that high effort hardware designs are relatively sparse, making it more difficult to use them in a design. While posit and floating point hardware is similar in that they both use an exponent and a fraction, posits first need to be decoded into fixed size fields before computation can start. Even though extra hardware is required to decode posits, the exception detection and handling is a lot simpler than for floats and there are no subnormal numbers: only zero and NaR are interpreted differently, and can be easily detected using mostly the same hardware. It has been found that the hardware to compute with posits and floats is quite similar in area for the same precision, as also discussed in the next section. For some applications however, smaller precision posits can be used to achieve similar results to floats, as discussed in Section 3.3. On the other hand, there are also applications where the use of posits is not so suitable, and floating point offers better performance [24].
A downside of the posit hardware is that the hardware needs to use an extended internal precision in order to accommodate the maximum length of the exponent and fraction fields. To illustrate this with an example, consider a (32, 2)-posit number. As described in Section PIF, the maximum exponent width is $1+es+\lceil log_2(N-1)\rceil = 8$ bits and the maximum fraction width is $N-(3+es) = 27$ bits. While the exponent and fraction can never be this wide for the same 32 bit number, the hardware needs to be able to calculate with the longest possible values for each of the fields. Since these components need to be wider, they will also consume more area and introduce more delay into the computation.

## 2.3 Related Works: Posit Hardware

Here some of the hardware designs that have been presented since the recent introduction of the posit number system are discussed. Jaiswal and So [31] were one of the first to present posit hardware, in the form of an arithmetic generator for FPGA implementation. The design had parameterized precision and exponent size, such that it can be used for any posit configuration. Designs for converting both IEEE float to posit, and posit to float are proposed, along with an adder/subtractor and multiplier. Similarly, Podobas and Matsuoka [29], propose hardware designs for FPGA implementation for decoding, addition/subtraction, multiplication, and encoding of posit numbers. They compare their posit adder/subtractor and multiplier to two different floating point intellectual property (IP) modules by Intel and FloPoCo. For both the adder and multiplier, the posit hardware operates at only slightly lower frequencies at more than 500 MHz for most of the bit widths. To achieve this the design of the posit multiplier is heavily pipelined, with between 33 and 38 stages. The posit multiplier also consumes about 6× more area than the floating point hardware, but does use only 1 DSP block compared to 2 for floats. While these results are not looking too promising for posit hardware, the authors also point out that this is a first design that has a lot of room for improvement to reduce area overhead: fine tuning and reducing the number of pipelining stages should result in a significant area reduction. Yet these hardware designs still outperform the software emulation of the posit number system by several orders of magnitude on three different linear algebra functions.

Similar results are reported in [32] by Chen et al., that implement a matrix-multiply unit for posit. Exact accumulation using the quire is implemented, splitting the quire in multiple blocks and using a carry-save format for faster carry resolution during computation. The module for (32,2)-posit MAC computation is compared to a single precision floating point multiply-adder as implemented by Vivado IP core. It operates at 200 MHz, compared to 300 MHz for the float hardware, and uses about 3× as much LUT modules in the FPGA implementation. It does, however, use less flip-flops, about 80% of the ones used by the float implementation. Again, it is reported to be about 1000× faster than posit implementation in software.
In [33], area utilization figures for posit adder and multiplier hardware are presented that are comparable to both their single and half precision floating point counterparts. The datapath delay is slightly increased compared to the floating point hardware. Chaurasiya et al. emphasize that smaller precision posits can be used to obtain performance similar to floats, due to the larger dynamic range and accuracy of posits. This reduces the costs of posit, potentially making them more attractive for certain applications.
In [34], a MAC generator for posits is proposed targeting DNN applications, that performs the accumulation exactly. The generator is written in C for high-level synthesis (HLS), and uses a parameterized bit width and exponent size. It also offers the option to generate combinational hardware or with 5 pipeline stages to increase operation frequency. For 8 bit, 16 bit, and 32 bit floating point MAC units are also implemented to compare the posit hardware to. The posit hardware is reported to have worse performance in terms of area, delay, and power consumption for the same precision: in the case of 8 bits posit consumes 28% more area, has a 67% longer delay, and uses 15% more power. However, Zhang et al. argue that for the deep learning applications they target, 8 bit posits might be sufficient instead of 16 bit floats. This would still result in efficient hardware designs.
PACoGen, developed by Jaiswal and So [12], provides detailed, pipelined computational flows for posit hardware and is the first to discuss division. Compared to the hardware designs described in [33] these new designs are reported to be slightly larger, but can run at an increased clock frequency. They are also compared to their IEEE compliant counterparts and show comparable performance, the exact comparison depending on whether the floating point hardware supports subnormal numbers and which design trade-offs have been made in their design. This is attributed to the fact that posits need to be decoded and encoded before and after computation, but the core arithmetic is simpler as there are no subnormals and a lot less exceptions that need to be detected and handled.
In [17], Uguen et al. present a hardware design for computation using the quire for exact accumulation. It is compared to Kulisch accumulators for IEEE floats, and shown to be very similar in terms of area cost and performance.
In [35], Sarkar et al. propose a hardware implementation of an adder and multiplier with reconfigurable exponent size to dynamically make the trade-off between dynamic range and accuracy, specifically targeting signal processing applications. To achieve this, the decoding is done using content addressable memory. They report significant increase in signal-to-noise ratios compared to floating point of more than 25dB. The

performance gain strongly depends on the chosen exponent size, highlighting the benefits of making that configurable after implementation. The optimal exponent size is also different for their two sample computations. A similar idea is proposed in [36] implementing a MAC unit. However, instead of using a content addressable memory, several shifters are introduced in the design to correctly compute with different es values.
Shang and Ko focus on the posit multiplier in [37]. As the number of fraction bits is variable in the posit number system, the entire width of the multiplier is not always required. In order to lower the power consumption it is decomposed into groups, such that groups that are not required for the computation can be disabled. This technique reduces power by more than 20% for 16 and 32 bit posits, at very minimal cost in terms of area and delay.
All this work shows that there is a lot of interest in developing hardware for the posit number system, allowing to discover its full potential in a range of different applications. The general trend is also that the more high effort posit designs offer comparable performance to their IEEE floating point counterparts. More work on posit hardware exists that has not been discussed here, some examples include [38, 39, 40, 41] if the reader is interested. Work that uses posit number system for neural network applications is discussed in more detail in Section 3.3.

# 3 Deep Neural Networks

Machine learning is a subset of artificial intelligence (AI) that uses algorithms that build a model based on examples, often called the training data. This enables the algorithms to make decisions or predictions about new data without explicit programming concerning these decisions or predictions. A deep convolutional neural network (DCNN), often called deep neural network (DNN) for brevity, is a specific machine learning (ML) algorithm that uses an artificial neural network (ANN) with multiple convolutional layers. The fact that the networks consist of multiple layers is why they are called deep. A network is called convolutional if at least one of the layers performs a convolution operation. Convolution is an attractive operation to use as it is able to exploit spatial or temporal correlation in the data.
CNNs do not only exist of convolution operations, the other components are discussed in Section 3.1.2 along with a more detailed explanation of convolution. Before that, in Section 3.1.1 two networks are discussed that played a significant role in the development of DNNs. As there are plenty of books and surveys that discuss the history and evolution of DNNs, this is not discussed in detail here [4, 5, 42].
After having explained the basics of DNNs, the hardware that can be used to compute with these algorithms is discussed in Section 3.2. This is also where a comparison is made between the different number systems that have been discussed in Chapter 2 for use in neural network applications. Related work that uses posits for DNN applications is discussed in Section 3.3.

## 3.1 Deep Neural Networks

In many machine learning tasks DNNs have achieved state of the art performance, for example in object detection [43] and image classification [3]. It can be proven that a neural network with only a single hidden layer and a finite number of neurons can approximate any continuous function [44]. However, it turns out to be quite difficult to train such a network efficiently. For this reason, deeper networks are used as these are more easily trainable [4].
In a DNN the first layer accepts the input data and the last layer provides the output. All the layers in between are called hidden layers: their inputs and outputs are not seen from outside the model. The use of multiple layers also enables the extraction of features in the data on different levels, extracting low, mid and high level features of the input data in different layers, where high level features are more abstract and extracted in the final layers of the network. Basically, the algorithm is building more complex representations from simpler representations.

### 3.1.1 Two Network Architectures

In this section two different DNNs are discussed, namely AlexNet [3] and VGG [45]. These networks are chosen as AlexNet popularized the convolutional neural network (CNN) model by achieving record-breaking accuracy in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012 [46]. VGG showed that increasing the network depth can result in an increased accuracy that can be achieved by the network.

#### 3.1.1.1 AlexNet

Since 2010, the ImageNet project hosts an annual competition called the ILSVRC [46]. The goal of this competition is to achieve the highest accuracy in object detection and image classification tasks using machine learning techniques. By taking on the same challenge this also allows researchers to compare their progress and learn from each others approaches. To this end, ImageNet also provides a large data set of 1.2 million images, hand labeled into 1.000 different categories that can be used to train the networks.

AlexNet was the winning entry for the 2012 iteration of ILSVRC, achieving a top-5 error rate of 15.3% on the test data. The second best entry that year achieved an error rate of 26.2%, which is significantly higher. The AlexNet network consist of 5 convolutional layers followed by 3 fully connected layers. The network consists of 650.000 neurons and 60 million parameters.

One of the reasons that it could achieve such high accuracy was that it was trained using an optimized implementation on two GPUs, enabling the training of a larger network. It was found that removing one of the convolutional layers, each of which contains no more than 1% of the models trainable parameters, resulted in inferior performance. This already suggests that the depth of the network is important, but at the time the size was limited by the available hardware to perform the computations.

The architecture of the AlexNet network is shown in Table 3.1, each of its components is explained in Section 3.1.2.

#### 3.1.1.2 VGG

A number of different VGG networks are designed in [45] that use a different number of convolutional layers. By fixing other parameters of the networks, a comparison can be made that evaluates the effect of the network depth on the achievable error rate of the networks. Networks of 11, 13, 16, and 19 layers have been trained and evaluated in [45], each containing the same number of max pooling layers and the three final layers are identical, fully connected layers.

Increasing the depth from 11 to 13 to 16 layers continuously reduced the error rate that was achieved by the network, but saturated for the network using 19 layers. Even so, deeper networks can still be beneficial for larger datasets. VGG19 achieved a final top 5 error rate of 7.1%, compared to 7.2% for VGG16.

The architecture of the VGG16 network is shown in Table 3.2, which consists of 138 million parameters.

Table 3.1: The architecture of the AlexNet network, where the input propagates from the top to the bottom.

| Layer type | Activation function | Input size, kernel size or # of neurons | # of kernels | Stride | Padding |
|---|---|---|---|---|---|
| Input | | 224×224×3 | | | |
| Convolution 1 | ReLU | 11×11 | 96 | 4 | 0 |
| Max. pooling | | 3×3 | | 2 | 0 |
| Convolution 2 | ReLU | 5×5 | 256 | 1 | 2 |
| Max. Pooling | | 3×3 | | 2 | 0 |
| Convolution 3 | ReLU | 3×3 | 384 | 1 | 1 |
| Convolution 4 | ReLU | 3×3 | 384 | 1 | 1 |
| Convolution 5 | ReLU | 3×3 | 256 | 1 | 1 |
| Max. pooling | | 3 | | 2 | 0 |
| Fully connected 1 | ReLU | 4096 neurons | | | |
| Fully connected 2 | ReLU | 4096 neurons | | | |
| Fully connected 3 | | 1000 neurons | | | |

### 3.1.2 Network Components

In this section the different component from which neural networks can be built are discussed briefly. As the goal of this research is not to improve network architectures, the level of detail will not be too large. It is, however, important to understand the basic components of different networks such that the hardware to compute with them can be optimized accordingly.

#### 3.1.2.1 Fully Connected Layer

In a fully connected layer, also called a dense layer, each neuron receives data from every neuron in the previous network layer. This is displayed in Fig. 3.1. Fully connected layers are typically added to the end of DNNs to take care of the final classification of the features that have been extracted from the input data.

**Neuron** In a fully connected layer the output of each neuron depends on the output of all the neurons in the previous layer. How much the input from the neurons in the previous layers contributes to the output depends on the weights of the connections between the different neurons. On top of that, a single bias value is also added that is independent of the output of the previous neurons. Both the biases and the weights are parameters of the model that need to be trained using training data.
The output of a neuron is computed using (3.1), where $f(x)$ is an activation function and the indices are according to Fig. 3.1. The reason the activation function is used is explained in the next section. Because of the use of this activation function, the final output of a neuron is also called its activation. Except for the bias, the input to the activation function is the dot product of the activation and weight vectors for each

Table 3.2: The architecture of the VGG16 network, where the input propagates from the top to the bottom.

| Layer type | Activation function | Input size, kernel size or # of neurons | # of kernels | Stride | Padding |
|---|---|---|---|---|---|
| Input | | 224×224×3 | | | |
| Convolution 1 | ReLU | 3×3 | 64 | 1 | 1 |
| Convolution 2 | ReLU | 3×3 | 64 | 1 | 1 |
| Max. pooling | | 2×2 | | 2 | 0 |
| Convolution 3 | ReLU | 3×3 | 128 | 1 | 1 |
| Convolution 4 | ReLU | 3×3 | 128 | 1 | 1 |
| Max. Pooling | | 2×2 | | 2 | 0 |
| Convolution 5 | ReLU | 3×3 | 256 | 1 | 1 |
| Convolution 6 | ReLU | 3×3 | 256 | 1 | 1 |
| Convolution 7 | ReLU | 3×3 | 256 | 1 | 1 |
| Max. pooling | | 2×2 | | 2 | 0 |
| Convolution 8 | ReLU | 3×3 | 512 | 1 | 1 |
| Convolution 9 | ReLU | 3×3 | 512 | 1 | 1 |
| Convolution 10 | ReLU | 3×3 | 512 | 1 | 1 |
| Max. pooling | | 2×2 | | 2 | 0 |
| Convolution 11 | ReLU | 3×3 | 512 | 1 | 1 |
| Convolution 12 | ReLU | 3×3 | 512 | 1 | 1 |
| Convolution 13 | ReLU | 3×3 | 512 | 1 | 1 |
| Max. pooling | | 2×2 | | 2 | 0 |
| Fully connected 1 | ReLU | 4096 neurons | | | |
| Fully connected 2 | ReLU | 4096 neurons | | | |
| Fully connected 3 | | 1000 neurons | | | |

neuron.

$$n_{1j} = f(bias_{1j} + \sum_{i=0}^{N-1} n_{0i} * w_{ij}) \quad (3.1)$$

**Activation Function** Activation functions are used to introduce non-linearity into the decision making process. In the past, the hyperbolic tangent or the sigmoid function, as in (3.2), were typically used for this [3, 47]. Recently, however, the rectified linear unit (ReLU) function is more popular: $f(x) = max(0, x)$. Krizhevsky et al. demonstrated that networks using the ReLU activation function instead of the hyperbolic tangent can be trained several times faster [3]. The ReLU function is also a lot cheaper to compute.

$$f(x) = \frac{1}{1 + e^{-x}} \quad (3.2)$$

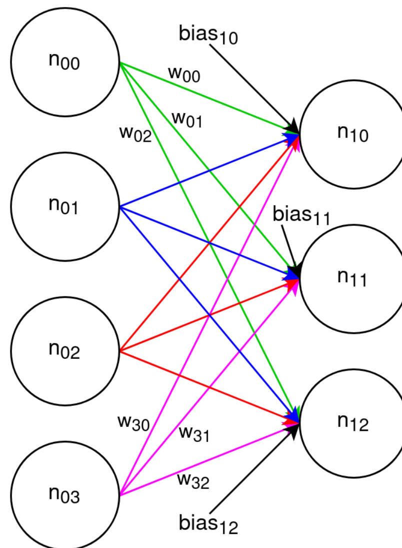



Figure 3.1: An example of a fully connected layer.

#### 3.1.2.2 Convolutional Layer

As opposed to a fully connected layer, neurons in a convolutional layer are only connected to local neurons of the previous layer. An example of a convolutional layer is shown in Fig. 3.2, where the single set of red connections helps indicate the neurons that are connected to the next layer. The number of connections each neuron has to the previous layer determines the number of different weights: this set of weights is often called the kernel. The size of the kernel is also called the receptive field size. Multiple different kernels are often applied to the same layer, to allow for different features to be extracted from the same layer for the same data. This corresponds to the number of kernels as shown in Tables 3.1 & 3.2. The set of kernels for a specific layer is called the filter, though the terms kernel and filter are often used interchangeably.

In a convolutional layer there are less connections between two layers than for a fully connected layer. On top of that, the same kernel is used for each neuron: this is often called weight or parameter sharing. This means that the number of trainable parameters is significantly lower for a convolutional layer than a fully connected layer, as well as the storage requirement for the parameters. As related features in the input data are often spatially local, it is often unnecessary to use a fully connected layer for feature extraction.
The benefit of these kernels that are used for different neurons is that the exact position of a feature in the input data can change position yet still be detected: this is called equivariance to translation: translated features can still be detected [5]. Basically, what this means is that an image classifier can classify images independent of the exact location of the subject of an image: a picture of an apple can be classified whether the apple is located in the top left corner of the input image, the center, or anywhere else.

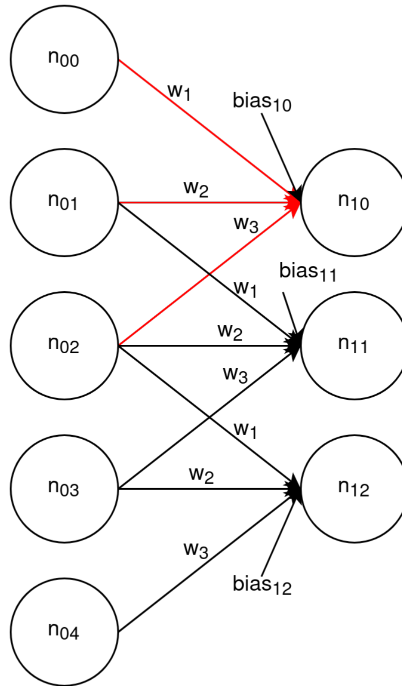



Figure 3.2: An example of a convolutional layer.

**Stride** Besides the receptive field size, another important parameter of a convolutional layer is the stride: it describes by how many neurons the kernel is shifted before being applied again. Fig. 3.2 uses a stride of 1. By using a larger stride the amount of overlap between kernels can be reduced, resulting in a downsampled convolution [5]. With a larger stride features will be extracted less fine grained, with the benefit of a lower computational complexity.

**Padding** Padding can be used at the edges of the input data to increase its size. Without padding, the size of the representation decreases by $kernel\ size - 1$ after every layer. Padding can be used to prevent the output to shrink due to convolution, which is especially a problem for larger receptive field sizes. Typically, zeros are used for padding. The amount of padding relative to the kernel size determines if and how much the output size shrinks compared to the input of a layer.

#### 3.1.2.3 Pooling

Pooling layers are often added after each, or after every few, convolutional layers, as also seen in Tables 3.1 & 3.2. The goal of pooling is to progressively reduce the size of the representation, as more high level features are being represented in the deeper layers. As such, pooling provides a sort of summary of neighbouring neurons. It also introduces some invariance to translation [5]. By reducing the size of the representation, the number of network parameters is reduced, and with that also the computational

complexity and the memory footprint. Pooling is a method of downsampling, where several non-linear functions can be used.
Max pooling is most commonly used, though there are many different functions that can be used as well like average pooling [4]. An example of max pooling with a receptive field size and stride of 2 can be seen in Fig. 3.3: the output of each receptive field is assigned the maximum value that is present in its input. This is helpful in increasing the invariance to translation. As can be seen from the figure, using a receptive field size and stride of 2 reduces the number of outputs by 75%.

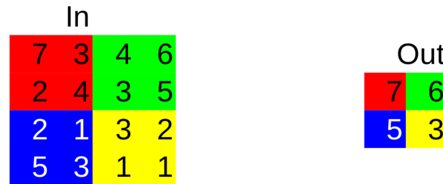



Figure 3.3: An example of the max pooling operation.

## 3.2 Inference Computation

The process of using a DNN to make decision or predictions about new data is called inference. It is preceded by training the network using training data, to teach it to perform its task accurately. As training is a process that is quite different from inference, and it is not the focus of this research, training will not be discussed here.
DNNs that are designed to complete challenging tasks need to be larger than for simpler tasks [5]. As such, the networks are becoming bigger and bigger, to handle more difficult tasks and to increase the accuracy that they can achieve. This means that the inference process also becomes increasingly expensive, as the computational complexity and size of the networks increases. At the same time, it can be interesting to run the inference process on edge devices, like a drone that can follow a target or an autonomous car that needs to detect traffic. In these cases, energy efficiency and a small delay are important to maximize flight time and safety.
Different compute architectures that can be used for DNN inference are discussed in the next section, with the systolic array architecture being explained in Section 3.2.2. In Section 3.2.4 the process of quantization is discussed, which can help reduce the computational complexity and energy consumption of inference computation, with an intermezzo about energy consumption of memory accesses compared to compute in Section 3.2.3.

### 3.2.1 Compute Architectures

The most straightforward way to perform inference computation would be to use the central processing unit (CPU) of a PC. It consists of a few general purpose cores that perform the computation, and can be easily programmed. However, due to the limited number of cores, the inference process can take a long time even for smaller networks.

The computation that is performed most is the vector or matrix multiplication of weights and activations, which can be parallelized for more efficient computation. This is where the use of a graphics processing unit (GPU) can provide a benefit: they consist of many, relatively simple compute cores that can be used for parallel computation. GPUs started out as a piece of hardware specifically optimized to handle graphics in a PC, but have over time become more programmable [48]. GPUs are now a general-purpose, programmable processor consisting of many compute cores, making them very suitable for parallel computation. One of the reasons AlexNet was so successful was that it was trained on two GPUs, allowing a larger network to be trained efficiently [3]. One of the downsides of using general-purpose hardware is that efficient computation is limited to the number systems that have been implemented to compute with in hardware. For GPUs this is typically single precision floating point, which may not be required for inference as discussed in Section 3.2.4. While it is possible to compute with different number systems using software solutions, this will come at a cost in speed [39]. On top of that, GPUs are notorious for being power hungry, which is not ideal for inference on edge devices. Using application specific hardware can be a solution to improve performance: as DNN inference is memory and compute intensive, this has become a popular approach.

#### 3.2.1.1 Application Specific Hardware

To improve performance of a compute system, optimized hardware can be used that is specifically designed for a target application. This is less flexible, but can significantly improve performance compared to using general purpose hardware. There are two options: programmable hardware in the form of a field programmable gate array (FPGA), or implementing an application-specific integrated circuit (ASIC). While ASICs offer the best performance in both speed and energy, developing them is a very expensive process that can in most cases only be justified when producing a large volume of chips: the one-time engineering cost is high, while the production cost of each chip is relatively low. FPGAs typically offer worse performance, but they are more cost efficient than ASICs for smaller volumes: the non-recurring costs are lower, while each unit is typically more expensive than a dedicated chip. The hardware in an FPGA can be configured, after manufacturing, to fulfill the desired behavior. This means that the same chip or development board can be used to implement different hardware structures, making FPGAs very suitable for prototyping and small volume hardware production. Other benefits include a shorter time to market than ASICs, and the ability to reprogram FPGAs after they have been deployed to fix design errors or update systems to new state of the art. In the case of ASICs it is not possible to make corrections after fabrication, making design errors very expensive to correct if that is required.
In general, the desired circuit is implemented in an FPGA using lookup table (LUT) instances in the programmable logic blocks: by programming the desired output of the LUTs for a specific input, the required logic function can be implemented. Programmable logic blocks often also include flip-flops (FFs) to provide sequential logic and memory. Besides the programmable logic blocks, many FPGAs also include dedicated resources that can be used to speed up the implemented design and to save other resources. Some examples include Block RAM (BRAM) to efficiently implement larger

memories, digital signal processing (DSP) units to perform computations like multiplication, and specific bus connections for high speed I/O connections.
The popularity of DNN systems has motivated a lot of research to develop dedicated hardware for its computation. The most important operation for both convolutional and fully connected layers are the MAC operations, which can be easily parallelized to increase throughput. Examples include Google's tensor processing unit (TPU) [49], which is a server scale ASIC designed to accelerate machine learning applications, the DianNao family of accelerators [50], the Eyeriss accelerator [51], and Facebook Zion that allows the integration of accelerators with CPUs for data center scale training [52]. What is notable about these examples is that most use a hardware architecture known as a systolic array for the computation, and use smaller precision numbers than the typical single or double precision floating point that is often used for computational applications. These concepts are elaborated on in Sections 3.2.2 & 3.2.4, respectively.

### 3.2.2 Systolic Arrays

A specific architecture that is a good fit for DNN computation is that of a systolic array (SA): an array of processing elements (PEs) as shown in Fig. 3.4, that is designed or programmed to compute different algorithms. SAs have a simple and regular structure, where each PE is only locally connected to its neighbors. Computation is typically triggered by the arrival of new data, rather than by instructions. Input data is received from neighboring PEs, and passed on in the next clock cycle. Partial results can also be propagated through the array to be used by downstream PEs. This reuse of data significantly lowers the amount of required memory accesses, as only the PEs at the edge of the array communicate input and output data to outside of the SA. As each PE is computing simultaneously and independently, a SA offers high concurrency. While the example given in Fig. 3.4 is 2 dimensional as suggested by the term "array", 1 or multidimensional architectures are also possible. A downside of using a systolic array is that the latency is quite long: the result needs to propagate through the array before it is available at the output. This makes SAs more suitable for applications where throughput is important, rather than short response times.
The computation that is executed in each of the PEs is arbitrary and depends on the application the SA will be used for. The functionality of PEs is not limited to combinational circuits, but can contain local memory as well. For the PEs a trade-off needs to be made: to keep them simple to minimize hardware area and the required control, at the cost of the SA being less flexible. In the case of DNN applications the most important operation for both convolutional and fully connected layers is the multiply-accumulate (MAC), or possibly a fused multiply-add (FMA), to compute the output of each neuron. This operation requires minimal branching and therefore also little control for correct computation. There are multiple strategies to correctly compute the required results, depending on how the data propagates through the array [13].

#### 3.2.2.1 Data Flows

For DNN applications there are four different data flows that are most interesting: output stationary (OS), weight stationary (WS), one called no local reuse (NLR), and

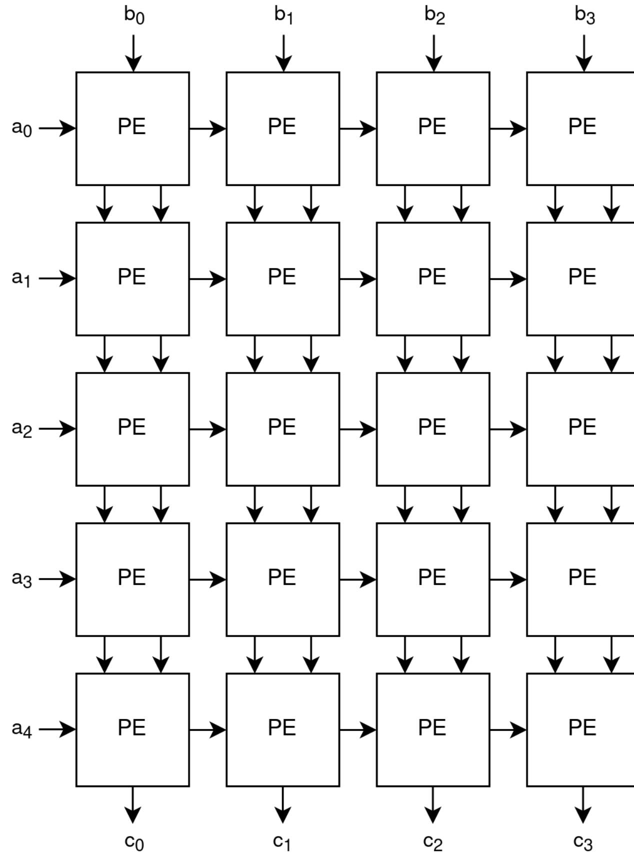



Figure 3.4: Example of a $5 \times 4$ systolic array.

row stationary (RS) [47]. These are by no means all the possible data flows to achieve correct results, but these have some attractive properties that are briefly discussed.
The choice of data flow impacts how different data is being reused, which affects the number of required accesses to different levels of the memory hierarchy. Memory that is closer to the compute is cheaper to access in terms of latency and energy, but is also smaller in size. Reusing data that has already been moved to the lower levels of the memory minimizes the number of required accesses to the more expensive levels of memory. This means that choosing the correct data flow for the application is important to optimize performance and minimize energy consumption.
In the output stationary data flow, the accumulated partial sums for each of the output activations remain stored locally in each of the PEs, while weights and input activations are propagated through the SA. This minimizes the reading and writing of partial sums to memory. After the accumulation is done, the final results need to be propagated out of the array, before the next computation can start. It may also be possible to overlap the output of results with the start of the next computation, to reduce the latency.
In the weight stationary data flow, the weights are loaded and stored in each of the PEs to be reused for as many computations as possible, minimizing the energy consumption

caused by reading weights. The input activations and partial results are propagated through the SA. In order to store the weights, there are two options: either propagate them through the SA, or broadcast the required weights to their corresponding PEs. The second option has a smaller latency for storing the weights in the PEs, but it does require more interconnect: each PE needs to be connected to the edge of the PE. The increased amount of interconnect reduces the compute density and the regularity of the SA and the longer interconnect increases energy consumption. Whether using broadcasting is worth the extra costs depends on the size of the SA and the application it is being used for. The same approach can actually also be used to read the results in an OS dataflow.
In the no local reuse data flow, none of the data is stationary inside the PEs: everything is propagated through the SA. While the local storage in each of the PEs is cheap in terms of energy consumption, it is expensive in terms of area. Instead of having storage in each of the PEs, this area is used for the on-chip buffer memory to maximize its capacity. As the weights, input activations, and partial sums need to be propagated, this will lead to increased traffic to and from the on-chip memory. By maximizing on-chip memory, the off-chip memory bandwidth can be minimized.
In the row stationary data flow, the aim is to maximize the reuse of all different data: the weights, input activations, and partial sums. This is done to minimize the total energy consumption, rather than minimizing it for either the weights or partial sums. The downside is that the mapping of the computation to the hardware structure depends on the architecture of the DNN. This means the hardware needs to be configured for different mappings to optimize the computation of different network layers [51], which complicates both the software work flow and the hardware.
Using the SCALE-Sim systolic accelerator simulator Samajdar et al. showed that even though network architecture and SA dimensions affect which data flow is optimal, there is not a dramatic difference in terms of energy consumption and the required number of execution cycles for different data flows [53]. For this reason, designing a systolic array with a fixed data flow should not cause a significant increase in execution time or energy consumption. However, if an accelerator is designed targeting a single, specific network, it of course makes sense to choose the optimal data flow for that application. To reach this conclusion they have simulated the runtime in cycles and energy consumption for seven different workloads using OS, WS, and input stationary (IS) data flows in square SAs of five different sizes. The IS data flow is similar to WS, but instead of the weights being stationary the input activations are. NLR is not explicitly modeled as it is argued to be any of the other data flows without memory in the PEs.

### 3.2.3 Energy: Memory vs. Compute

As is well known, power has become the limiting factor in increasing the performance of compute platforms: chips generate too much heat to be cooled efficiently [8]. In the era of dark silicon not all available transistors on a chip can be run at maximum performance as this would create more heat than can be dissipated [54]. This means that to improve performance, the energy consumption per operation needs to be reduced to keep the power consumption constant. Parallel computation is more power efficient, which was the major reason for moving to multicore processors to increase performance:

each core can have lower peak performance to increase energy efficiency, yet increasing overall performance by using multiple cores. This is also the reason why dedicated accelerators are being added to compute platforms, for example for handling encryption algorithms: they are more energy efficient for the task they are optimized for, and consume minimal power when they are idle.
In [8], the energy consumption of cache and DRAM accesses are compared to compute operations for a 45nm technology. 64 bit L1 cache accesses are reported to consume roughly 10 pJ of energy, L3 cache 100 pJ, while a DRAM access consumes roughly 1.3-2.6 nJ. 32 bit floating point compute cost roughly 3.7 pJ for multiplication, and 0.9 pJ for addition. Integer compute is even cheaper, at 3.1 pJ and 0.1 pJ for multiplication and addition respectively. The Eyeriss SA [51] shows similar results, where memory access energy consumption is compared to a MAC operation performed in each PE: off chip DRAM access consumes 200× as much energy, on chip buffer access 6×, and inter-PE communication twice as much energy as the MAC computation.
These results show that moving data around on chip, and especially off chip to DRAM, is a lot more energy consuming than the actual computations. This in turn implies that reducing memory access energy consumption can be very important to improve performance of the system. This is one of the reasons the SA architecture is so popular: the reuse of data between the PEs means there are less accesses to the memory hierarchy, reducing energy consumption.

### 3.2.4 Quantization

DNNs are often trained using single precision floating point numbers, as the accuracy of the numbers is important for the backpropagation algorithm that is typically used for training [22]. However, it has been found that smaller precision numbers can often be used for inference without incurring a significant reduction in inference accuracy. The process of converting a number in a certain representation to a number in a different representation with a smaller set of quantization levels is often called quantization. The benefits of doing this are that it reduces the required memory size and bandwidth, as the data that needs to be stored and moved consists of fewer bits. Data that consists of fewer bits also means that either fewer memory accesses will be required as a single access can contain more data points, or memory accesses can contain fewer bits in total. As was discussed in Section 3.2.3, this can have a huge effect on the total energy consumption of the system. Quantization also reduces the computational complexity as the numbers to compute with are smaller, which in turn increases the processing speed that can be achieved while reducing the hardware area.
Of course the benefits of quatization do not come for free: smaller precision numbers either have a smaller dynamic range, accuracy, or both. This can affect the inference accuracy that can be achieved, if the quantization error becomes too large: the error that is introduced by representing a number in fewer bits. Quantization can also cause overflow, if the magnitude of the number is too large to be represented in the new number system. This makes it important to make a proper trade-off between the number precision and the required inference accuracy.
Google introduced the bfloat16 floating point format, which is a truncated version of IEEE single precision numbers: it uses 8 exponent bits and 7 fraction bits, where IEEE

half precision uses 5 and 10 bits, respectively [55]. This retains almost the same dynamic range at the cost of a reduced accuracy, which should be a better fit to DNN applications than half precision floats according to [55]: reportedly, accuracy using bfloat16 is similar to single precision floats for a wide range of deep learning models. It has the added benefit that the multipliers are significantly smaller, as their width depends on the number of fraction bits.

Instead of using floating point numbers, fixed point numbers can be used for inference as well. This has the added benefit that fixed point, or integer, computation is cheaper in terms of latency and energy consumption than floating point computation. 8 bit numbers are often used for inference, yet the possibilities are endless: some have taken it as far as using a single bit to represent weights and activations [56], and different quantization techniques can be applied to the weights and activations [56], or different precision numbers can be used for different layers [57, 58].

In [9], the authors take a model that is trained using floating point numbers and quantize it to use fixed point numbers. To this end they use dynamic fixed point numbers: as the number distribution can be different for weights and activations, as well as for different network layers, the number of integer and fraction bits can be different even for numbers of the same precision. A trade-off needs to be made between overflow error and quantization error: more integer bits reduce the overflow error but less fraction bits remain, which increases the quantization error. Dynamic fixed point allows this trade-off to be made for weights, activations, and each network layer independently. As the weights are static after training, the number of integer bits should accommodate the dynamic range of the parameters. After determining the number of integer and fraction bits, the numbers are quantized using round to nearest. The statistical information of the activations can be gathered by performing the network computations using some representative input data, and storing the activations of each layer.

The results in [9] show that static fixed point inference accuracy start to degrade significantly when using less than 17 bit numbers, for a quantized model based on AlexNet (specifically, CaffeNet) using the ImageNet dataset. The top-1 accuracy for single precision floats is reported to be 56.9%. 8 bit dynamic fixed point achieved an inference accuracy of 55.8%, a degradation of only 1.1%. By performing fine-tuning, the inference accuracy can be further improved to 56.0%. Fine-tuning is a technique where a trained network is trained again for a number of iterations, taking into account the desired quantization in the weight updates to account for the quantization error. This is done to reduce the inference accuracy drop that may be caused by performing inference with the quantized network [59].

Rather than using the same precision for every network layer, the Stripes accelerator allows the precision to be selected per layer [58]. To this end it uses bit-serial computations, instead of bit-parallel computation that is more typically used: instead of computing with all bits of the input operands in one cycle, each cycle computes partial results on 1 bit of each operand and require as many cycles as there are bits in the input numbers to complete the computation. This also allows for an on-the-fly energy and performance trade-off to be made, by changing the precision of the numbers: lower precision numbers have a lower latency and energy consumption, at the cost of being less accurate. For convolutional layers, this approach resulted in an average speedup of

2.26× compared to DaDianNao [50] that is used as reference, while achieving the same inference accuracy. At the same time, Stripes is reported to be 57% more energy efficient. It does have an increased area, being 34% larger than the reference accelerator. Additional performance increase can be achieved if a small inference accuracy degradation is acceptable: with a maximum relative error of 1%, average speedup increases to 2.48× and increased energy efficiency increases to 68%. While these results are promising, it would be hard to achieve similar results using bit-parallel computation, because input operands are typically limited to powers-of-2.

Lin et al. [57] give a strong theoretical background of the effect quantization noise has on the signal to noise ratio in DNN applications. With that the minimal required number of bits per layer is optimized to maintain the inference accuracy, achieving more than 20% model size reduction. A hardware platform to compute with different precision numbers efficiently is not described in the paper.

In [56], Rastegari et al. take it a step further by representing weights using a single bit in Binary-Weight-Networks (BWN), and representing both weights and activations using a single bit in XNOR-Networks. As the weights are represented by a single bit, the network are approximately 32× smaller than than equivalent networks using single precision numbers, making them very suitable to be computed on edge devices. In BWN, the multiplication required for convolution can be approximated by just addition or subtraction as weights are binary, resulting in approximately 2× speedup. In the case of XNOR-Networks, multiplication can even be reduced to XNOR and bit counting operations, hence the name. This results in approximately 58× speedup in CPUs. Compared to single precision AlexNet, BWN incurs only 0.8% top 5 accuracy degradation, XNOR-Net achieves 69.2% top 5 accuracy which is a quite significant degradation of 11%.

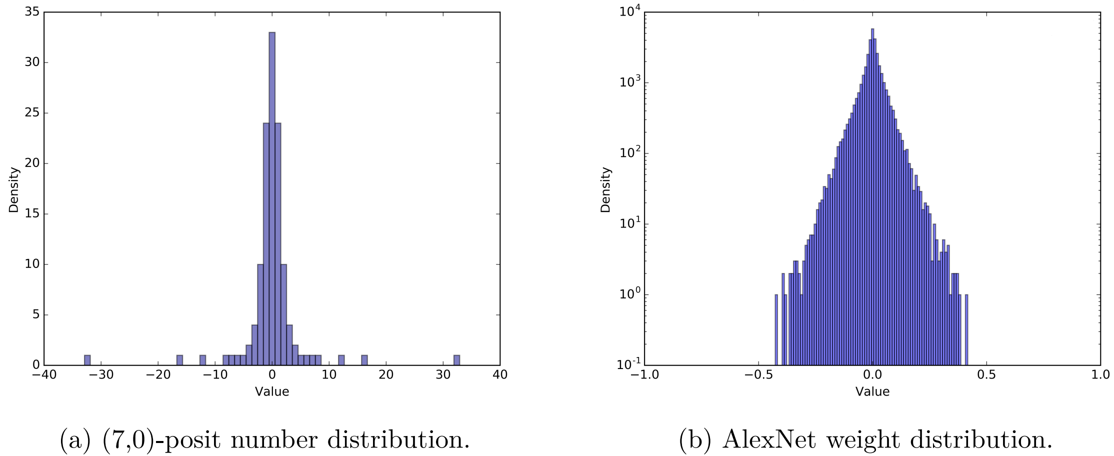
Besides the benefits that have already been mentioned for quantization, it can also act as a regularization in neural networks. Generalization is the capability of a trained network to correctly make predictions about new data [5]. In general, regularization techniques can be applied during training to make sure the network generalizes well to new input data, rather than overfitting to the training data. Examples exist where DNNs perform inference more accurately after quantization, possibly with fine-tuning, than with the baseline number system the network was trained with. This implies that the quantization has helped the network to generalize, acting as a regularization. [60] shows a reduction in test set error rate when using 8 bits for the errors and gradients during training, rather than 16 bits. In [57], the error rate is reduced by 0.2% when using 8 bit fixed point numbers for the activations, compared to single precision floating point. This is after 30 epochs of fine-tuning to account for the quantization though.

The use of the posit number system for DNN applications is discussed in the next section.

## 3.3 Related Works: Posit in DNNs

When the posit number system was introduced, it was soon recognized that it was well suited to be used in DNN applications: it offers both high accuracy and dynamic range, and the precision is flexible. Using low precision fixed point has become standard

practice for DNN inference, yet the small dynamic range is a drawback in using this number system. The number distribution of posits has been shown to be similar to the distribution of weights in neural networks, as shown in Fig. 3.5 [1], making it a very good fit for the application.




(a) (7,0)-posit number distribution.

(b) AlexNet weight distribution.

Figure 3.5: A comparison between the distribution of posit numbers and the weight distribution in a DNN. Figures from [1].

Langroudi et al. [61] were one of the first to use posits for DNN inference: weights are stored in the posit format, and converted to single precision floats for computation on general purpose hardware. Inference accuracy is compared for three different networks and classification tasks, where the reference uses fixed point quantization. With the posit number system 2-4 bits smaller precision can be used, while incurring less than 1% accuracy degradation. This is reported to reduce memory utilization by 23 to 36%. With Deep Positron, Carmichael et al. [1] take it a step further: they implement an exact MAC unit using the quire, and compare it to exact accumulation performed using fixed point and floating point number systems. Their posit and floating point implementation have comparable delay and energy consumption, while fixed point is faster and consumes less energy. The posit hardware does pose a larger area consumption. On three different (low-dimensional) data sets, the 8 bit posit MAC is shown to achieve the same or higher inference accuracy than both 8 bit floating point and fixed point inference. On one of the data sets, it even matches the performance of single precision floating point, and for another it causes only 0.4% accuracy degradation. This work is continued in [62], evaluating the quantization error introduced by each of the number systems in each network layer. For both MNIST and Fashion MNIST, (8,1)-posit matches the performance of single precision floating point. Posit does require more power than the other 8 bit number systems.
In [23], Johnson introduces a logarithmic number system using the same tapering that is used as in the posit number system. Multiplication can be computed by addition thanks to the logarithmic representation, preventing the need for the most power and area consuming component of a MAC unit: the multiplier. The products are then converted to a linear representation and accumulated exactly in a Kulisch accumula-

tor (or quire), though the conversion of the product from log to linear representation is not exact. The ResNet-50 network is used for accuracy testing, on the ImageNet validation set. The 8 bit logarithmic representation introduced in the paper achieves a top 5 accuracy degradation of only 0.20% compared to 32 bit floats. Using 8 bit posits, (8,1)-posit is reported to perform better than exponent sizes 0 or 2, resulting in a top 5 accuracy degradation of 0.19%. The floating point reference is computed using FMA operations rather than MACs, and no retraining is done to account for the quantization.
There is also work focussing on training DNNs using the posit number system, in order to reduce the memory requirements and computational complexity of that process by using lower precision. Examples include [63], that shows networks trained with posits outperform floating point trained networks for both 16 and 32 bit precision. Using 16 bit posit for training results in an accuracy degradation of 1.7% compared to single precision floating point on the Fashion MNIST dataset. In [64], an accuracy degradation of 0.11% is reported for training using 8 bit posit compared to single precision using the ResNet-18 network for ImageNet classification. Results showing less than 1% accuracy degradation for different networks and classification tasks are also presented.

## 3.4 Methodology: Hybrid Hardware

There are two important approaches to increasing the performance of a systolic array: scaling-up and scaling-out [53]. In a scale-up approach a single SA is made bigger by adding more PEs to increase the performance. The scale-out approach uses multiple arrays and divides the compute between them. Which of these approaches results in the smallest runtime and memory utilization depends on the application that is being computed [53]. In the research group where this graduation project is being fulfilled, a systolic array accelerator for DNN inference is being developed for object detection in autonomous vehicles. It uses a scale-out approach, using 9 SA tiles as shown in Fig. 3.6, where each tile is a systolic array of $9 \times 8$ PEs and the selection multiplexers take care of the data flow between the tiles.
It has been found that when quantizing a DNN, the input and output layer are most sensitive to the quantization error that this introduces [57, 65]. In order to minimize the accuracy degradation due to quantization, these layers can be computed with larger precision numbers. One of the benefits of using the scale-out approach is that the tiles do not need to be homogeneous: in this design, the tiles in the first column (tiles #0, #3, and #6) compute using 16 bit fixed point numbers, while the remaining tiles use 8 bit fixed point. The first column of SA tiles can be used to compute the first and last network layers, the rest is used for the hidden layer computation at a reduced computational complexity compared to 16 bit precision. Of course it would be a waste to have the 16 bit hardware idle during hidden layer computation, so to increase the throughput it can be used for hidden layer computation by padding the input data to a length of 16 bits.
From the design of this system, the idea was hatched to replace the 16 bit fixed point tiles with 8 bit posit SA tiles. The posit number system has been shown to be very suitable for the use in DNN systems, as discussed in Section 3.3, which should enable

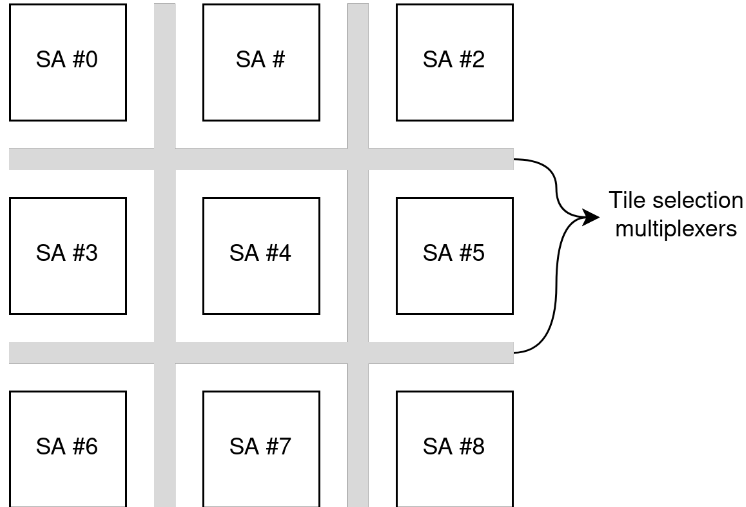



Figure 3.6: 9 tile systolic array design.

the use of lower precision data when using posits. The most important reason to do this is that the reduction in data precision should result in an energy consumption reduction, as memory accesses consume a lot more energy than the actual compute. This was discussed in Section 3.2.3. This is especially helpful for the last network layer, as a lot of the network weights are in the fully connected layers that are typically used for the final layers. The area of the posit hardware will be larger compared to the fixed point tiles that are replaced, but in the era of dark silicon this is probably a worthwhile trade-off to increase energy efficiency. For the hidden layer computations the 8 bit fixed point tiles are maintained, as these have a lower computational complexity and area utilization, and the quantization error in these layers has a smaller effect on the inference accuracy.

Another benefit of using 8 bit data in each of the compute tiles is that the entire memory hierarchy can be optimized for this, rather than also having to account for 16 bit data. This reduces the complexity of the memory accesses, and a smaller memory bandwidth should be sufficient to supply the SA with the required data.

As different data types are being used in the different SA tiles, it is no longer straightforward to perform the hidden layer computations on all of the tiles. To this end, the posit decoder is modified such that it can encode fixed point numbers to the PIF. While this means that the higher complexity posit hardware is used to compute on fixed point data, the increase in throughput that can be achieved by this approach is probably worthwhile for many applications.

To the best of my knowledge there is no other work that describes the use of both posit and fixed point data in the computation of the same network: all previous work using posits has done so for all network layers. Therefore, in the next chapter the effect this has on the network inference accuracy is investigated.

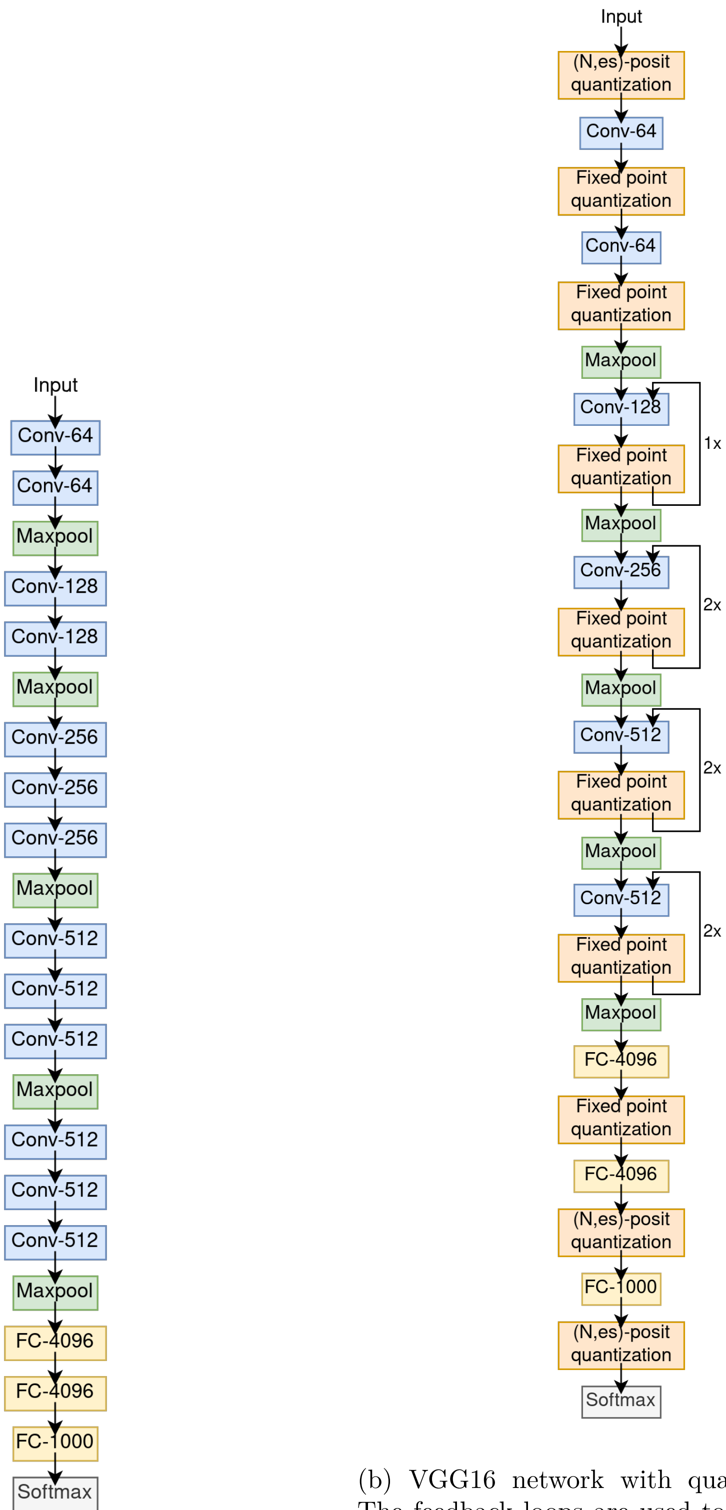
# Inference Accuracy Analysis 4

In this chapter the effect of using different quantization techniques in the same network is investigated, using both posit and fixed point numbers. The baseline network uses 16 bit fixed point numbers for the first and last network layers, and 8 bit precision for the hidden layers. Then, the first and last network layer are quantized to posits instead, to quantify how this affects the inference accuracy that is achieved.
To achieve this, a software model is developed in PyTorch [66], capable of both fixed point and posit quantization of different precisions. The goal here is not to improve the performance in the software flow, but to check whether this approach of mixing number systems does not cause a large degradation of the inference accuracy of the network. Using a simulation in software it is also possible to perform a design space exploration, to get a general idea of the network performance when using certain number representations for different network layers, without having to implement any dedicated hardware.

## 4.1 Network: VGG16

The first step is to decide which network to use for this verification step. In the end, the goal is to use the hardware designed in this thesis in an accelerator meant for object recognition. Therefore it makes sense to use a network designed for such a task. Progress in this field is quite fast so it makes sense to use a modern network for this evaluation because older networks can be obsolete already. Taking these requirements into account, the choice is made to use the VGG16 network [45]. While there are different options that might be newer, like ResNet [67] and variations, the benefit of VGG16 is that it has relatively few network layers, making the addition of quantization layers easier and more clear. It is also a network that is often used in the literature, allowing for easy comparisons. The goal is to write the software as generic as possible, to allow the same verification to be done for different networks if required.
The structure of the VGG16 network can be seen in Fig. 4.1a, where each convolution is $3\times3$ and the number indicates the number of channels for each layer. Instead of using large receptive field sizes, Simonyan et al. decided to use multiple consecutive smaller convolutions of size $3\times3$, without pooling in between. By stacking multiple convolutions without intermediate pooling, the effective receptive field of the convolution is increased: three consecutive $3 \times 3$ convolutions have an effective receptive field of $7 \times 7$. This approach introduces more non-linearity and decreases the number of parameters in the model, compared to a single $7 \times 7$ convolution [45]. The convolutional layers all have a stride of 1. The last three network layers are fully connected, to perform the final classification of the detected features.
There are three types of data in a network calculation: the trained weights and biases,

and the computed activations. Since the computations are done on general purpose hardware, it is not possible to perform the computations using optimized hardware: all of the data is represented using single precision floating point, similar to [61]. This also means exact accumulation is not supported. It is however possible to use only those values of single precision floats that can also be represented in the desired number representation. When, for example, the floating point number with a value of 0.6 needs to be represented in a fixed point format with only two fractional bits, it can not be exactly represented: the ULP of the fixed point format has a value of 0.25. Therefore, the number needs to be rounded to be represented in the fixed point format, where it will have a value of 0.5. Because the hardware can not compute with this custom number type, the data will still be stored as a single precision float but with the new value of 0.5, instead of 0.6.
The quantization to posit values can not be easily implemented to be performed on GPU, meaning this process is performed on CPU instead. For this reason the choice is made not to perform any fine-tuning to account for the quantization, as this would result in a lot of workload. While this may not result in optimal inference accuracy, the results are obtained under the same conditions. This results in a valid comparison between the different quantization techniques. The pre-trained network from the TorchVision PyTorch package is used for simplicity. This pre-trained model contains the network architecture, as well as the weights and biases for each network layer. First, these weights and biases are quantized to the correct number representation. Then, the network is modified such that the computed activations are also quantized to the correct number representations during inference.




(a) VGG16 network architecture.

(b) VGG16 network with quantization layers. The feedback loops are used to simplify the figure, in reality they are different layers with their own weight sets.

Figure 4.1: VGG16 before and after adding quantization layers.

### 4.1.1 Weight & Bias Quantization

The pretrained weights and biases are loaded from the TorchVision package. Since these parameters remain constant during inference they can be quantized before evaluation starts, without any changes to the architecture of the model.
For the quantization to fixed point format, the Utee Python package [68] is used. Dynamic fixed point quantization is used, as in [9], meaning the trade-off between overflow error and quantization error is made for every layer independently. Using more integer bits reduces the overflow error but less fraction bits remain, which increases the quantization error. Looking at all the weights for a single layer, it computes how many integer bits are needed to prevent overflow in the fixed point representation after quantizing the number. The Utee package can allow a certain percentage of the weights to overflow after quantization, to prevent a few outliers from increasing the integer part of the numbers at the cost of fraction bits. The bits that are not used to represent the integer part or the sign represent the fractional part of the fixed point number. After deciding on the number of integer and fraction bits, each weight value is quantized to a value that is representable in this fixed point representation. The same process is done for the biases, and repeated for each network layer.
This process can be clarified with an example. Say that the set of weights of a certain layer consists of the following values, which need to be quantized to 8 bit fixed point:

$$\{1.78, 0.63, 0.20, -0.91, 0.03, -0.64, -0.09, 0.83, 0.07\}$$

The largest magnitude number of this set is 1.78, which can be represented with $\lceil log_2 1.78 \rceil = 1$ integer bit. Accounting for the sign bit, this leaves 6 bits to represent the fractions for a total precision of 8 bits. Using this number system, some examples of the quantized weight values would be $quantize(1.78) = 1.78125$ and $quantize(0.07) = 0.0625$. Now, say that 10% of the quantized number are allowed to overflow for the sake of this example. In that case the largest magnitude number to represent would be $-0.91$, which can be represented with $\lceil log_2 0.91 \rceil = 0$ integer bits. This means an extra bit will be available for the fraction of each number.
Using dynamic fixed point quantization makes sure that an optimal integer/fraction bit ratio is used for each layer individually, given a fixed width of the total number representation. Since integer hardware is typically used to compute with fixed point numbers, it should be feasible to use different parameters for each layer in hardware as well. If that is not the case, the code written for this software verification can relatively easily be modified to compute the optimal fixed point parameters considering the weights and biases of all layers. This would cost accuracy in the layers that could be computed with a smaller integer part, since these now have a shorter fractional part.
For the quantization to posit numbers, the Julia language SigmoidNumbers library is used [69, 70]. This library is chosen because it is fully parameterized: posits of any bit width and exponent size can be computed with. A lot of the other software implementations of posit numbers only offer a few possible configurations. It is also chosen because one of the supervisors of this project had a positive experience using this library. Using the Julia library PyCall [71], Julia functions can be called from Python such that they can be used with the PyTorch DNN model. A function was written that quantizes an input floating point number to a posit with the given width and exponent

size parameters, and then converts it back to single precision floating point data type. This function can then be called from python to quantize the weights and biases of the first and last network layer to the posit number system.
While it is not ideal to convert the quantized posit values back to single precision, this should not be a source of large inaccuracies in this case. For example, minpos of (8,2)-posit is $2^{-24}$ which should be representable as a subnormal single precision value, but not a normal one. For posits of larger precision or with a larger maximum exponent size this problem becomes more pronounced, as more posit values will be outside of the representable range of single precision floats. However, for the small precision posits that are of interest here, almost all values should be exactly representable as a single precision float. (8,1)-posits for example are in the range $\pm[2^{-12}, 2^{12}]$, all of which are exactly representable as floats. The choice is therefore made not to use a more complicated software emulation of the posit number system during the inference process, to prevent the longer run times this would cause.

### 4.1.2 Activation Quantization

The computed activations during inference should also be quantized to the desired number representations. This can not be done beforehand, since the computed values are obviously not known then. This means that the quantization should be incorporated into the network in order to do it during inference. To achieve this, additional layers are added in between the existing layers of the network model. Instead of performing convolution or pooling, these layers quantize the activations to the desired number format before the next computation. How the network looks after these quantization layers have been added can be seen in Fig. 4.1b.
For the posit quantization a new Torch class is implemented that calls the same Julia function as for weight and bias quantization during a forward pass through the network. This means that the computed, single precision, activation values are all quantized to their corresponding posit values before being used as an input for the next layer.
For the quantization to fixed point, an extra step is required before inference can start: the number of integer bits in the representation need to be decided for each layer, which should be based on the number distribution of the activations of each layer. To do this, a certain number of input images are used to calculate their activations. For the first input image, for each layer the number of required integer bits is calculated according to the same process as for weight and bias quantization, but now for the computed activation values of each layer. The output of the quantization layers is equal to their input during this process, to make sure the deeper network layers can also correctly calculate the required fixed point integer bits. Then the same is done for a second input image and the number of integer bits is updated for a layer if the second image requires a larger number of integer bits. How many images are used for this can easily be changed, but is at 10 by default. After this process, the network is ready to be used for inference where the calculations are done in the desired number formats.

## 4.2 Verification Dataset

After implementing the necessary changes to the network, the next step is to feed it labeled images to see what the achieved inference accuracy is. For this, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) dataset [46] is used. This dataset contains more than 1.2 million training images classified into 1000 categories, as well as 50.000 validation images: 50 from each of the categories. This dataset is chosen because it is often used to train and validate image classification networks, and it is also the dataset that the VGG16 network was trained on. During training, the images were scaled and cropped to a size of $224 \times 224$ pixels, so this is also done for the images used for verification. For this verification, a subset of the validation images is used in order to reduce the test time: 5000 random validation images are used instead of the complete set. The same images are used for each of the experiments, to make sure the results are consistent.

## 4.3 Results

The first computation to run is the baseline VGG16 network using single precision floating point, in order to compare the networks to that use different quantization techniques. After that, the experiment is performed again under the same conditions, but with different quantization techniques: the hidden layers are quantized to 8 bit fixed point, while the first and last network layers are quantized to different number systems. Because the maximum exponent size has quite a large impact on the number distribution of posit numbers of the same precision, the exponent size is varied for the same precision as well to find the optimal value: the exponent size is set to zero first and incremented until the achieved accuracy starts to decrease.
Table 4.1 shows the obtained results, including both top-1 and top-5 accuracy. If the image is assigned to the correct class with the highest probability, this is counted towards the top-1 accuracy. ILSVRC however, ranks the challenge entries on top-5 accuracy. This means the correct class is assigned one of the five highest probabilities of being the correct one by the network. The reason for choosing this metric is that there might be multiple objects in an image while it is assigned only a single label [46]: a picture of a bowl of fruit can contain both an apple and a pear but will only be labeled as one of the two. However, the correct class should in this case be assigned a high probability because the network should detect both the apple and pear. Therefore, the labeled category should be in the top-5 classifications.
As can be seen from the results in Table 4.1, the same parameters result in the lowest accuracy degradation when considering top-1 and top-5 accuracy for 8 and 9 bit posits. For 10 bit posits there is even an instances where the inference accuracy has increased compared to using single precision floating point numbers, indicated by a negative degradation. This can probably be attributed to the regularization effect quantization can cause, as discussed in Section 3.2.4. For 10 bit posit, an exponent size of 0 results in the smallest top-5 accuracy degradation, while an exponent size of 2 results in the smallest top-1 accuracy degradation. Because of the large difference between top-1 accuracy and negligible difference in top-5 accuracy, (10,2)-posit is probably the best

Table 4.1: Simulation results for different number systems, where the hidden layers are quantized to 8 bit fixed point numbers. The single precision baseline uses no quantization in any layer.

| First & last layer | Top-1 accuracy [%] | Top-5 accuracy [%] | Top-1 degradation [%] | Top-5 degradation [%] |
|---|---|---|---|---|
| Single precision floating point | 71.92 | 89.80 | Baseline | Baseline |
| 16-bit fixed point | 71.50 | 89.72 | 0.42 | 0.08 |
| (8,0)-posit | 61.98 | 88.90 | 9.94 | 0.90 |
| (8,1)-posit | 68.16 | 89.48 | **3.76** | **0.32** |
| (8,2)-posit | 68.06 | 89.42 | 3.86 | 0.38 |
| (9,0)-posit | 67.80 | 89.68 | 4.12 | 0.12 |
| (9,1)-posit | 70.94 | 89.76 | **0.98** | **0.04** |
| (9,2)-posit | 70.86 | 89.52 | 1.06 | 0.28 |
| (10,0)-posit | 70.92 | 89.80 | 1.00 | **0.00** |
| (10,1)-posit | 71.60 | 89.70 | 0.32 | 0.10 |
| (10,2)-posit | 72.02 | 89.78 | **-0.10** | 0.02 |
| (10,3)-posit | 71.22 | 89.40 | 0.70 | 0.40 |

option according to these results.
These results were obtained by using the activations of 10 input images to decide the fixed point parameters for the activations. Furthermore, none of the numbers were allowed to overflow due to quantization to fixed point, meaning a single outlier could cause an extra integer bit to be used for the fixed point representation, at the cost of a fraction bit. By changing these parameters, it is likely that marginally better results could be achieved for the same number precisions. Since it would require a lot of extra run time to perform all the experiments, these optimization have not been pursued here: it should not have a large impact on the comparison of the different quantization techniques.
For (8,1)-posit, the top-1 accuracy degradation is quite significant at 3.34% compared to the network using 16 bit fixed point quantization for the first and last network layers. However, according to the designers of the ImageNet dataset and the accompanying challenge [46], the top-5 accuracy is more important as a metric to rate the network performance. This shows a degradation of only 0.24% compared to the fixed point network, which is probably an acceptable degradation to reduce the required memory size and bandwidth for inference, and with that reduce the energy consumption as well. If this amount of accuracy degradation is not acceptable for a certain application, 9 or 10 bit posit can be used instead. This does come at the cost of increasing the required memory size and bandwidth, and hardware cost. This is also a nice example of one of the beauties of the posit number system: the designer has total freedom in choosing the required precision. If 8 bits are not sufficient, 10 bits can be used. If a lower accuracy would still be acceptable, perhaps 7 bits of precision would also suffice.
While these results already look quite promising, especially for (10,2)-posit, they are

not a fully accurate representation of the results that could be achieved using optimized hardware. These computations are done using general purpose single precision hardware, and as such the accumulations are not computed in an exact manner. The exact accumulation in the quire is an important feature of posit arithmetic that could have a large impact on the accuracy of the computed result, as discussed in Section 2.1.3.4. For the final application, it would also make sense to perform some form of fine-tuning of the parameters to account for the network quantization. This should help reduce the accuracy degradation.

# 5 Hardware Designs

In this chapter the design of a parameterized posit processing element performing exact accumulation is discussed, as well as its use in a parameterized systolic array. This means a rectangular systolic array of any size can be generated that computes exact dot products with arbitrary (N,es)-posit numbers. On top of that, the SA should be compatible to be used in a system that also contains hardware that performs fixed point computations as discussed in Section 3.4. To this end, a hybrid decoder is designed that converts fixed point inputs to the PIF, as well as a quire to fixed point converter. In this chapter the hardware designs are discussed: first, the design of the SA is discussed in Section 5.1, followed by the design of the PEs in Section 5.2. The verification and results of the hardware implementation are discussed in Sections 5.3 & 5.4, respectively.

## 5.1 Posit Systolic Array Design

In this section the design of the systolic array (SA) is described, as well as some tradeoffs that will be evaluated. The size of the SA is parameterized for easy design space exploration and for the design to be easily used in different applications. First, the used data flows through the SA are explained. Then in Section 5.1.2 two different approaches for the decoding of the posit numbers are discussed.

### 5.1.1 Data Flow

In the processing elements accumulation is done exactly, which means that intermediate results are a lot larger than the input operands. For this reason, the choice is made to implement an output stationary data flow in the SA. The goal of implementing the posit SA is to reduce the memory access energy consumption compared to using a fixed point representation of a larger precision. Having to store partial quire results in memory would therefore defeat the purpose of using the posit number system in the first place.
The general structure of an output stationary SA can be seen in Fig. 5.1, where the array size $N \times M$ is $5 \times 4$. The first input $A$ is fed into the SA from the left and propagates to the right, the second input $B$ propagates from the top to the bottom. While the data flow is output stationary, the outputs $C$ still need to be read from the SA after the computation is finished. When this happens, the outputs propagate in the same direction as input $B$, as indicated in Fig. 5.1. Another approach is to read all the outputs in a single cycle, but this would increase the amount of required interconnect and memory bandwidth, and reduce the regularity of the SA architecture.
The output of each PE is stored in registers, to be output to the next PE in the next

clock cycle. This is in accordance with the designs discussed in Section 5.2.1.

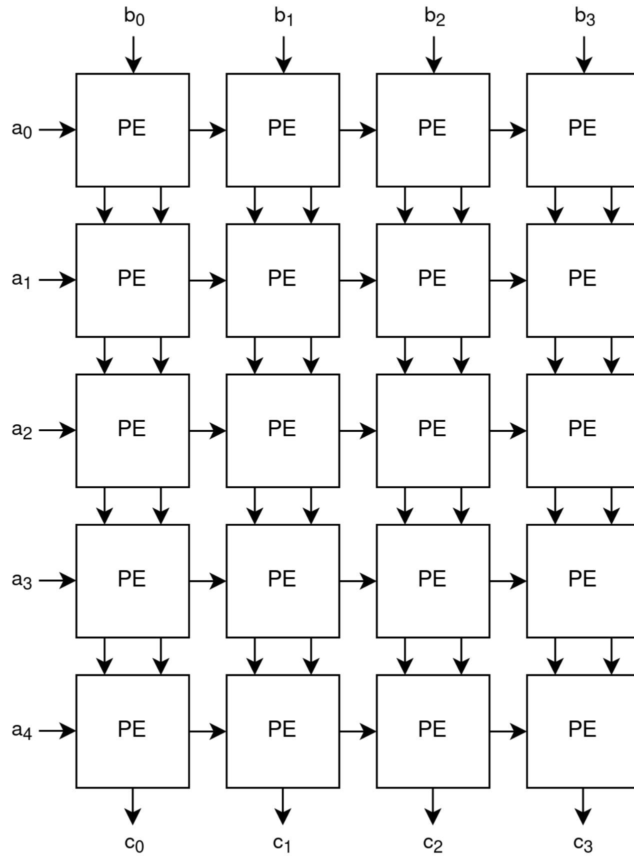
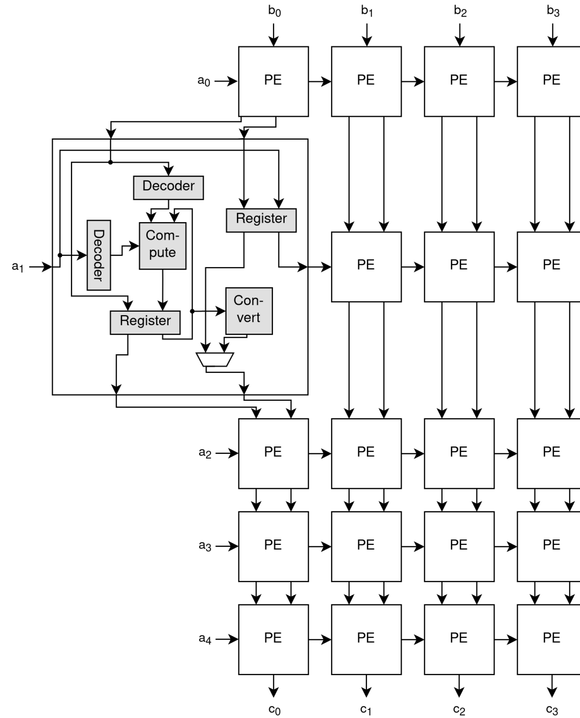



Figure 5.1: Output stationary systolic array of size $5 \times 4$.

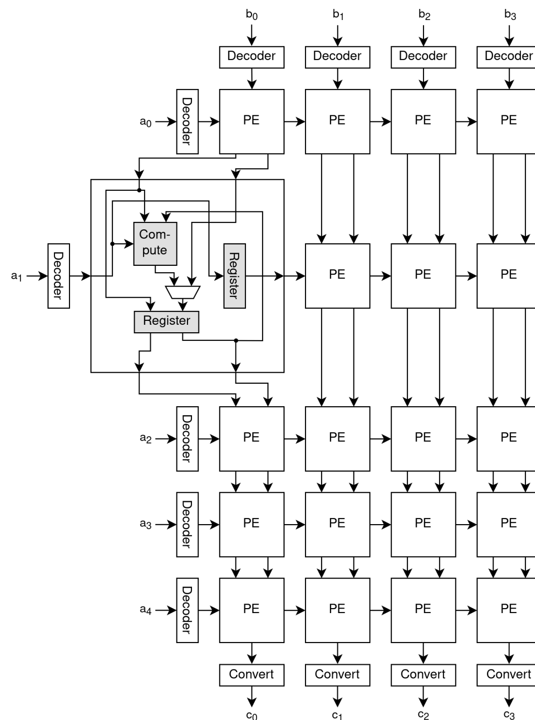
### 5.1.2 Edge & Distributed Conversion

In the SA, there is data reuse of the two input numbers. This also gives rise to different approaches of how to handle the decoding, such that the numbers can be computed with: they can be decoded in each of the PEs, or they can be decoded before entering the array. The same is true for the conversion of the quire result to a shorter number representation. Both techniques are explained and evaluated.

The SA in which the decoding and quire conversion is done in each of the PEs is called distributed here: the decoding and quire conversion are distributed over the SA. This design can be seen in Fig. 5.2a. In the communication between the different PEs the encoded posit numbers and converted quire results are used, and $N \times M$ decoders and quire converters are required to perform these operations in each PE.




(a) Systolic array with distributed posit decoding and quire conversion.




(b) Systolic array with edge decoding and quire conversion.

Figure 5.2: Two different approaches to handle posit decoding and quire conversion in a systolic array architecture.

With edge conversion, the posit numbers are decoded before being input into the SA: at the edges. The full quire is propagated through the SA when the result is ready, and is converted at the edge of the array. This is shown in Fig. 5.2b, where the communication between the different PEs is now done in the posit intermediate format and the full quire precision. Only $N + M$ decoders and $M$ quire converters are required in this case.
Of course there is a trade-off that needs to be made when deciding between these two different approaches to handle the conversion of data in the SA. The benefit of using edge conversion is that less decode and quire conversion modules need to be present in the hardware, which should reduce the hardware utilization compared to distributed conversion. This is especially true for larger SA sizes. The downside lies in the fact that the PIF consists of more bits than encoded posit numbers: this increases the number of registers required in each of the PEs to store the data, as well as the amount of interconnect that is required between the PEs. By decoding the data in each of the PEs this can be avoided.
However, in the case of distributed conversion, registers are present to store both the full precision quire, as well as the converted quire. Depending on the number systems that are being used, this may end up costing a similar amount of registers as are saved by storing the encoded posit numbers. For this reason two different distributed PEs are designed, one of which reuses the full precision quire registers to store the converted quire result. This reduces the register utilization at the cost of some extra logic. The PE designs are discussed in Section 5.2.1.2.
The exact impact of this trade-off is evaluated by implementing the different architectures. Which of the approaches is more appropriate depends on the specific application: if the number of available flip-flops is limited on a target device, or a lot of them are required for a different part of the system, it may be more attractive to use distributed decoding because it should require less flip-flops. If minimizing the amount of combinational hardware is more important, edge conversion is probably more suitable.

## 5.2 Posit Processing Element Design

In this section, the design of a PE that can compute with the posit number system is discussed. The design of a hybrid decoder is also discussed, which can convert both posit and fixed point input numbers into the PIF. The implementation of the designs is done at the register-transfer level (RTL) in SystemVerilog [72], and is fully parameterized to make sure it is easy to use and to perform design space explorations. Then, some variations of the PE are discussed, in order to evaluate different trade-offs that can be made in the design.

### 5.2.1 Processing Element Design

The design of the PE is split into two parts, in order to separate the data flow through the PE from the computation. This prevents the need to check the correctness of the computation when a change is made to the data flow. The top-level PE that takes care of the data flow is described in Section 5.2.1.2, first the computational functionality is

discussed in the next section.

#### 5.2.1.1 Computational Processing Element

In the basic PE, there are three inputs and a single output. The inputs are the multiplicand and multiplier in the PIF, and the current accumulator value in quire format. The single output is the result of adding the product to the quire input. The design can be seen in Fig. 5.3.

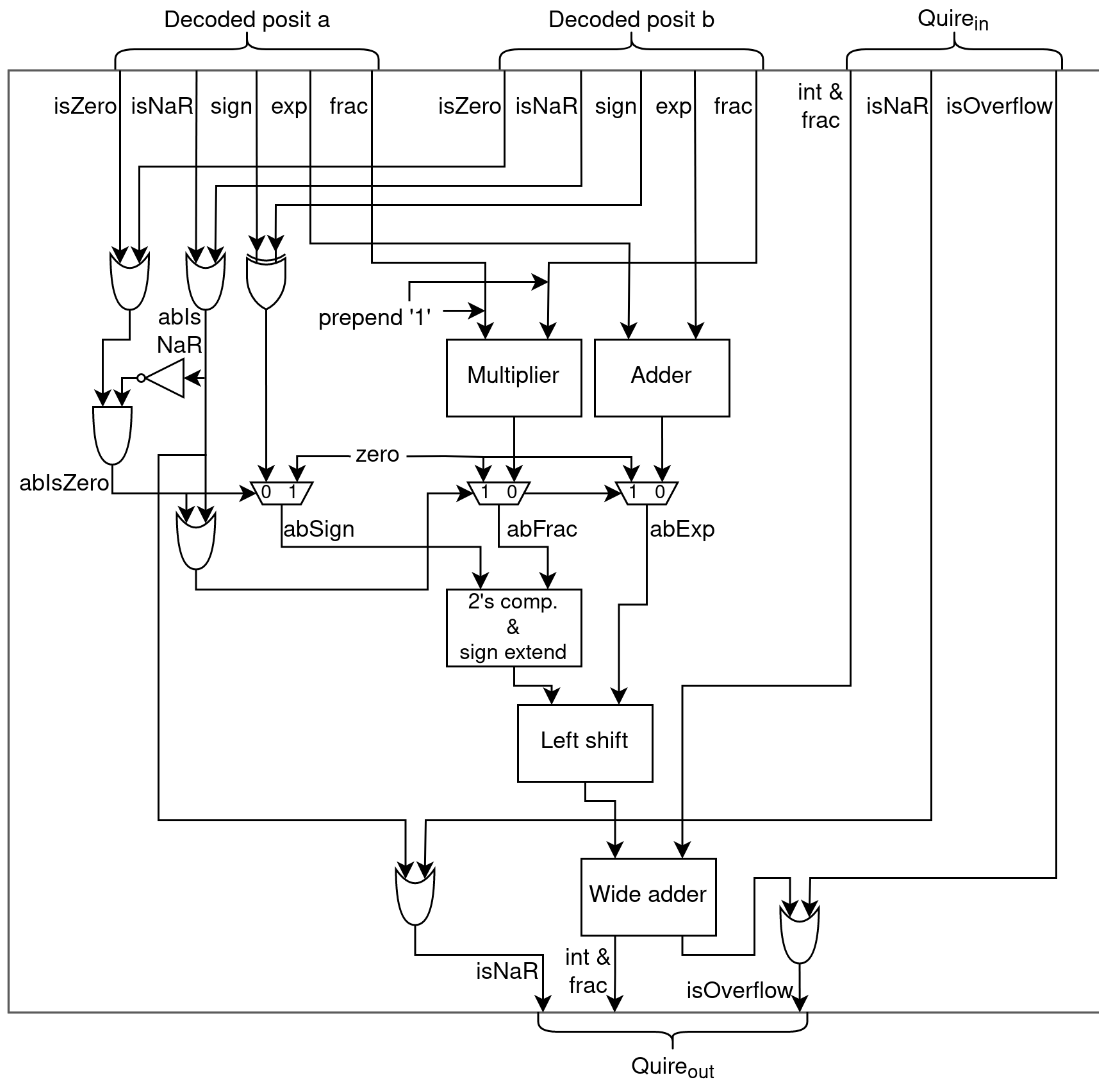



Figure 5.3: Posit processing element for exact accumulation.

The first step in this module is to multiply the two PIF inputs: the exponents are added and the significands multiplied. To this end a set bit is prepended to the input fraction before multiplication. The accumulation is done exactly, as such there is no

need to perform rounding or normalization on the multiplication results. By choosing a smart bias value for the exponent representation in the PIF, the compensation step to account for the bias after computing with biased exponents can be prevented. This is explained in more detail in Section 5.2.2.2. Since the PIF contains flags indicating the sign, and whether the number is zero or NaR, these conditions can easily be detected for the result of the multiplication: if either of the operands is zero or NaR, so is the result. If this is the case, the exponent and fraction are also set to zero accordingly.
After the multiplication the 2's complement of the fraction is taken if the sign after multiplication is negative. This is because the fractions only represent a magnitude after decoding, the sign is represented separately. The same is therefore true for the fraction result after multiplication. The quire, however, is represented in 2's complement representation for easy addition of negative products. The number that is added to the quire is a shifted version of the fraction product, so therefore it first needs to be converted to the 2's complement representation. After the conversion the fraction is sign extended for a correct alignment to the quire. Because the amount it is extended is related to the exponent bias, it is also discussed in Section 5.2.2.2 in more detail.
The fraction is then shifted to the left, where the shift amount is dictated by the value of the exponent of the product: as the location of the binary point in the quire is fixed, it does not influence the required shift amount. As the amount of this shift is not constant, a barrel shifter is required. After the shift the fraction is exactly aligned with the fixed point quire and can be added to it. In the current implementation this addition is done rather naively using a ripple-carry adder. Because the quire is wide compared to the precision of the input numbers, in order to accommodate exact accumulation, replacing this adder with a different type that introduces less delay could be worthwhile.

#### 5.2.1.2 Top-Level Processing Element

The top-level processing element instantiates the computational module discussed in the previous section and takes care of the data flow through the systolic array. The benefit of making this distinction in functionality is that it becomes easier to locate potential errors in the design, and it makes sure that the computational correctness of the PE does not need to be verified again after making a change to the data flow. The data flow described here is an output stationary one, as discussed in Section 5.1.1, which means the computed quire will remain in the same PE until the computation is finished. It also makes it easier to design the different PEs that are required for the edge and distributed SAs that were discussed in Section 5.1.2.

**Distributed Processing Element** The design of the top-level PE with the decoding and quire conversion included can be seen in Fig. 5.4. First, the input posit numbers are decoded into the PIF, which is then used as input for the computation. The quire value that was computed in the previous cycle is used as input for the computation to continue the accumulation. The encoded input posits and the newly computed quire are stored in registers. The posit inputs are passed on to the next PEs in the next clock cycle that will use them again for computation.

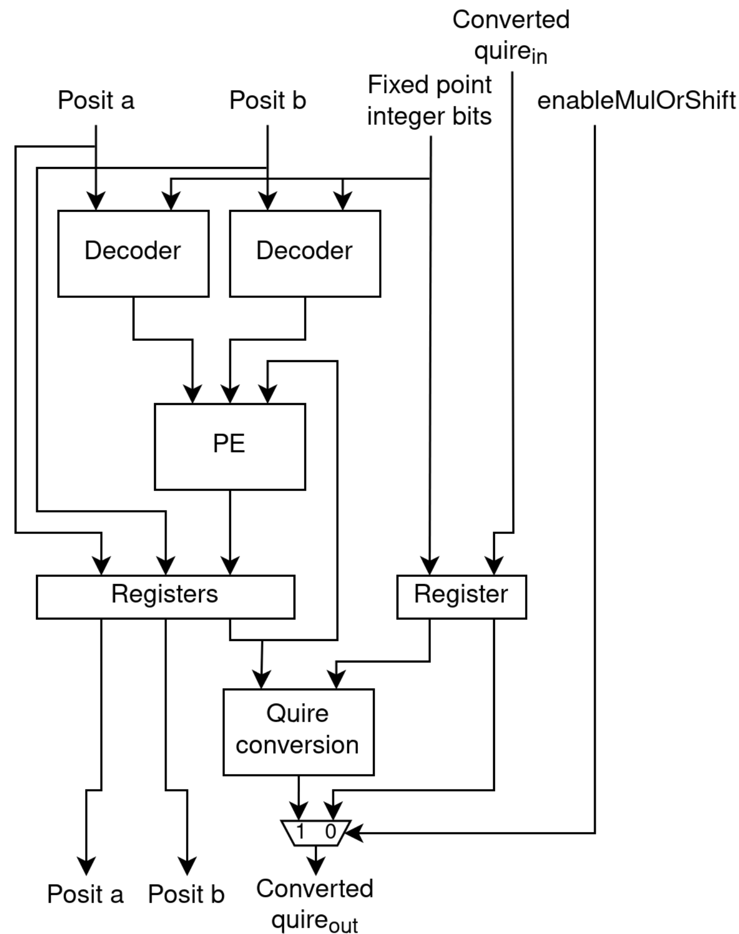



Figure 5.4: Top-level distributed processing element for an output stationary data flow.

Depending on what number representation is required in the next computation, the fixed point quire is converted to another representation before the output. The loss in accuracy that is possibly caused by this conversion is the only loss of accuracy in the PE, before that the result is exact. This conversion can for example be to a floating point number, a posit number, or a fixed point number. It is also possible to use the entire quire as output, if the next computation can handle the large precision of this number. The conversion used here is explained in Section 5.2.4.
What happens with the quire depends on the required operation. During normal computation in an output stationary flow, the quire is fed back to be used for computation in the same PE in the next clock cycle. The current quire is also converted, the result of which is sent to the next PE in the array, where it is stored in a register. Then, when the accumulation is finished, this converted quire value can be immediately propagated to the next PE, using the control signal *enableMulOrShift*.
Alternatively, the full precision quire register can be reused to store the converted quire results when the computation is finished, as shown in Fig. 5.5. This reduces the reg-

ister utilization, but does require an extra set of multiplexers to select which signal to store in the registers. A small, two state finite-state machine (FSM) is also required to select the output of the PE: the converted result of the current PE, or the result that was already converted in a previous PE. This increases the logic utilization compared to the approach shown in Fig. 5.4. Both of these approaches are evaluated, and the second one is referred to as the distributed PE with FF reuse.

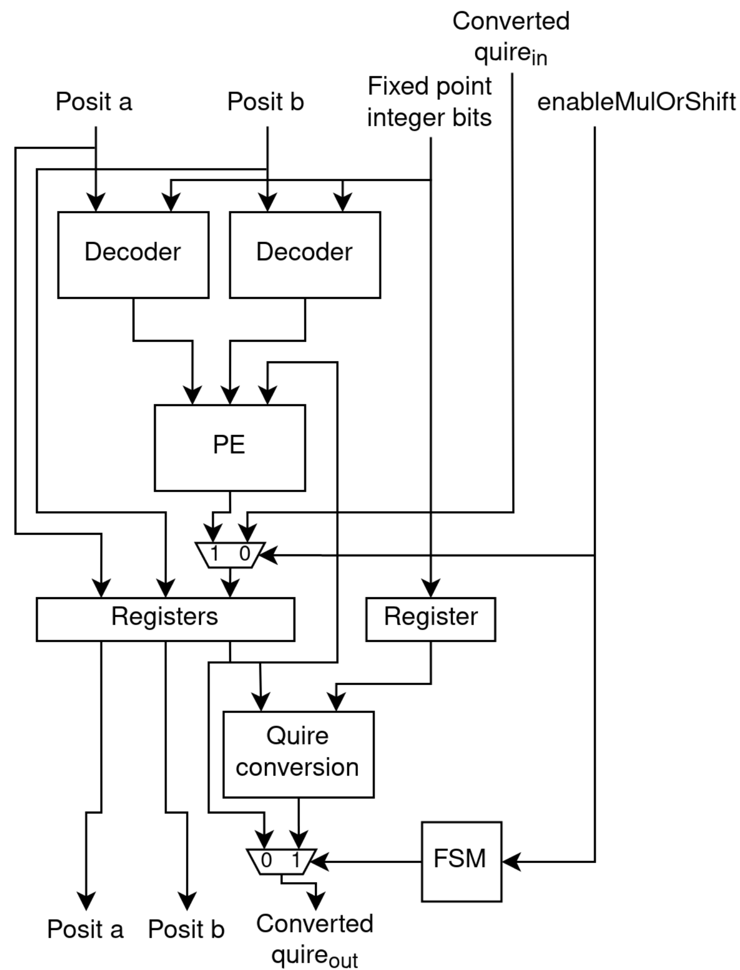



Figure 5.5: Top-level distributed processing element for an output stationary data flow, that reuses the quire registers for the converted results.

**Edge Processing Element** As discussed in Section 5.1.2, it could be beneficial to implement a PE where the decoder is not included, but instead the inputs are already decoded posits. This prevents the need to decode the same numbers in each of the PEs. The same can be done for the final conversion of the quire to another number representation. The design of this top-level PE is very similar to the one that does include the decoding, and can be seen in Fig. 5.6.

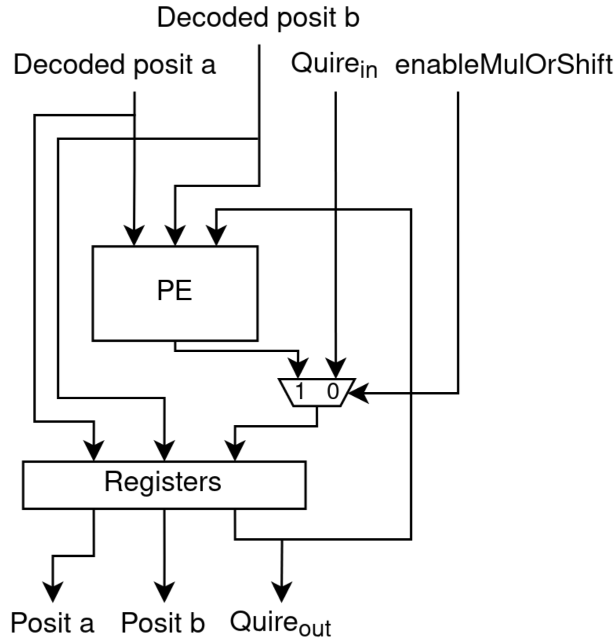



Figure 5.6: Top-level edge processing element for an output stationary data flow.

### 5.2.2 Open-Source Posit Hardware

Some of the work described in Sections 2.3 & 3.3 describes hardware designs that are made open-source [17, 23, 31, 12]. Before rushing into designing all the hardware from scratch, it makes sense to check out some of the designs that have already been made. If anything can be used as a base, this would save a lot of design and debugging time. Of the designs mentioned before, only [17, 23] implement support for using the quire. However, [17] has performed the designs in a C++ library targeting HLS, while this work targets RTL implementation in order to have more control over the implementation details.

This leaves the Deepfloat design developed by Jeff Johnson at Facebook AI Research [23]. In that paper, Jeff Johnson makes a comparison of the ResNet-50 network [67] performance using single precision floats using fused multiply-add, and different posit configurations of 7, 8, and 9 bit precision using exact accumulation. For a precision of 8 bits, (8,1)-posit achieves the best accuracy on the ImageNet validation set [46], similarly to the results discussed in Section 4.3, with a top-5 accuracy degradation of 0.19%.

Upon inspection of the Deepfloat designs that are written in SystemVerilog [72], they are parameterized, modular and easy to understand. This makes them easy to modify to the needs of this project and to evaluate different design trade-offs. For this reason, the choice is made to use these designs as a base during this project, to save time implementing and testing the basic hardware components. This means there will be more

time to investigate design choices at a higher level, to provide insight into different trade-offs that can be made.
The Deepfloat design suite is quite large, and also contains modules and interfaces for different number systems than posit. The most important components that are used here consist both of interfaces that group certain signals together, and hardware modules that perform the computations: the most important components for the PE are the decoder from posit to the intermediate format, the multiplier, and the adder to the quire. For the correct addition of the multiplier result to the quire, there is also a conversion module that takes care of correctly shifting the operands and detecting overflow. A posit PE is also present that instantiates these components, but there is no testbench to verify correct functionality. More details about the hardware components are discussed in Section 5.2.2.2, some details about the interfaces are given next.

#### 5.2.2.1 Deepfloat Interfaces

Interfaces are a construct in SystemVerilog that enables the encapsulation of different signals into one, enabling design reuse and easy integration of different components. It also allows the designer to define functions on interface signals, for example to easily read or write signals that are part of the interface.
The interfaces that are used are those that represent posit numbers, decoded posits in the intermediate format, and the quire for exact accumulation. The posit interface is parameterized by its width and the maximum exponent size, and the only signal it contains is a bit vector of the given length. The reason for defining this as an interface is to make use of the functions: for example to return a posit of value zero for easy reset, or a posit of maximum magnitude that can be used in case of overflow. Using these functions makes the hardware designs more clear, and less prone to errors.
The decoded posit interface is a bit more involved: it is again parameterized by its width ($N$) and maximum exponent size ($es$), and is similar to the PIF as discussed in Section 2.1.3.3, containing the following signals:

- *isZero* on 1 bit, to indicate a zero value posit,
- *isInf* on 1 bit, to indicate a NaR posit. *isNaR* would have been more accurate,
- *sign* on 1 bit, indicating the sign,
- *exponent* on $1 + es + \lceil log_2(N-1) \rceil$ bits, indicating the biased scaling factor,
- *fraction* on $N - (3 + es)$ bits, indicating the fraction.

Functions are defined for example to read the regime part of the scaling factor, and to calculate the unbiased scaling factor.
Finally, there is the Kulisch interface that describes the wide fixed point number that is used for exact accumulation. In posit arithmetic this is typically called the quire, as discussed in Section 2.1.3.4. It is parameterized by the number of integer and fraction bits it contains, and has the following signals:

- *isInf* on 1 bit, to indicate that at least one NaR value has been added to it,

- *isOverflow* on 1 bit, to indicate the accumulator has overflown,
- *overflowSign* on 1 bit, to indicate the sign of the accumulator before overflow occured,
- *accumulator* on $1 + integer + fraction$ bits, indicating the current value of accumulation.

The additional bit in the accumulator holds the sign: as the quire is represented in 2's complement representation the sign is not represented separately. The functions include ones to read the fraction and integer bits separately, read the sign, and set the accumulator value.

#### 5.2.2.2 Deepfloat Hardware Implementation

As already hinted at in Section 5.2.1.1, some choices can be made in the design of the PE that affect the exact implementation details of the hardware. Here, some of those design choices made in the Deepfloat hardware are discussed.
First off, in the Deepfloat design there is already a split between the computation and data flow of the PE. However, the feedback of the new quire value back to the input to be used during the next computation is already done in the computational part of the design. This limits the flexibility of the design, so this was changed to be done in the top-level PE. This makes it easier to change the data flow, if this is required.
An important design choice that was made in the Deepfloat designs was to interpret posit numbers as being in sign & magnitude representation, rather than the standard 2's complement representation. For positive numbers this has no effect, but negative numbers now get assigned a different magnitude than what would be expected according to the standard posit interpretation. This was done to make the decoding easier: the step of taking the 2's complement can be skipped, which saves a negation and an addition for negative numbers. Because the two representations for zero in a sign & magnitude representation are interpreted differently for posit anyway, this does not introduce a positive and negative zero in the posit representation. In theory this makes the optimization worthwhile, because the decoder can be optimized without there being a real drawback. However, it is quite a significant change to how the numbers are interpreted while the benefit is small compared to the area and delay of the other computational modules, as shown in Section 5.4.1. Therefore, it may make more sense to adhere to the expected posit interpretation according to the standard number system at the cost of a small area and delay overhead.
An important optimization that is in the Deepfloat design is the choice of the bias value for the exponent representation in the PIF. According to Section 2.1.3.3, the typical choice is to use $bias = (N - 2) * 2^{es} + 1$ where $N$ is the precision, in order to let the smallest representable exponent be represented by an unsigned value of 1 in the biased representation, similar to the floating point standard. For floating point numbers this is done because the unsigned exponent value 0 is used to represent subnormal numbers and zero. However, because there is a separate flag to indicate a zero value in the PIF and there are no subnormal numbers in the posit number system, the exponent of value 0 does not need to be reserved for this. By making the bias used in the Deepfloat design

equal to $bias_{Deepfloat} = (N-2)*2^{es}$, the smallest representable exponent becomes equal to 0 in the biased representation. This bias value is equal to the exponent magnitude of *minpos* and *maxpos*.
The benefit of choosing this bias is that the summation of two biases in the exponent is exactly equal to the number of fraction bits of the quire: $2*(N-2)*2^{es}$. This means that if the binary point of the significand product is aligned to be after the least significant quire bit, the biased exponent result can be used to correctly align the product to the quire for addition. This can be achieved by sign extension as mentioned in Section 5.2.1.1, and saves the cost of an extra addition to the exponent that would otherwise be required to compensate for the bias. It also means that the alignment shifter only needs to be able to shift to the left, as the product fraction is already aligned to the right side of the quire. To achieve this, the significand product is sign extended to the length of the quire, plus an additional number of bits equal to the number of fraction bit in the quire. This should become more clear by the use of an example.

Table 5.1: Alignment of a product to the quire.

| | Integer part | Fractional part |
|---|---|---|
| Significand product | $si_1i_0.$ | $f_{-1}f_{-2}f_{-3}f_{-4}$ |
| Quire | $si_7i_6i_5i_4i_3i_2i_1i_0.$ | $f_{-1}f_{-2}f_{-3}f_{-4}$ |
| Extended to quire length | $sssssssi_1i_0.$ | $f_{-1}f_{-2}f_{-3}f_{-4}$ |
| Extended additional *fraction* (4) bits | $sssssssss.$ | $ssi_1i_0\ f_{-1}f_{-2}f_{-3}f_{-4}$ |

(4,0)-posit is used for this example, to keep the number of bits small for the sake of clarity. For this number system, the $minpos = 2^{-(N-2)*2^{es}} = 2^{-2}$ which means that the smallest possible product is $minpos^2 = 2^{-4}$, which requires 4 bits to be represented exactly in the quire. As $maxpos^2 = 2^4$, the number of integer bits is $1+(4-1)+5 = 9$ bits as shown in Table 5.1, accounting for the sign bit and $N-1$ bits to prevent overflow. The bias for (4,0)-posit is $bias_{Deepfloat} = (N-2)*2^{es} = 2$. After the product, the bias will be in the exponent twice, as shown in (5.1), meaning it will have a value of 4: exactly equal to the number of fraction bits in the quire.

$$2^{x+bias} * 2^{y+bias} = 2^{x+y+2*bias} \tag{5.1}$$

The result of a single product of two fractions can be represented as in the first row of Table 5.1, with a sign bit, 2 integer bits and 4 fraction bits. The number of integer bits is independent of the chosen parameters for the posit number system: as the significands are in the range $[1,2)$ their product will be in the range $[1,4)$. To understand the amount of sign extension, the product $minpos^2 = 001.0000 * 2^{-4}$ can be used as example: it will have a biased exponent with value $-4+2*bias = 0$. If this is to be directly used as the required left shift of the significand product, the least significant integer bit of the significand product should be aligned with the LSB of the quire to be added to the quire correctly. This can be achieved by extending the significand product to the length of the quire, plus another *quire fraction* bits, as shown in Table 5.1. This

extension also works for all other products than $minpos^2$: when the product is larger and has a non-zero biased exponent, the significand product is left shifted according to the exponent value.
As seen from the table, this extension does shift the fraction bits of the significand product beyond the precision of the quire. After correctly aligning the product to the quire by shifting it to the left, the extra $fraction$ bits are truncated to make the length equal to the quire length again. The truncated bits will always be zero, either due to the tapered precision of posits, reducing the number of fraction bits, or because the fraction bits have been left shifted into the quire range. After truncation, the product can be added to the quire.
Another benefit of this bias is that in the decoder it can be added to the PIF exponent before concatenation of the es bits, meaning a shorter adder can be used to add the bias during decoding. The decoder that is used is also more optimized compared to the one discussed in Section 2.2.4.1. For more details on the decoder refer to Section 5.2.3, where the design is discussed in more detail because more changes will be made to it.
Subtleties like using this value for the exponent bias demonstrate the benefit of using an existing design, instead of starting a design from scratch. Realizing that such an optimization can be made from something simple as changing the exponent bias value can be hard, while it does offer multiple benefits. On top of that, the paper for which Deepfloat was developed [23] focuses on the effects the posit number system has on DNN inference accuracy, not on the implementation details of the hardware that has been designed. It would therefore be easy to miss such an optimization if one is not extensively studying and using the hardware designs.

### 5.2.3 Encode Fixed Point Numbers to PIF

In order to increase the throughput of the compute system it would be beneficial to be able to use the posit SA tiles to compute on fixed point data, as discussed in Section 3.4. The benefit of doing this is that the posit hardware does not have to be idle during the hidden layer computations, increasing the maximum throughput during this stage of computation. However, this effectively uses more complex hardware than required by the data, which comes at a cost in power and delay. Whether this increase in complexity and cost is worth it to achieve a higher throughput is a trade-off that depends on the application and design goals.
The first approach to achieve fixed point computation using posit hardware that springs to mind is to use the significand multiplier for the fixed point numbers directly. This would require modifications to be made to the alignment shift for correct addition to the quire, or changing the interpretation of the quire in case of fixed point accumulation. As the significand multiplier uses unsigned operands, more changes would be required than just increasing its precision to accommodate signed fixed point computation. Another approach is to encode the fixed point numbers to the PIF: doing this, the rest of the hardware components of the PE will not need to be changed. This simplifies the modifications that need to be made, and limits them to a single component. This approach is further investigated here.
The PIF that the posit hardware computes on consists of an exponent and fraction part, representing fixed point numbers in this format is not a straightforward operation.

However, many components of the posit decoder can be reused to convert fixed point input numbers to the posit intermediate representation, as is shown later in this section. First, the Deepfloat decoder design is described as it is a more optimized version than the one discussed in Section 2.2.4.1. Then, the PIF field length is reevaluated in order to accommodate the representation of fixed point data in Section 5.2.3.2. Finally, a new decoder design is described that can convert both posit and fixed point inputs to the PIF in Section 5.2.3.3.

#### 5.2.3.1 Deepfloat Decoder Design

The first major difference to the decoder described in Section 2.2.4.1 is that the Deepfloat hardware interprets posit numbers as being sign & magnitude instead of 2's complement, as mentioned before in Section 5.2.2.2. The second difference is that XOR gates are not used to selectively invert bits if the regime starts with a set bit, but instead are always used to XOR the neighboring bits of the input, as shown in Fig. 5.7. This works as shown in Table 5.2: the XORed signal always starts with zeros, even when the regime starts with set bits.
There are multiple benefits to using this approach, the first being that one less XOR gate is required, slightly reducing the hardware area. As there is one less gate, the XORed signal has a length of $N-2$, reducing the required size of the leading zero counter by 1 bit as well. It also means that the result of counting the sequence of most significant zeros returns the length of the regime $-2$. This is beneficial because the minimum length of the regime is known to be 2 bits, so there is no need to shift these bits out to correctly access the exponent and fraction bits of the number that is being decoded. This means the size of the shifter can be reduced, again saving some hardware area.

Table 5.2: The functionality of the neighboring bit XOR.

| Input bits | S0000011 | S1111001 |
|---|---|---|
| Neighboring XOR | 000010 | 000101 |

This change in input to the leading zero counter also means the count should be interpreted differently in order to assign the exponent the correct value. A summary of the regime bits, their value, and the intermediate signal is shown in Table 5.3. The reason the all-0's regime does not get a defined value is because it is only used to indicate zero and NaR in the posit system. For the LZC(XOR) and the regime value rows in the table the 2's complement representation of the unsigned integer values is also shown, to make it obvious that the negative regime values can be obtained simply by inverting the leading zero count. The leading zero count is unsigned and is represented in the hardware with only 2 bits in this case, but is extended in the table for the sake of clarity. The sign is prepended in the hardware as in Fig. 5.7. This inversion is significantly cheaper than the subtraction or 2's complement conversion that is required in the simpler decoder described in Section 2.2.4.1.
As the bias value that is being used is $(N-2) * 2^{es}$, there is an alternative to adding the complete bias to the final exponent. Instead, a value of $N-2$ can be added to the

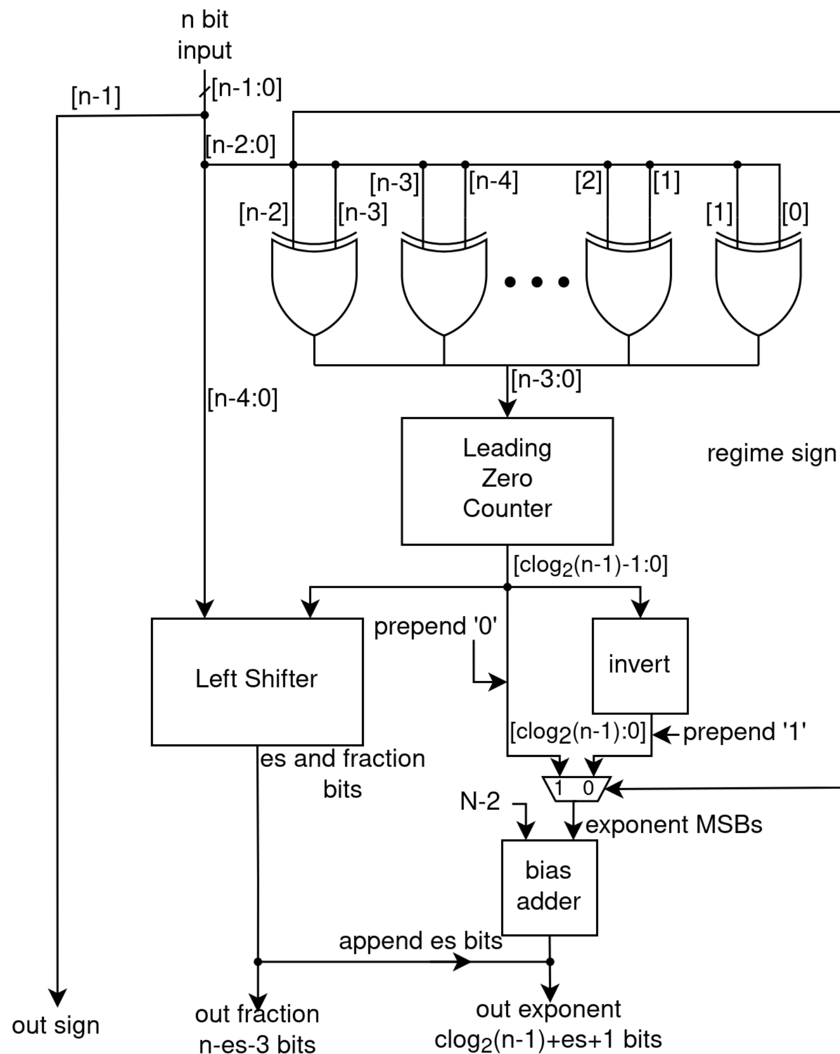



Figure 5.7: The Deepfloat decoder design.

value of the regime before the *es* bits are appended that are determined by the shift. The appending of the *es* bits then takes care of correcting the value of the bias. The benefit is that the adder can be *es* bits shorter, saving both area and delay.

#### 5.2.3.2 PIF Field Lengths

The PIF will now not only be used to represent decoded posit numbers, but fixed point numbers as well. This means that the length of the fraction and exponent fields may need to be longer than required to only represent posits. During the design of this decoder two assumptions are made with regard to the chosen parameters:

1. The bit width of the fixed point numbers will not be wider than the posit numbers,

Table 5.3: How the leading zero count of the XORed input signal is interpreted, where LZC() is the leading zero count. The signals in brackets are the 2's complement representation of the signed integers.

| Regime bits | 0000 | 0001 | 001x | 01xx | 10xx | 110x | 1110 | 1111 |
|---|---|---|---|---|---|---|---|---|
| Neighbor XOR | 000 | 001 | 01x | 1xx | 1xx | 01x | 001 | 000 |
| LZC(XOR) | 3 (011) | 2 (010) | 1 (001) | 0 (000) | 0 (000) | 1 (001) | 2 (010) | 3 (011) |
| Regime value | x | -3 (101) | -2 (110) | -1 (111) | 0 (000) | 1 (001) | 2 (010) | 3 (011) |

2. The bit width of the fixed point numbers can be shorter than the posit numbers.

Assumption 1 is justified from the main reason for using posit numbers for the first and last layer computation: to decrease the size of memory accesses, to reduce energy consumption. Allowing a wider fixed point input would undo a lot of the work done to achieve this goal. Making the design with assumption 2 will increase the flexibility of the final design, meaning it can be used in more cases and allows for a larger design space exploration. As is discussed later, this may affect the PIF field size. Furthermore, it is important to note that the number of integer bits used in the fixed point representation is an input to the decoder, not a parameter. This means that the same hardware can be used to compute with a different number of bits for the fraction and integer parts. While this complicates the design, it makes the hardware significantly more flexible.
The equations to compute the PIF fraction and exponent field lengths are repeated below for convenience, in (5.2) & (5.3) respectively, where $N_{posit}$ is the total bit width of the posit number, and $es$ the maximum length of the exponent field. When also encoding fixed point numbers into the PIF, it may be necessary to extend the fraction and exponent fields in order to accurately represent the fixed point numbers.

$$w_{F-posit} = N_{posit} - (3 + es) \tag{5.2}$$

$$w_{E-posit} = 1 + es + \lceil log_2(N_{posit} - 1) \rceil \tag{5.3}$$

For a fixed point number of width $N$, one bit is used for the sign and the first set bit is used as the implicit bit of the significand. This means that the first set bit after the sign determines the magnitude of the exponent when represented in the PIF and is not part of the fraction. This makes the required fraction width $w_{F-fixed\,point} = N_{fixed\,point} - 2$, and the complete equation to determine the PIF fraction width as in (5.4) where $N$ describes the width of the subscripted input type. For (8,1)-posit and 8 bit fixed point, for example, this would make $w_F = MAX(4_{posit}, 6_{fixed\,point}) = 6$ bits, as required by the fixed point number system. The number of bits used to represent the integer part of the fixed point number does not influence the fraction in PIF representation, but will affect the length of the exponent. Effectively, if the fraction size is increased to accommodate the fixed point numbers, this means the significand multiplier needs to

be made larger as well. This increases the hardware area of the PEs.

$$w_F = MAX(N_{posit} - (3 + es),\ N_{fixed\,point} - 2) \tag{5.4}$$

The maximum exponent value required to represent a fixed point input in the PIF depends on the number of bits used to represent the integer part of the number. The representable range of a fixed point 2s complement number is $[-2^I, 2^I - 2^{-F}]$, where $I$ is the number of integer bits, $F$ the number of fraction bits and the total length of the number including the sign is $1 + I + F$. The ULP for this number system has a value of $2^{-F}$ and the number with the largest magnitude is $-2^I$. This makes the required exponent width, including its sign, as in (5.5). The reason for rounding down the logarithm of $I$ while rounding up the logarithm of $F$ originates from the asymmetric range of 2's complement integers: $-4$ can be represented using 3 bits, while $+4$ requires an extra bit.

$$w_{E-fixed\,point} = 1 + MAX(\lfloor log_2(I) \rfloor + 1,\ \lceil log_2(F) \rceil) \tag{5.5}$$

Comparing this equation to (5.3), it can be seen that the fixed point exponent can only be larger than required by the posit numbers when $es = 0$ and either the fixed point integer or fractional part takes up almost all of the bits except the sign. Because $es > 0$ always achieves better inference accuracy as shown in Section 4.3, these are not cases that are of interest for this application. Therefore, the decision is made to design this unit without taking into account the fact that the exponent field may need to be expanded when using $es = 0$. This can always be added later on, if it is needed for a different use case.

#### 5.2.3.3 Hybrid Decoder Design

When encoding a fixed point number to the PIF, it needs to be encoded to a fraction and an exponent value. These values can be determined by locating the most significant set bit: this can be used as the implicit bit of the significand, meaning the remaining bits represent the fraction. The location of the most significant set bit also determines the magnitude of the number, from which the exponent can be calculated.
The assumption here is that the fixed point numbers is in 2's complement representation, as this is the most widely used representation for signed numbers. To determine the fraction and exponent field, the magnitude of negative numbers needs to be determined before the encoding can start. It therefore makes sense to also use this conversion for decoding posit input values, instead of interpreting them as sign & magnitude numbers as in the Deepfloat design: the hardware for the conversion needs to be present for fixed point interpretation anyway.
To detect the first set bit after the sign, a leading zero counter can be used. The number of leading zeros can be used to determine the value of both the fraction and the exponent. By shifting out the bits that are to the left of the first set bit, only the significand remains from which the fraction can be easily determined: the significand is $1.fraction$. This requires the use of a barrel shifter. The exponent depends both on the leading zero count and the number of integer bits in the fixed point representation. In case the first bit after the sign is set in the fixed point representation, the exponent

is $2^{integer\ bits-1}$. When the first set bit is more to the right, the leading zero count should be subtracted from the exponent: $2^{integer\ bits-LZC-1}$. This can be clarified using an example.
For example take a fixed point representation using 3 integer bits and 2 fraction bits, ignoring the sign for simplicity. The number 101.10 can be represented using an exponent and fraction as $2^2 * 1.0110$, where the exponent is indeed $2^{3-0-1}$. When the number does not start with a set bit the leading zero count does influence the exponent value: 010.11 can be represented as $2^1 * 1.0110$ where the exponent is $2^{integer\ bits-LZC-1} = 2^{3-1-1}$. This also works when the first set bit is to the right of the binary point: 000.011 can be represented as $2^{-2} * 1.1000$ where the exponent is $2^{3-4-1}$.
From this description can already be seen that the most important components of the regular decoder can be reused for fixed point to PIF conversion: the leading zero counter and the barrel shifter. The hardware structure for the hybrid decoder can be seen in Fig. 5.8, where the posit and fixed point inputs have the same width for simplicity. The *is FxP* input signal indicates whether the input should be interpreted as a posit ('0') or fixed point number ('1'). In order to accommodate the fixed point input width, the leading zero counter and left shifter need to be widened. To compensate, the input of these modules for regular posit decoding needs to be padded to the correct length. If the fixed point precision is smaller than the posit precision, the size of the leading zero counter and shifter can be reduced, as well as the amount of posit padding.

First, the fixed point datapath is explained, then the impact this has on the posit datapath is discussed. First, the magnitude of the input number needs to be calculated, which is done by selectively computing the 2's complement conversion when the input number is negative. This is the same for both fixed point and posit inputs. The exponent of fixed point input is computed using the equation $exp = integer\ bits - (LZC+1)$, so it would be beneficial to directly compute $LZC + 1$ in order to save an addition. This can be achieved by prepending an extra zero to the leading zero counter input: at the cost of a 1 bit wider leading zero counter, only one subtraction is required rather than two. This should be cheaper in terms of area and delay.
The result of the leading zero counter is shorter than one might expect from the input width: $\lceil log_2(N-1) \rceil$ bits instead of $\lceil log_2(N+1) \rceil$ bits. This is because the extra output bit is only required to represent the leading zero count output in case of an x0 . . . 0 fixed point input, in which case the leading zero count is not used anyway.
The result of the leading zero counter can then be used to shift the input to determine the fraction bits. By performing this variable shift, the fraction bits are located as the most significant bits of the shifter output. During this left shift, unset bits are inserted at the least significant end. This ensures that the length of the fraction remains correct, and does not change its value. Finally, the exponent is determined by subtracting the leading zero count from the input number that indicates how many integer bits are being used in the fixed point number representation. As the exponents in the PIF are biased, the bias is then added to the exponent.
The range of fixed point numbers is asymmetrical, which makes that the smallest representable number needs to be handled separately. This number has a value of $-2^{integer\ bits}$, with binary representation $10 \dots 0$. As this is equal to one of the exceptions of the posit numbers, the same hardware can be used to detect it. The exponent

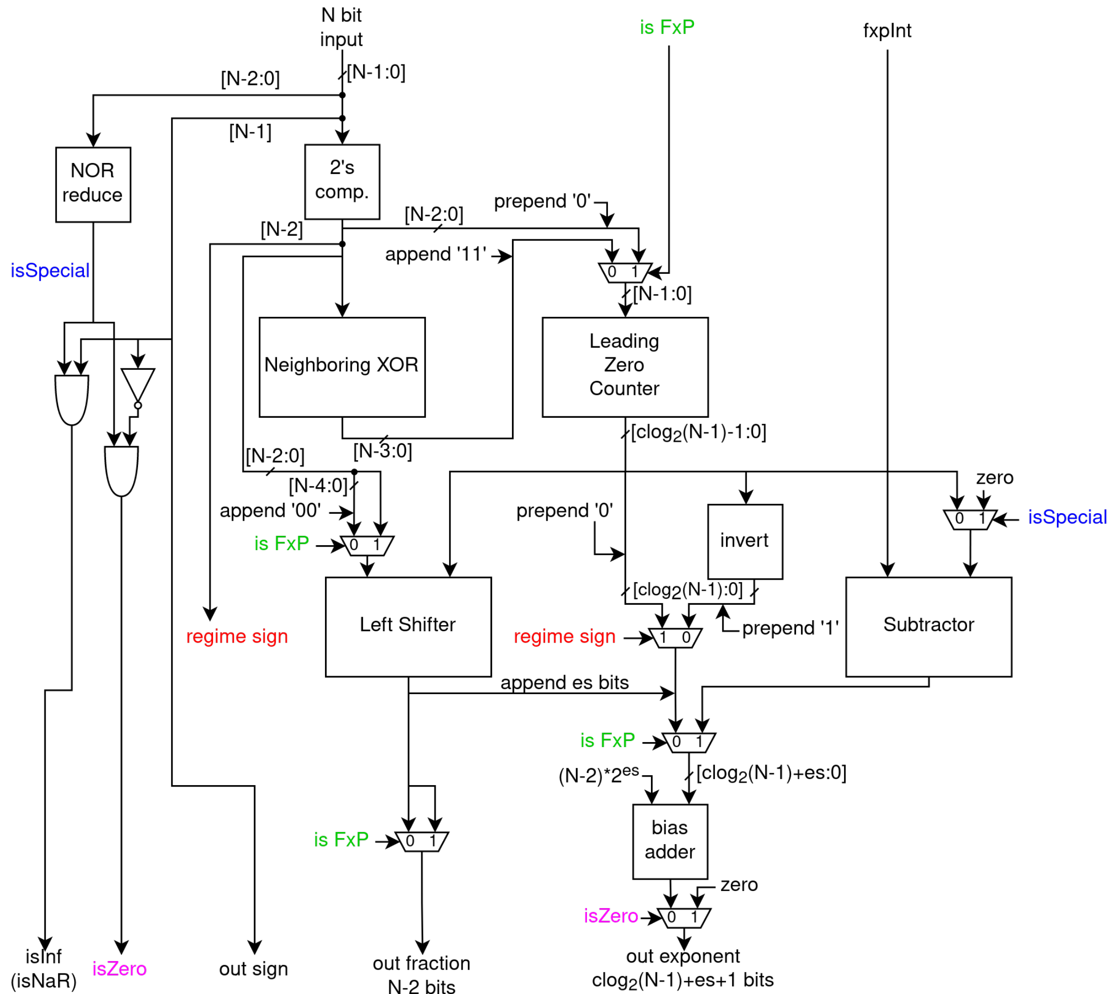



Figure 5.8: Hybrid decoder design that can convert both posits and fixed point input numbers to the PIF. Posit and fixed point precision is assumed to be equal.

value for this number should be *integer bits* + *bias*: this can be achieved by setting the input to the subtractor to zero, as in Fig. 5.8.

As for the posit dataflow, the neighboring XOR of the 2's complement of the input is computed, as discussed in Section 5.2.3.1. The length of this signal is $N-2$, while the input to the leading zero counter has a length of $N$ bits to accommodate the fixed point inputs. Therefore, the neighboring XOR signal is padded with set bits to make it the correct length: zeros are being counted, so the set bits will not affect the result of the leading zero counter.

In the regular posit decoder, the input to the shifter has a length of $N-3$: the sign and first two regime bits do not need to be shifted out as their location is known beforehand. As the shifter has a length of $N-1$ for the fixed point numbers, only excluding the sign, the posit input to this module again needs to be padded. Here, it is padded with unset bits: the shifter also shifts in unset bits in order not to affect the result. However,

instead of increasing the shift amount by 2, this padding can be done for free.
The output of the leading zero counter can be used the same as for the regular posit case, except that extra multiplexers are required to ensure that the bias is added to the correct exponent. The es bits obtained from the shifter are appended before the bias is added to the exponent: the bias adder needs to be wide enough to add the complete bias in this case due to the fixed point dataflow, and this approach simplifies the indexing of the results.
The set of multiplexers takes care of correctly indexing the shifter output to determine the final fraction bits. For larger es values, the output of the shifter may need to be extended with zeros when decoding posits to fill the extended fraction field of the PIF. When encoding fixed point inputs, the least significant bits may not be required to correctly compute the fraction, as there are no es bits in the shifter output, and would result in a fraction that is too long. This is all solved with the set of multiplexers before the fraction field output.
As mentioned, the assumption in Fig. 5.8 is that the input widths of posit and fixed point numbers is equal. When using shorter fixed point numbers, the width of the shifter and leading zero counter also changes, as well as the required padding to all the signals. From the description above it should be clear what these changes entail, so they are not further discussed. The developed hardware description in SystemVerilog accepts the fixed point precision as a parameter, so different precision results will be discussed.

### 5.2.4 Quire to Fixed Point

After the accumulation is finished, an activation function will be applied to the quire result. As the activation function expects a fixed point input, the quire needs to be converted to this format as well. The quire is basically a wide fixed point number, so this can be achieved by correctly indexing the result. However, the number of integer bits used in this representation should again be variable, to allow a trade-off to be made between the accumulated range and accuracy. Therefore, a shifter needs to be used to access the correct bits of the quire. How this can be achieved is explained using an example.
Using the (4,0)-posit as example again, the quire will have $2*(N-2)*2^{es} = 4$ fraction bits, $N + 2*(N-2)*2^{es} = 8$ integer bits, and a bit for the sign, as shown in Table 5.4. The quire will be converted to a 6 bit fixed point number. First, the number is padded with extra zero's equal to the maximum number of integer bits in the fixed point representation: $N_{fixed\,point} - 1$. This is to make sure none of the required fraction bits are lost during the shift, and that after the shift the required bits can be easily accessed.
The extended quire is then arithmetically shifted to the right by an amount equal to the number of integer bits desired in the fixed point result. After this shift, the most significant result bit is at index $quire\ fraction\ bits + N_{fixed\,point} - 1$, where the $-1$ accounts for the sign in the result. As the required bits are now always in the same location, they can be accessed by wiring the correct indices of the shifter output to the output signal. This is shown in bold for 6 bit fixed point using 2 and 3 integer bits in Table 5.4.

The bits more significant than the resulting number can be used to detect overflow: they should all be equal to the sign bit in case there is no overflow. The bits that are less significant could be used to round the fixed point result, in case any of them are non-zero. However, this would come at the cost of an adder and a long OR reduction, while the effect on the inference accuracy is expected to be very limited. Therefore the decision is made to just truncate these bits to save on these costs. A proper rounding scheme can always be added if required, or to investigate the impact this would have on the system accuracy and cost.
As the circuit schematic for the quire to fixed point conversion is rather straightforward it is omitted here. The implementation details should be clear from the discussion above.

Table 5.4: Quire to fixed point example using (4,0)-posit quire and 6 bit fixed point.

| Signal | Bits |
|---|---|
| Quire | $si_7i_6i_5i_4i_3i_2i_1i_0.f_{-1}f_{-2}f_{-3}f_{-4}$ |
| Extended Quire | $si_7i_6i_5i_4i_3i_2i_1i_0.f_{-1}f_{-2}f_{-3}f_{-4}00000$ |
| Shifted by 2 | $\mathbf{s}ssi_7i_6i_5i_4i_3i_2\mathbf{i_1i_0.f_{-1}f_{-2}f_{-3}}f_{-4}000$ |
| Shifted by 3 | $\mathbf{s}sssi_7i_6i_5i_4i_3\mathbf{i_2i_1i_0.f_{-1}f_{-2}}f_{-3}f_{-4}00$ |

### 5.2.5 Shorter Quire

The quire size prescribed by the posit standard draft [2] requires that $N-1$ extra bits are used for the integer part of the quire. This is done to ensure that at least $2^{N-1}-1$ products can be accumulated into it without causing an overflow. Taking into account the number distributions of the application as seen in Section 3.3, it seems a bit excessive to use so many bits for the integer part of the quire. Reducing the size of the integer part of the quire would require less registers to store it, and a smaller shifter and adder for the addition of the products. By making the number of integer bits that are used for the quire a parameter in the hardware designs, it can be easily changed for different application. This allows the effects of using a shorter integer part for the quire to be explored. Using less integer bits does increase the chance of overflow, especially for the accumulation of many products. This means it is important to make a proper trade-off between the risk of overflow and the benefits of using a shorter quire. If the integer part is reduced by more than $N-1$ bits, even a single product can cause the quire to overflow.
Aside from reducing the integer part of the accumulator, it is also a possibility to reduce the size of the fractional part. Instead of increasing the probability of an overflow, this reduces the accuracy of the result: if there is a single product with a magnitude that can not be represented in the fractional part of the quire, the result of the computation will no longer be exact. Unless truncation is used, this also means a rounding method would need to be implemented, which would cost hardware and introduce delay, undoing the benefit of using a shorter quire in the first place. As exact accumulation is an important concept in posit arithmetic, the effect of reducing the fraction size of the quire is not investigated here.

### 5.2.6 Pipelining

A technique that is often used to shorten the critical path of a compute module, and with that increase the maximum clock frequency, is to add pipelining to the module. Instead of doing the entire computation and storing the final result in a register during a single clock cycle, extra registers are added in the module that are used to store intermediate signals and results. These extra registers reduce the amount of computation that needs to be finished within a single clock cycle, which in turn increases the maximum achievable clock frequency, at the cost of an increased latency. Increasing the clock frequency is beneficial to maximize the utilization of all the hardware, as shown in Fig. 5.9 with an example.
Say, the first component $A$ takes 6 ns to compute a specific signal required by the second component $B$, which in turn computes for 5 ns before storing its result in a register. If the clock period is 11 ns to allow for all of the computation to finish in a single clock cycle, component $A$ computes for $\frac{6}{11}ths$ of the time, and component $B$ for the remaining $\frac{5}{11}ths$. By introducing an extra register on the dotted line, the clock period can be reduced to 6 ns. This increases the utilization of component $A$ and $B$ to $\frac{6}{6}ths$ and $\frac{5}{6}ths$, respectively.

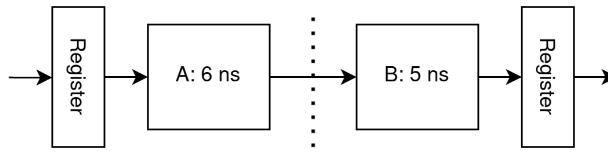



Figure 5.9: Simple pipelining example.

In the PEs the addition to the quire is quite wide compared to the multiplication operands, so introducing a pipelining register between these two operations is probably a good place to start. Whether the sign extension and shifting of the significand product should be placed in the first or second stage depends on which of the two stages has more delay: the goal is to make the delay in both stages as equal as possible. In Fig. 5.10 a simplified schematic of the PE is shown, where the three dotted lines indicate the potential position of the pipelining registers. Of these, the middle one is most expensive: the sign extended significand product contains more bits than the quire, requiring a lot of register in order to be stored. The first option is cheapest, as the exponent and significand product will contain less bits than the quire.
The quire is explicitly not stored in the pipelining register, as it is already stored in the top-level PE to be used in the next cycle. If there is a second copy of the quire in pipelining registers, two separate accumulators would be computed that would then need to be summed after the computation is done, increasing the complexity and delay of the module. On top of that, it would cost extra registers to store both accumulators. By not pipelining the quire value all of this extra complexity can be prevented.
From here, the delay of the first stage can be further decreased by pipelining the multiplier. As discussed in Section 2.2.1.3 this is quite straightforward for an array multiplier. For the addition to the quire, an adder with a shorter delay can be utilized:

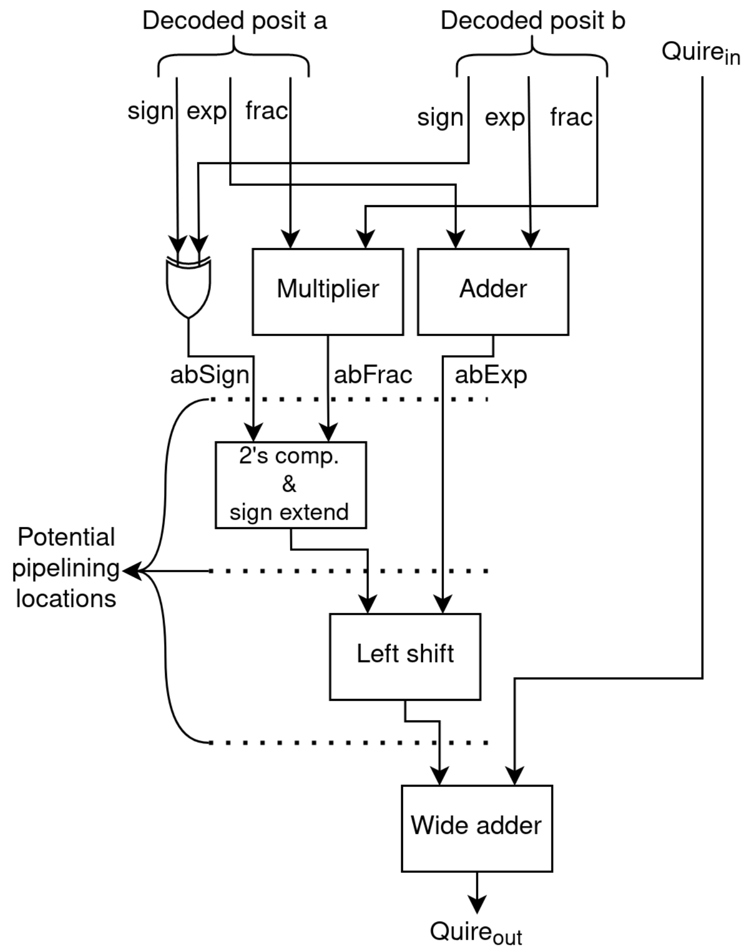



Figure 5.10: Simplified schematic of a processing element, where the dotted lines indicate potential locations to introduce pipelining registers.

[19] gives an example to increase the speed of carry resolution by splitting the quire into smaller blocks and using a flag to indicate whether each block will consume a carry or propagate it through the entire block. This is similar to the discussion of carry-skip adders in [15]. Another approach is discussed in [32], that also splits the quire into multiple blocks. The carries generated by each block are stored during accumulation using some extra bits for each quire block, and full carry propagation is only done after the entire computation is finished. This reduces the maximum length of carry propagation during accumulation, reducing the delay.
In the case of the distributed PE, where the decoding of the posit is done inside the PE, it is probably worthwhile to put the decoding in a separate pipelining stage as well. The same is true for the SA with edge decoding: if there are no registers between the decoding and the inputs to the SA, each PE will pay a speed penalty. This is because the decoding logic would essentially be part of the PEs at the edge of the SA, requiring

a longer clock period to finish the computation.

## 5.3 Verification

After designing and implementing the hardware, its correctness needs to be verified to make sure the behavior of the implemented hardware matches the desired behavior. This can be done by writing a testbench for the module that needs to be tested. In the testbench the module is instantiated, and input signals are applied. In this case, the input data is read from a file, which also contains the expected result of the computation. The expected results from the file are used in the testbench to compare to the output data of the device under test (DUT). If the output computed by the DUT differs from the result that is read from the data file, there is an error in the hardware design.

### 5.3.1 Generating Verification Data

The in- and output data that are used in the testbench to verify the hardware first needs to be generated. For this, the Julia SigmoidNumbers library [69, 70] is used again, just as in Chapter 4. This library is capable of performing the exact accumulation as is done by the designed hardware, and as such can be used to generate the verification data.
Before the data can be generated, it is important to come up with a testing strategy. The naive way to test the correct functionality of a module would be to apply each of the possible input combinations and check whether the computed output is correct. However, because of the width of the data inputs this is not a viable method: the number of input combinations would be too large to test all of them within a reasonable amount of time, so an alternative needs to be found.
A simple approach to test the general behavior of the hardware is to use random input data: this provides an easy way to test the overall correctness, but does not guarantee that the corner cases are tested. The corner cases include for example the addition of a zero product to the quire, addition of NaR, and quire overflow. Testing these cases is important to make sure the functionality is also correct in case a computation is executed that may not be very common: the behavior always needs to be correct.
The approach that is used here to verify the correct functionality of the processing elements is to use 10.000 random posit pairs, the products of which are accumulated. After each addition to the quire, its current value is compared to the expected value. This is useful to pinpoint at which point in the computation an error might be located, instead of only knowing that the final accumulation result is incorrect. Besides these random input data, two corner cases are tested separately: the addition of a NaR value to the quire, and quire overflow. In both these cases a separate flag in the interface needs to be set, rather than performing the regular computation. To make sure this functions correctly, they are tested separately. The addition of a zero product is most likely covered using the random data, as 20.000 numbers are used with a length of 8 or 9 bits.
To generate the random input data two uniformly random, integer numbers from the

range $[0, 2^{posit\ width})$ are generated, excluding the number that represents NaR which has an unsigned value of $2^{posit\ width-1}$. This number is excluded because its functionality is checked separately. The binary representations of these two random numbers are then interpreted as (N,es)-posit numbers using the SigmoidNumbers library [70]. Using the library function for the fused dot product, the product of the two numbers is computed and added to the quire exactly. This is repeated 10.000 times, to test a wide range of inputs.
Effectively, more data is used to verify the correctness of the hardware designs though, as it is parameterized for different precision and exponent sizes. If an error is found in one of the configurations, the same case might also be erroneous in a different number system, and fixing the error also corrects the hardware for the other configurations.
In the SigmoidNumbers library the quire is represented as 64 unsigned integers of 64 bits each, for a total of 4096 quire bits. This size was chosen to be able to accommodate a quire for (64,4)-posit numbers, as well as the quire for smaller posit number systems. The fixed point is placed in the middle, after 32 unsigned integers. In order to interpret the quire, to be able to use it to verify the correctness of the hardware, a function is written that makes a single vector of all the quire bits instead of representing them as 64 separate values. From this single vector of bits, only the quire bits that are required for the used posit number system are read. This correctly sized quire value is then written to a file, along with the two random posit input numbers.
Because the Deepfloat hardware designs interpret posits as being in sign & magnitude representation instead of the more typical 2's complement representation that is used in the SigmoidNumbers library, two separate files are generated: one in sign & magnitude and one in 2's complement representation. This way the same data can be used to verify the correctness of both number interpretations, and the cost of using the 2's complement interpretation can be evaluated. By generating two separate files, there is no need to perform the conversion in the testbench. The quire is always represented in 2's complement representation.
The processing elements that are used in the systolic arrays have already been verified to be working correctly. Therefore, it is only required to verify the correctness of the data flow through the systolic array here. On the other hand, because random input data has been used to verify the processing elements this is a nice opportunity to verify their correctness using some more data. Two more data files containing 10.000 input data pairs and accumulation results are generated, which are then used as input data along with the data file that was generated for the verification of the processing element. Using this data, the data flow through the systolic array is checked, as well as an extra verification of the processing elements.
To make sure the PEs function correctly when using the hybrid decoders that also convert fixed point inputs to the PIF, some reference data is also generated in integer format. This can then be interpreted to any fixed point format of the same precision. Exact accumulation can be achieved by using more bits to represent the accumulator than would be required to store a single product. For the fixed point data 5.000 random data pairs are generated for each precision.
A brief manual check of the generated data confirms that some zero products will be computed, to make sure this functionality is correct. The quire is also checked to both

have a positive and negative value during accumulation, to ensure that both function correctly.

## 5.4 Results: Synthesis & Timing Analysis

For simulating and synthesizing the hardware designs, Xilinx Vivado version 2020.2 is used, and the results in this section target implementation on the Zedboard development board FPGA, using the Zynq-7000 XC7Z020-CLG484-1 SoC. The first step is to verify the functional correctness of the designed modules. This is done by writing a testbench that instantiates the module, feeds it the generated input data and clock signal, and reads and checks the output data. If there is an error in the functional simulation, it means the hardware is not correctly designed or there is an implementation error. When these errors have been solved, the next step is to synthesize the design.
In Vivado, going from the RTL hardware designs to an FPGA implementing the hardware is done in three distinct steps: synthesis, implementation, and generating the bitstream to program the FPGA. During synthesis, the RTL description is translated into a netlist of library modules that can be used during implementation, for example LUTs and DSPs. Synthesis also includes many optimizations, for example to prevent implementing hardware of which the output is not used by any other module. The synthesized netlist can then be implemented, which includes the placement and routing of the hardware modules on the FPGA board. After implementation the exact resource utilization is known, as well as an accurate prediction of the interconnect and logic delays. By generating a bitstream, the implemented design can be programmed onto the hardware. However, this last step is skipped here: everything is simulated to check correctness but not implemented on an actual FPGA board.
The first step before running the synthesis of a design is to check the functional correctness of the hardware description. This can be done by using the previously generated verification data and applying it to the DUT in a testbench that instantiates the module. This step will also point out any syntax errors that may be present in the code. If this behavioral simulation is incorrect the design can be corrected before having performed the synthesis and implementation, which can be time consuming processes. If it is correct, the synthesis and implementation steps can be run.
For synthesis and implementation it is important to include design constraints, for example to indicate the desired clock frequency for the design. These are taken into account during synthesis and implementation, to make sure the implemented design adheres to these constraints within reason: while it would be possible to pose a very short clock period constraint for a large, non-pipelined design, the tool will not be able to achieve non-realistic constraints. On the other hand, using constraints that are too loose may prevent that the design is properly optimized.
After implementation, it is important to check that the implemented hardware still has the correct behavior. To verify this, the same testbench and data can be used to check that the behavior is the same as before the implementation. It is important to note that for every design of which the results are reported here, the full quire precision has been verified to be correct, not only the reduced precision output.
Then, it is also important to the check timing correctness by running a timing simu-

lation. Actually, two different timing simulations are required to fully verify that all timing requirements are met: one to verify the setup time, and one for the hold time. This is to make sure that each signal is constant long enough both before and after each register, to allow the signal value to be stored correctly.
As the final designs will again be incorporated into a larger design, a wrapper-module is made for each of the designs for which results are reported. This wrapper makes sure that the input and output signals to the DUT are buffered, to make the timing results representative of how the final hardware will be used. The resource utilization of these wrappers is not included in the reported results, as they will not be present in the final hardware.
All of the hardware is implemented using only the LUTs that are available on the FPGA, not the DSP blocks. The DSPs could be used to speed up the significand multiplication and the addition to the quire. However, the designed hardware is parameterized, and the parameters affect the width of the operands in both these cases. The addition to the quire can also be too long to fit in the adder that is provided in the DSPs, which makes it less straightforward to use it for that addition. To make sure all of the results can be easily and fairly compared to each other, only the LUT hardware is used for implementation.
Results are shown primarily for (8,1)-posit, according to the results of Chapter 4. However, for the hybrid PEs and final SA implementation, different exponent sizes and number precision are also reported, to evaluate and compare their resource utilization and maximum operating frequency. All the discussed hardware has been verified to be correct for the following set of number systems: (8,es)-posit with $es = \{0, 1, 2\}$ and (9,1)-posit. Results are evaluated based on their LUT and flip-flop (FF) utilization, as well as the maximum clock frequency that the hardware can be operated at.

### 5.4.1 Using Different Decoders

First, the different decoders are compared. The results are shown in Table 5.5, for the standard posit decoder using sign & magnitude and 2's complement interpretation, and for the hybrid decoder that accepts fixed point inputs as well. The hybrid decoder interprets both posit and fixed point inputs to be in 2's complement representation, as this is the standard to represent fixed point numbers. It therefore makes sense to also interpret the posit numbers according to thier original definition, as the 2's complement computation needs to be present anyway. As the number of different possible inputs is limited for small precision numbers, all decoders have been tested exhaustively for each input posit, and fixed point where applicable.
From the number of fraction bits in the PIF, $MAX(N_{posit} - (3+es),\ N_{fixed\,point} - 2)$, an interesting fixed point precision can be found: when using $N_{fixed\,point} = N_{posit} - (1+es)$ there is no need to extend the fraction field to accommodate fixed point numbers. As the fraction length also determines the width of the multiplier, this should have a positive effect on the area utilization of the PE. Therefore, the results for the hybrid decoder for both 8 bit and 6 bit fixed point are shown: to see what the impact is of having to extend the fraction field for 8 bit fixed point. This extension not required for 6 bit fixed point inputs.
As can be seen from these results, the 2's complement interpretation is relatively quite

Table 5.5: Decoder implementation results.

| Number System | Type | # LUTs | # FFs | $f_{max}$[MHz] |
|---|---|---|---|---|
| (8,1)-posit | Sign & magnitude | 19 | 0 | 564.3 |
| (8,1)-posit | 2's comp-lement | 30 | 0 | 429.0 |
| (8,1)-posit | Hybrid, 8 bit FxP | 55 | 0 | 230.5 |
| (8,1)-posit | Hybrid, 6 bit FxP | 60 | 0 | 340.4 |

a lot more expensive than the sign & magnitude interpretation of posit numbers, both in terms of area utilization and clock frequency: the LUT utilization increases by 58%, while the frequency decreases by 24%. This is probably why the Deepfloat hardware interprets posits as being sign & magnitude. Due to the longer width of the leading zero counter and shifter, and the extra subtractor, the hybrid decoder is larger again than the 2's complement posit decoder.
The fact that the 6 bit fixed point hybrid decoder is larger than the one that accepts 8 bit fixed point inputs is an unexpected result. No clear reason for this has been found. However, it does operate at a significantly higher frequency as expected.

### 5.4.2 Quire Conversion

The quire is converted to a shorter fixed point representation after the accumulation is completed. The area utilization and maximum clock frequency of this module is reported in Table 5.6. From this can be seen that the module is quite small. As expected, the hardware to convert the quire to 6 bit fixed point is a bit smaller and faster than to convert to 8 bits.

Table 5.6: Quire to fixed point converter.

| Number System | Output type | # LUTs | # FFs | $f_{max}$[MHz] |
|---|---|---|---|---|
| (8,1)-posit | 8 bit FxP | 28 | 0 | 349.0 |
| (8,1)-posit | 6 bit FxP | 24 | 0 | 371.0 |

### 5.4.3 Processing Elements

In this section the results for the different processing elements are discussed. The different optimizations are added incrementally, such that their effect on the performance can be evaluated. Each time the results are provided for processing elements both including and excluding decoding and quire conversion, indicated with distributed and edge conversion respectively: they are used in either the SA with the conversions done

in each PE, or at the edges. When discussing the distributed PEs that reuse the quire registers to store the converted quire this is mentioned explicitly. This was discussed in Section 5.1.2.
During the design and verification process of the processing element, the use of DSP blocks for the multiplication and addition to the quire was briefly explored. The *use_dsp* attribute was used for this, and it was noticed that having this attribute present in the hardware description reduced the LUT utilization significantly, even when it is set not to use DSPs. Removing the attributes from the hardware description resulted in larger designs. As the smaller designs function correctly, all the reported results are with these attributes in place, set not to use the DSP blocks. Having the attributes present in the hardware description probably triggers an optimization during the synthesis process, but the synthesis logs do not provide any information about this.

#### 5.4.3.1 Regular Posit Decoding

First, the implementation results of the PEs using regular posit decoding are presented. The input posits are interpreted to be in 2's complement representation. These can be used as a reference to see what the impact is of using the PEs for fixed point computation as well. In these results the quire is converted to an 8 bit fixed point number when the computation is finished. As can be seen in Table 5.7, the distributed PE consumes more LUTs and has a longer critical path. This makes sense, as more computations need to be performed than in the edge PE. Due to the fact that extra registers are used to store the converted quire in the distributed PE, the FF utilization is comparable to the edge PE. By reusing the full quire registers to store the converted quire, the FF utilization of the distributed PE can be reduced, as shown in the last row of Table 5.7. This does come at the cost of a slightly higher LUT utilization, as well as a longer critical path.
However, the distributed PE with FF reuse does actually offer a trade-off compared to the edge PE: the distributed PE with separate registers to store the converted quire performs worse in all three metrics compared to the edge PE. While the distributed PE with FF reuse can be operated at a lower maximum frequency and requires more LUTs, it does use less FFs than the edge PE.

Table 5.7: Implementation results of processing elements using regular posit decoding.

| Number System | PE type | # LUTs | # FFs | $f_{max}$[MHz] |
|---|---|---|---|---|
| (8,1)-posit | Edge | 177 | 84 | 98.1 |
| (8,1)-posit | Distributed | 285 | 87 | 83.5 |
| (8,1)-posit | Distributed FF reuse | 307 | 78 | 81.9 |

#### 5.4.3.2 Hybrid Processing Element

Here, the regular posit decoder is replaced with the hybrid decoder that was discussed in Section 5.2.3.3. To see what the effect is of extending the fraction field of the PIF

to accommodate the fixed point numbers, results are also shown for 6 bit fixed point precision as these do not require a longer fraction to be represented in the PIF than (8,1)-posit. The quire results are converted to the same fixed point precision as the input numbers. The results are shown in Table 5.8.

Table 5.8: Hybrid processing element results for different fixed point bit widths.

| Number System | PE type | FxP precision | # LUTs | # FFs | $f_{max}$[MHz] |
|---|---|---|---|---|---|
| (8,1)-posit | Edge | 8 | 259 | 88 | 79.4 |
| (8,1)-posit | Distributed | 8 | 390 | 87 | 64.9 |
| (8,1)-posit | Distributed FF reuse | 8 | 433 | 78 | 61.0 |
| (8,1)-posit | Edge | 6 | 177 | 84 | 98.1 |
| (8,1)-posit | Distributed | 6 | 314 | 85 | 84.8 |
| (8,1)-posit | Distributed FF reuse | 6 | 349 | 78 | 77.1 |

Comparing these results to the ones in Table 5.7, it can be seen that the results for the edge PE using 6 bit fixed point is exactly the same as with regular decoding. This is as expected, as the fraction length is the same and the decoding is handled outside of the PE: nothing changes to the PE hardware. The two distributed PEs do consume some more LUT resources, due to the larger decoder sizes.
When using 8 bits precision fixed point inputs, the fraction length of the PIF needs to be extended by 2 bits. This means that some components in the PE need to increase in size as well, the multiplier having the largest impact on the area. Being able to compute with 8 bit fixed point numbers comes at a cost of 46%, 37%, and 41% LUT utilization increase for the edge, regular distributed, and distributed with FF reuse PEs, respectively, compared to the results reported in Table 5.7.
The overhead from going from 6 bit to 8 bit fixed point input is an increase of LUT utilization of 46%, 24%, and 24% for the edge, distributed, and distributed with FF reuse PEs respectively. The maximum operation frequencies are reduced by respectively 19%, 23%, and 21%.
For the hybrid PEs the FF utilization is again comparable between the edge and distributed case, while the edge PE uses less LUTs and can operate at a higher maximum frequency. The distributed PE with register reuse for the converted quire results in the lowest FF utilization, but has the longest critical path and highest LUT utilization.

**Different Number Systems** For the sake of comparison, the hybrid PE is also implemented for some different parameter number systems, as shown in Table 5.9. What can be seen from the 8 bit posits with different maximum exponent sizes, is that the exponent size has a large impact on the hardware utilization and clock frequency. This is because a larger exponent size increases the dynamic range of the number system, and with that increases the number of quire bits that are required to accommodate exact accumulation. The quire sizes for 8 bit posit are 33, 57, and 105 bits for (8,0)-, (8,1)-, and (8,2)-posit respectively. This directly affects the size of the significand

shifter and adder, as well as the required registers to store the quire, and the size of the quire converter.

Table 5.9: Hybrid processing element results for different parameters, where the fixed point input precision is set to 8 bits.

| Number System | PE type | # LUTs | # FFs | $f_{max}$[MHz] |
|---|---|---|---|---|
| (8,0)-posit | Edge | 164 | 62 | 86.5 |
| (8,0)-posit | Distributed | 305 | 63 | 71.8 |
| (8,0)-posit | Distributed FF reuse | 323 | 54 | 62.0 |
| (8,2)-posit | Edge | 326 | 138 | 73.6 |
| (8,2)-posit | Distributed | 464 | 135 | 58.8 |
| (8,2)-posit | Distributed FF reuse | 528 | 126 | 56.3 |
| (9,1)-posit | Edge | 243 | 97 | 80.9 |
| (9,1)-posit | Distributed | 400 | 98 | 59.8 |
| (9,1)-posit | Distributed FF reuse | 458 | 89 | 58.3 |

An interesting result here is that the (9,1)-posit edge PE consumes less LUTs and can be operated at a slightly higher frequency than the (8,1)-posit edge PE. It does require 9 extra FFs to store the longer quire that has a length of 66 bits. Both distributed PEs do require more LUTs than their 8 bit counterpart, and can be operated at a lower maximum clock frequency.

#### 5.4.3.3 Reduced Quire Size

To reduce the hardware utilization of the processing elements, the size of the quire can be reduced as discussed in Section 5.2.5. In Table 5.10 the results can be seen when the size of the quire is such that a single product can always be exactly accumulated. The $N-1$ extra integer bits that are added in the posit standard draft [2] to prevent overflow are not used in this case, reducing the size of the shifter and adder, and reducing the number of required registers to store the quire. Depending on the application it can be possible to reduce the quire size further, but in that case a single product can already cause an overflow. The reduction applied here should be quite safe for the use in DNNs, and the effect of reducing the quire size further could be easily explored.

Comparing these results to the 8 bit fixed point PEs in Table 5.8 can be seen that the FF reduction is exactly equal to the reduction in the quire size, as expected. The LUT utilization is also reduced, by 15%, 4%, and 7% for the edge, distributed and distributed with FF reuse PEs respectively. Strangely, the maximum clock frequency is slightly reduced for both distributed PEs.

Table 5.10: Hybrid PE results using a shorter quire. The fixed point input precision is 8 bits.

| Number System | PE type | Quire reduction | # LUTs | # FFs | $f_{max}$[MHz] |
|---|---|---|---|---|---|
| (8,1)-posit | Edge | 7 | 220 | 81 | 81.3 |
| (8,1)-posit | Distributed | 7 | 375 | 80 | 64.2 |
| (8,1)-posit | Distributed FF reuse | 7 | 403 | 71 | 58.5 |

#### 5.4.3.4 Pipelined Processing Element

In order to increase the maximum clock frequency the designs can be operated at, pipelining register can be added to the PEs at the cost of a higher FF utilization. Three different options for pipelining the computational PE are discussed in Section 5.2.6. The middle of these options is discarded, as it would require an excessive number of registers to store the extended significand product, for a relatively small difference in delay between the pipelining stages. After implementation the first option turns out to provide the most balanced pipelining stages, at least for (8,1)-posit: the significand multiplication and exponent addition results are stored in registers, before the 2's complement conversion, shifting, and addition to the quire are done in the next clock cycle. In the distributed PEs, the decoder is also placed in a separate pipelining stage. The results are shown in in Table 5.11, where the critical path is through the quire adder in each of the cases.

Table 5.11: Pipelined PE implementation results. The fixed point input precision is set to 8 bits.

| Number System | PE type | Quire reduction | # LUTs | # FFs | $f_{max}$[MHz] |
|---|---|---|---|---|---|
| (8,1)-posit | Edge | 0 | 222 | 112 | 129.5 |
| (8,1)-posit | Distributed | 0 | 340 | 137 | 147.3 |
| (8,1)-posit | Distributed FF reuse | 0 | 410 | 124 | 138.9 |
| (8,1)-posit | Edge | 7 | 225 | 104 | 117.1 |
| (8,1)-posit | Distributed | 7 | 331 | 130 | 149.4 |
| (8,1)-posit | Distributed FF reuse | 7 | 398 | 121 | 141.1 |

When comparing the PEs that use the full quire to their non-pipelined version, the LUT utilization is reduced by the use of pipelining: 14%, 13%, and 5% for the edge, distributed, and distributed with FF reuse PEs respectively. As there is no change to the logic of the PEs, this is probably caused by an optimization performed by the tool during the synthesis or implementation process.
What is strange in these results is that the edge PE with the shorter quire actually consumes a few more LUTs, and it has a lower operating frequency than the edge PE with the full quire. This is not the case for both distributed PEs, yet the reduction in

LUT utilization and increase in $f_{max}$ is also quite small for these.
Due to the extra pipelining registers that are added after the decoder in the distributed PEs, these use more FFs than the edge PE. However, it also makes that the critical path is shorter, resulting in a higher $f_{max}$. This is probably because the registers are placed closer to the compute in case of the distributed PEs, resulting in shorter interconnect delays. The distributed PE with separate registers for the converted quire achieves the highest maximum operating frequency, at the cost of the highest FF utilization. It does require less LUTs than the distributed PE that reuses the registers though.

### 5.4.4 Systolic Arrays

Now that the implementation results of the individual PEs has been discussed, their performance in the SA is evaluated here. This is first done for the SA without pipelining, where the only registers that are present take care of the correct dataflow through the array. Then the pipelining is added to increase the operating frequency.
The size of the SA is set to $9 \times 8$ for all the results presented in this section. This size was chosen as it is the same size as the fixed point SA tiles in the scale-out SA system that this design may be used in, as discussed in Section 3.4. As the designs are parameterized, the results for different array sizes could be easily obtained.

#### 5.4.4.1 Non-Pipelined Systolic Arrays

The results of the SA implementation without pipelining are shown in Table 5.12, both for 8 bit and 6 bit fixed point inputs.

Table 5.12: Non-pipelined $9 \times 8$ systolic arrays, with different input fixed point precision.

| Number System | SA type | FxP precision | # LUTs | # FFs | $f_{max}$[MHz] |
|---|---|---|---|---|---|
| (8,1)-posit | Edge | 8 | 18827 | 5954 | 55.8 |
| (8,1)-posit | Distributed | 8 | 22644 | 5945 | 52.2 |
| (8,1)-posit | Distributed FF reuse | 8 | 24489 | 5223 | 49.6 |
| (8,1)-posit | Edge | 6 | 14302 | 5700 | 73.9 |
| (8,1)-posit | Distributed | 6 | 21824 | 5790 | 69.5 |
| (8,1)-posit | Distributed FF reuse | 6 | 23830 | 5222 | 64.4 |

What can be seen from these results is that the SA with edge conversion has a significantly lower LUT utilization than both distributed PEs, as expected: there are less decoders and quire converters present in the SA. The FF utilization is comparable between the edge and distributed PEs, but is lower for the distributed PE that reuses the full precision quire registers to store the converted quire result. For both fixed point input precisions the edge SA can be operated at the highest frequency.
The increase in LUT utilization from having to increase the fraction field size by 2 bits to accommodate 8 bit fixed point is larger in the case of the edge SA: an increase of

31.6%, against 3.8% and 2.8% for the distributed and distributed with FF reuse SAs respectively. The increase of the fraction field also means the multiplier width increases, which increases the length of the critical path compared to the 6 bit fixed point input implementations.
For the distributed SA with FF reuse the increase of fixed point precision does not affect the FF utilization. This makes sense, because the decoded numbers are not stored in registers, so the increase of the PIF fraction field does not affect the FF utilization.

#### 5.4.4.2 Pipelined Systolic Arrays

The implementation results of the pipelined SAs can be seen in Table 5.13. For these results, the decoding is placed in a separate pipelining stage for both the edge and distributed SAs. This results in a three stage pipeline: decoding, multiplication of significands, and the addition of the product to the quire. Again, the critical path is through the addition to the quire for every implementation.

Table 5.13: Pipelined $9 \times 8$ systolic array implementation results for different bit widths and with reduced quire sizes.

| Number System | SA type | FxP precision | Quire reduction | # LUTs | # FFs | $f_{max}$[MHz] |
|---|---|---|---|---|---|---|
| (8,1)-posit | Edge | 8 | 0 | 19519 | 7848 | 104.6 |
| (8,1)-posit | Distributed | 8 | 0 | 23191 | 9546 | 123.4 |
| (8,1)-posit | Distributed FF reuse | 8 | 0 | 24633 | 8826 | 102.4 |
| (8,1)-posit | Edge | 8 | 7 | 18980 | 7339 | 112.0 |
| (8,1)-posit | Distributed | 8 | 7 | 22495 | 9037 | 127.3 |
| (8,1)-posit | Distributed FF reuse | 8 | 7 | 23975 | 8313 | 112.1 |
| (8,1)-posit | Edge | 6 | 0 | 17372 | 7272 | 126.5 |
| (8,1)-posit | Distributed | 6 | 0 | 20862 | 8814 | 135.2 |
| (8,1)-posit | Distributed FF reuse | 6 | 0 | 22804 | 8244 | 119.8 |
| (8,1)-posit | Edge | 6 | 7 | 16386 | 6768 | 129.6 |
| (8,1)-posit | Distributed | 6 | 7 | 20829 | 8309 | 142.9 |
| (8,1)-posit | Distributed FF reuse | 6 | 7 | 22355 | 7739 | 124.1 |

As the decoding is placed in a separate pipelining stage, the FF utilization of both distributed SAs is larger than for the edge SA. The same trends as for the PEs can be seen in these results: the LUT and FF utilization is reduced by shortening the integer part of the quire, and increased due to having to extend the PIF fraction field to accommodate 8 bit fixed point computation.
The clock frequency is the highest for the distributed SAs with separate registers to store the converted quire results, followed by the edge SA. While in the previous results the distributed PE with FF reuse offered a trade-off due to the lower FF utilization,

this is not the case for the pipelined SAs. The edge SA outperforms the distributed SA with FF reuse in all three metrics, except for a negligible increase in $f_{max}$ in the case of 8 bit fixed point inputs and a reduced quire size.
A trade-off does exist between the edge and distributed SAs though: the distributed SAs require more LUTs and FFs to implement, but can be operated at a higher frequency. For 8 bit fixed point inputs and reduced quire size, LUT and FF utilization are increased by 18.5% and 23.1%, respectively, resulting in an increase in operating frequency of 13.7%. Whether this trade-off is worth it depends on the application the hardware will be used in.

**Different Number Systems** Finally, the results for some different parameters are also presented, in each case for 8 bit fixed point inputs and with reduced quire sizes, and can be found in Table 5.14.

Table 5.14: Pipelined $9 \times 8$ systolic array implementation results for different parameters and with reduced quire sizes, where the fixed point input precision is set to 8 bits.

| Number System | SA type | Quire reduction | # LUTs | # FFs | $f_{max}$[MHz] |
|---|---|---|---|---|---|
| (8,0)-posit | Edge | 7 | 13783 | 5400 | 125.2 |
| (8,0)-posit | Distributed | 7 | 17264 | 7086 | 137.1 |
| (8,0)-posit | Distributed FF reuse | 7 | 18481 | 6376 | 123.2 |
| (8,2)-posit | Edge | 7 | 28836 | 11011 | 93.1 |
| (8,2)-posit | Distributed | 7 | 28441 | 12709 | 103.7 |
| (8,2)-posit | Distributed FF reuse | 7 | 31748 | 11997 | 94.8 |
| (9,1)-posit | Edge | 8 | 20165 | 7920 | 105.5 |
| (9,1)-posit | Distributed | 8 | 24813 | 9741 | 113.9 |
| (9,1)-posit | Distributed FF reuse | 8 | 26618 | 9023 | 99.8 |

For (8,0)-posit and (9,1)-posit the results are comparable to the (8,1)-posit: the edge SA has the lowest LUT and FF utilization, while the distributed PE can be operated at the highest frequency. The distributed SA with FF reuse is outperformed by the edge SA in all three metrics.
For (8,2)-posit the results are slightly different: the edge SA has a slightly higher LUT utilization than the distributed SA. Contrary to the other number systems, the maximum operating frequency of the SA with FF reuse is marginally higher than for the edge SA. This difference can probably be attributed to the length of the quire, as explained next.
What can be seen from the 8 bit posits with different maximum exponent sizes, similarly to the PE results, is that the exponent size has a large impact on the hardware utilization and clock frequency. The reduced quire sizes for 8 bit posit are 26, 50, and 98 bits for (8,0)-, (8,1)-, and (8,2)-posit respectively. This directly affects the size of the significand shifter and adder, as well as the required registers to store the quire,

and the size of the quire converter. A longer quire also requires more interconnect, which can also affect the area utilization and routing, especially in the edge SA where the full precision quire is communicated between the PEs.
An interesting comparison to make is (8,1)-posit with (9,1)-posit, to see how expensive it is to increase the accuracy of posits by 1 bit. With regular decoding the size of the fraction would be increased by 1 bit. However, both also accept 8 bit fixed point numbers in this case, meaning the size of the fraction field in the PIF remains the same. This means the size of the multiplier is not affected. The quire size, however, will increase from 50 to 58 bits. For the edge SA, the LUT and FF utilization increase by 6.2% and 7.9% respectively, and the clock frequency decreases by 5.8%. According to the results in Section 4.3, this would result in a top-5 inference accuracy increase of 0.28%. In the case of the distributed SA, the LUT and FF utilization increases by 10.3% and 7.8% respectively, and the clock frequency decreases by 21.6%.

# 6 Conclusions

Deep neural networks can be used for many different applications, and performing the computations on edge devices poses new challenges in their implementation. Quantization is a technique that is often used to reduce the computational complexity and energy consumption, and to increase throughput. Using low precision fixed point numbers for this is popular, as high inference accuracy can still be achieved using this number system. The recent introduction of the posit number system offers an attractive alternative, as it offers a larger dynamic range and accuracy compared to floating point numbers of the same bit width. The designer is completely free in choosing the required number of bits to represent their data. Recent work using the posit number system in DNN applications has shown that 8 bit posits can achieve similar accuracy as using 32 bit single precision floating point numbers.
In this work the posit number system is used in order to reduce the memory access energy consumption of an inference accelerator. For a scale-out SA design that uses both 16 bit and 8 bit fixed point systolic array tiles, a posit SA is designed that can replace the 16 bit fixed point tiles. In the fixed point system the 16 bit hardware is used to compute the first and last network layers, as the inference accuracy benefits from the increased accuracy provided by the larger bit width in these specific layers. An inference accuracy analysis of replacing the 16 bit tiles with low precision posit numbers is performed, and shows that using (8,1)-posit for the first and last network layer computation results in a top-5 accuracy degradation of 0.24% on the ImageNet image classification task using the VGG16 network. For the hidden layers, 8 bit fixed point numbers are being used. This is in comparison to the same network that uses 16 bit fixed point for the first and last network layer computations.
While using posits increases the computational complexity of the first and last network layer, 8 bit data can now be used throughout the network, instead of also using 16 bit data. It has been shown that memory accesses consume significantly more energy than performing the actual compute, up to 200× more for off-chip memory accesses. By reducing the data precision, this energy consumption can be reduced. Additionally, it also reduces the complexity of the memory hierarchy, as it does not have to accommodate two different data precisions.
To achieve this, the design of a posit processing element is described. The design is parameterized, such that it can be used for different bit widths and with different exponent sizes. This is done to increase the flexibility of the design and to allow for easy design space exploration. To increase the hardware utilization of the system during hidden layer computation, a hybrid posit decoder is designed that can also encode fixed point input data to the posit intermediate format. This way, the posit hardware can also be used to compute with fixed point data, rather than being idle during the hidden layer computation. The posit PE computes exact accumulation, meaning only the final result of a vector dot product is rounded. This increases the accuracy of the

result, and is an important aspect of posit arithmetic.
Three different PE designs are described, that allow a trade-off to be made between combinational and sequential hardware component utilization. One of them is used in a SA design that performs the decoding of posits and conversion of the quire accumulator at the edge of the SA. The distributed SA designs include the decoding and quire conversion in each of the PEs in order to reduce the flip-flop utilization at the cost of using more LUTs.
While this trade-off is shown to work for a non-pipelined SA with a low clock frequency, the efficient pipelining of the distributed SA requires extra pipelining registers after the decoder. This increases the FF utilization to be larger than the edge SA, which means the edge configuration can be implemented using the least hardware. However, the distributed SA can be operated at a higher frequency. If the area constraints allow for it, this can be an attractive option to use.

## 6.1 Future Work

There are multiple different opportunities that can be explored in order to improve the work described in this thesis.
First off, an accurate analysis and comparison of the energy consumption of the fixed point and hybrid compute systems has not been provided in this thesis. The reason for this is that the memory hierarchy is a complicated system, consisting of multiple levels of on-chip and off-chip memory. Making an accurate analysis of the energy consumption in this system would require modeling every component, their interactions, and the computational flow of the target application. As the bit width of the data changes, the size of the memory components may also require changes. The higher complexity of the posit hardware compared to the fixed point hardware should also be taken into account in this comparison. This is an extensive process, that was outside of the scope of this project. Performing this analysis is therefore left for future work.
An interesting exploration that could build on the work described in Chapter 4 would be to make a more fine grained split in which number system and which precision are used in each of the network layers. For a network that is quantized to only fixed point numbers, it has been shown that tuning the number precision for each layer separately can decrease the required precision for some layers without (significantly) reducing the inference accuracy [57, 58]. Using this technique, and extending it to also optimize the posit parameters, could be very interesting to optimize the obtained results. It would also be interesting to see the result of mixing number systems within the same layer, for example only using 8 bit posit for the weights and biases to reduce the required memory size and bandwidth, while using 16 bit fixed point for the activations.
While the hardware designs have been pipelined, there are opportunities to further increase the clock frequency that have not been pursued due to time constraints. Currently the addition to the quire is implemented using a simple ripple-carry adder, which is not especially fast for such a wide addition. Multiple techniques are discussed in the literature to speed up this process, for example in [19, 32]. The significand multiplication can be sped up by pipelining the module.
It can also be interesting to further reduce the quire size, as such large accumulator

values are likely very rarely used in DNN applications. This can further reduce both the combinational and sequential hardware utilization, at the risk of a single product causing an overflow. By implementing the hardware and using it for computations of the target application, it would be possible to track how many quire bits are actually being used in the application. This way a safe and more informed trade-off could be made.

Finally, a logical next step would be to integrate the hardware design discussed in this thesis into a scale-out SA system along with fixed point SA tiles. Implementing it on a FPGA would enable the compute system to be used for DNN inference, and to quantify the achievable performance.

# Bibliography


[1] Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi, "Deep positron: A deep neural network using the posit number system," *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, March 2019.

[2] "*Standard for Posit Arithmetic*, Posit Working Group 3.2-Draft," June 2018.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, June 2012.

[4] F. Emmert-Streib, Z. Yang, H. Feng, S. Tripathi, and M. Dehmer, "An introductory review of deep learning for prediction models with big data," *Frontiers in Artificial Intelligence*, vol. 3, February 2020.

[5] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016. Available: https://www.deeplearningbook.org/ [Accessed: Dec. 22nd 2020].

[6] T. Lee, S. Mckeever, and J. Courtney, "Flying free: a research overview of deep learning in drone navigation autonomy," *Drones*, vol. 5, June 2021.

[7] J. Kocić, N. Jovičić, and V. Drndarević, "An end-to-end deep neural network for autonomous driving designed for embedded automotive platforms," *Sensors*, vol. 19, May 2019.

[8] M. Horowitz, "Computing's energy problem (and what we can do about it)," *IEEE Int. Solid-State Circuits Conf. Dig. of Tech. Papers (ISSCC)*, February 2014.

[9] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," October 2016.

[10] *IEEE Standard for Floating-Point Arithmetic,* IEEE 754-2008. August 2008.

[11] J. L. Gustafson, "Posit arithmetic," October 2017.

[12] M. K. Jaiswal and H. K.-H. So, "PACoGen: a hardware posit arithmetic core generator," *IEEE Access*, vol. 7, June 2019.

[13] H. T. Kung, "Why systolic architectures?," *Computer*, vol. 1, January 1982.

[14] E. L. Oberstar, "Fixed-point representation & fractional math," August 2007.

[15] B. Parhami, *Algorithms and Design Methods for Digital Computer Arithmetic.* New York, USA: Oxford University Press, international 2nd ed., 2012.

[16] Y. Uguen, "High-Level Synthesis and Arithmetic Optimizations,". PhD thesis, Univ. de Lyon, 2019.

[17] Y. Uguen, L. Forget, and F. de Dinechin, "Evaluating the hardware cost of the posit number system," *29th Int. Conf. on Field Programmable Logic and Applications (FPL)*, September 2019.

[18] U. Kulisch, *Computer Arithmetic and Validity: Theory, Implementations, and Applications*, vol. 33. Berlin, Germany: De Gruyter, second ed., 2013.

[19] U. Kulisch and G. Bohlender, "High speed associative accumulation of floating-point numbers and floating-point intervals," *Reliable Computing*, vol. 23, July 2016.

[20] J. Koenig, D. Biancolin, J. Bachrach, and K. Asanović, "A hardware accelerator for computing an exact dot product," *IEEE 24th Symp. on Computer Arithmetic (ARITH)*, July 2017.

[21] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres, *Handbook of Floating-Point Arithmetic*. Cham, Switzerland: Birkhäuser, 2nd ed., 2018. Available from: https://link.springer.com/book/10.1007%2F978-3-319-76526-6#toc.

[22] D. D. Lin and S. S. Talathi, "Overcoming challenges in fixed point training of deep convolutional networks," *33rd Int. Conf. on Machine Learning - Workshop on On-Device Intelligence*, July 2016.

[23] J. Johnson, "Rethinking floating point for deep learning," *32nd Conf. on Neural Information Processing Systems (NIPS), Montréal, Canada*, 2018.

[24] F. de Dinechin, L. Forget, J.-M. Muller, and Y. Uguen, "Posits: the good, the bad and the ugly," *hal-01959581v3*, May 2019.

[25] R. Morris, "Tapered floating point: a new floating point representation," *IEEE Transactions on Computers*, vol. C-20, April 1971.

[26] J. L. Gustafson and I. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing Frontiers and Innovations: an International Journal (SCFI)*, vol. 4, June 2017.

[27] V. G. Oklobdzija, "An algorithmic and novel design of a leading zero detection circuit: Comparison with logic synthesis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, March 1994.

[28] M. R. Pilmer, M. J. Schulte, and E. G. W. III, "Design alternatives for barrel shifters," *Proc. SPIE 4791, Advanced Signal Processing Algorithms, Architectures, and Implementations XII*, December 2002.

[29] A. Podobas and S. Matsuoka, "Hardware implementations of posits and their application in fpgas," *IEEE Int. Parallel and Distributed Processing Symp. Workshops*, May 2018.

[30] T. R. Nicely, "Pentium FDIV flaw," August 2011. Available: https://web.archive.org/web/20190618044444/http://www.trnicely.net/pentbug/pentbug.html, [accessed: Sep. 5th 2021].

[31] M. K. Jaiswal and H. K.-H. So, "Universal number posit arithmetic generator on FPGA," *Design, Automation and Test in Europe Conf. and Exhib. (DATE)*, March 2018.

[32] J. Chen, Z. Al-Ars, and H. P. Hofstee, "A matrix-multiply unit for posits in revonfigurable logic leveraging (Open)CAPI," *Proc. of the Conf. for Next Generation Arithmetic (CoNGA)*, March 2018.

[33] R. Chaurasiya, J. Gustafson, R. Shrestha, J. Neudorfer, S. Nambiar, K. Niyogi, F. Merchant, and R. Leupers, "Parameterized posit arithmetic hardware generator," *IEEE 36th Int. Conf. on Computer Design (ICCD)*, October 2018.

[34] H. Zhang, J. He, and S.-B. Ko, "Efficient posit multiply-accumulate unit generator for deep learning applications," *IEEE Int. Sym. on Circuits and Systems (ISCAS)*, May 2019.

[35] S. Sarkar, P. M. Velayuthan, and M. D. Gomony, "A reconfigurable architecture for posit arithmetic," *22nd Euromicro Conf. on Digital System Design (DSD)*, October 2019.

[36] N. Neves, P. Tomás, and N. Roma, "Dynamic fused multiply-accumulate posit unit with variable exponent size for low-precision DSP applications," *IEEE Workshop on Signal Processing Systems (SiPS)*, September 2020.

[37] H. Zhang and S.-B. Ko, "Design of power efficient posit multiplier," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, March 2020.

[38] J. Hou, Y. Zhu, S. Du, and S. Song, "Enhancing accuracy and dynamic range of scientific data analytics by implementing posit arithmetic on FPGA," *Journal of Signal Processing Systems*, vol. 91, November 2018.

[39] L. van Dam, J. Peltenburg, Z. Al-Ars, and H. P. Hofstee, "An accelerator for posit arithmetic targeting posit level 1 BLAS routines and pair-HMM," *CoNGA: Proc. of the Conf. for Next Generation Arithmetic*, March 2019.

[40] N. Neves, P. Tomás, and N. Roma, "Reconfigurable stream-based tensor unit with variable-precision posit arithmetic," *IEEE 31st Int. Conf. on Application Specific Systems, Architectures and Processors (ASAP)*, July 2020.

[41] R. Murillo, A. A. D. Barrio, and G. Botella, "Customized posit adders and multipliers using the FloPoCo core generator," *IEEE Int. Symp. on Circuit and Systems (ISCAS)*, October 2020.

[42] A. Khan, A. Sohail, U. Zahoora, and A. S. Qureshi, "A survey of the recent architectures of deep convolutional neural networks," *Artificial Intelligence Review 53*, April 2020.

[43] Z. Zou, Z. Shi, Y. Guo, and J. Ye, "Object detection in 20 years: a survey," May 2019.

[44] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, vol. 4, October 1990.

[45] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *3rd International Conference on Learning Representations*, April 2015.

[46] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[47] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. of the IEEE*, vol. 105, August 2017.

[48] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: the Hardware/Software Interface*. Massachusetts, USA: Morgan Kaufmann (an imprint of Elsevier), 5th ed., 2013. Appendix C, concerning GPUs, is available from: https://booksite.elsevier.com/9780124077263/appendices.php [Accessed: Jan. 3rd 2021].

[49] N. P. Jouppi, C. Young, N. Patil, and D. Patterson, et al., "In-datacenter performance analysis of a tensor processing unit," *44th Int. Symp. on Computer Architecture (ISCA)*, June 2017.

[50] Y. Chen, T. Chen, Z. Xu, N. Sun, and O. Temam, "DianNao family: energy-efficient hardware accelerators for mahine learning," *Communications of the ACM*, vol. 59, November 2016.

[51] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: a spatial architecture for energy-efficient dataflow for convolutional neural networks," *ACM/IEEE 43rd Annu. Int. Symp. on Computer Architecture (ISCA)*, June 2016.

[52] M. Naumov, J. Kim, D. Mudigere, S. Sridharan, X. Wang, W. Zhao, S. Yilmaz, C. Kim, H. Yuen, M. Ozdal, K. Nair, I. Gao, B.-Y. Su, J. Yang, and M. Smelyanskiy, "Deep learning training in Facebook data centers: design of scale-up and scale-out systems," August 2018.

[53] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "SCALE-Sim: systolic CNN accelerator simulator," February 2019.

[54] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *38th Annu. Int. Symp. on Computer Architecture (ISCA)*, June 2011.

[55] D. Chen, C. Chou, Y. Xu, and J. Hsue, "BFloat16: The secret to high performance on cloud TPUs," August 2019. Available:

https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus, [accessed: Aug. 27th 2021].

[56] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet classification using binary convolutional neural networks," August 2016.

[57] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, "Fixed point quantization of deep convolutional networks," *Proc. of the 33rd Int. Conf. on Machine Learning*, vol. 38, June 2016.

[58] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: bit-serial deep neural network computing," *49th Annu. IEEE/ACM Int. Symp. on Microarchitecture (MICRO)*, October 2016.

[59] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys*, vol. 48, March 2016.

[60] S. Wu, G. Li, F. Sheng, and L. Shi, "Training and inference with integers in deep neural networks," *6th Int. Conf. on Learning Representations ICLR*, February 2018.

[61] S. H. F. Langroudi, T. Pandit, and D. Kudithipudi, "Deep learning inference on embedded devices: fixed-point vs posit," *1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*, March 2018.

[62] Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi, "Performance-efficiency trade-off of low-precision numerical formats in deep neural networks," *Proc. of the Conf. for Next Generation Arithmetic (CoNGA)*, March 2019.

[63] H. F. Langroudi, Z. Carmichael, and D. Kudithipudi, "Deep learning training on the edge with low-precision posits," July 2019.

[64] J. Lu, C. Fang, M. Xu, J. Lin, and Z. Wang, "Evaluations on deep neural networks training using posit number system," *IEEE Transactions on Computers*, vol. 70, April 2020.

[65] S. Anwar, K. Hwang, and W. Sung, "Fixed point optimization of deep convolutional neural networks for object recognition," *IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP)*, April 2015.

[66] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[67] K. He, X. Zhang, and S. R. J. Sun, "Deep residual learning for image recognition," December 2015.

[68] A. Chen and G. Smith, "Utee: PyTorch Fixed Point Quantization," Available: https://github.com/aaron-xichen/pytorch-playground/tree/master/utee.

[69] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017.

[70] I. Yonemoto, "Sigmoid Numbers for Julia," 2016. Available: https://github.com/interplanetary-robot/SigmoidNumbers.

[71] S. G. Johnson, "PyCall: Calling Python functions from the Julia language," Available: https://github.com/JuliaPy/PyCall.jl.

[72] *IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language,* IEEE 1800-2017. February 2017.