Architectural Exploration of Block Floating Point Support for Matrix-Vector Processor

Antti Tarkka

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.
Espoo 30.7.2025

Supervisor

Assoc. Prof. Marko Kosunen

Advisor

MSc. Farhan Moin

Aalto University
School of Electrical
Engineering

Aalto University
School of Electrical
Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Antti Tarkka

Title Architectural Exploration of Block Floating Point Support for Matrix-Vector Processor

Degree programme Electronics and Nanotechnology

Major Microelectronic Circuit Design **Code of major** ELEC3036

Supervisor Assoc. Prof. Marko Kosunen

Advisor MSc. Farhan Moin

Date 30.7.2025 **Number of pages** 64 **Language** English

Abstract

The demand for hardware acceleration in mobile network baseband processing has increased significantly in the fifth generation (5G) due to higher throughput and coverage requirements. To meet these high requirements, 5G base stations are often deployed with massive multiple input, multiple output (MIMO) antenna arrays. These systems generate vast amounts of data, which increases the challenges of its efficient storage, movement, and processing.

These challenges are especially prominent in the fronthaul interface that connects a radio unit (RU) with a baseband unit (BBU). To reduce the bandwidth required for the fronthaul interface, different compression techniques such as Block Floating Point (BFP) are used. At the receiving end of the interface, the data is decompressed. As a result, the challenge of storing large volumes of decompressed data persists, posing ongoing concerns for data management and system scalability.

The thesis explores an alternative approach by integrating BFP support into an application-specific instruction set processor (ASIP). The goal is to enable the ASIP to process BFP-compressed data directly to reduces the data memory consumption. Two distinct instructions are implemented and evaluated: A stand-alone decompression instruction and a merged decompress and Complex Multiply-Accumulate (CMAC) instruction.

The main objective is to demonstrate the benefits and limits of directly processing compressed data on an ASIP. This offers a promising direction for improving memory efficiency in future mobile network baseband systems.

Keywords 5G, 6G, BFP, ASIP, SIMD, VLIW

Aalto University
School of Electrical Engineering

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Antti Tarkka

Työn nimi Arkitehtuuri tason tutkimus lohkoliukulukutuesta matriisivektoriprosessorille

Koulutusohjelma Elektroniikka ja nanotekniikka

Pääaine Mikroelektroniikkapiirisuunnittelu **Pääaineen koodi** ELEC3036

Työn valvoja Assoc. Prof. Marko Kosunen

Työn ohjaaja MSc. Farhan Moin

Päivämäärä 30.7.2025 **Sivumäärä** 64 **Kieli** Englanti

Tiivistelmä

Viidennen sukupolven (5G) mobiiliverkkojen myötä laitteistokiihdytyksen tarve kantataajuuskäsittelyssä on kasvanut huomattavasti. Tämä johtuu suuremmista datamääristä ja parantuneesta kattavuudesta, jotka molemmat asettavat korkeita vaatimuksia verkon infrastruktuurille. Erityisesti 5G-tukiasemissa yleisesti käytetyt suuret MIMO (Multiple Input, Multiple Output) -antennijärjestelmät tuottavat valtavia määriä dataa, mikä puolestaan luo haasteita datan tehokkaaseen tallennukseen, siirtoon ja käsittelyyn.

Nämä haasteet korostuvat erityisesti fronthaul-rajapinnassa, joka yhdistää radioyksikön ja kantataajuusyksikön. Fronthaul-rajapinnan vaatimaa kaistanleveyttä pyritään pienentämään erilaisilla pakkaustekniikoilla, kuten lohkokohtaisilla liukuluvuilla. Rajapinnan vastaanottavassa päässä data täytyy kuitenkin purkaa. Suurten data määrien purkaminen aiheuttaa ongelmia datan tallentamisessa, lisää haasteita datanhallinnassa ja heikentää järjestelmän skaalautuvuutta.

Tässä diplomityössä esitellään uudenlainen lähestymistapa: lohkokohtaisen liukulukutuen integrointi sovelluskohtaisen käskykannan vektoriprosessoriin. Työn tavoitteena on mahdollistaa lohkokohtaisilla liukuluvuilla pakatun datan suora käsittely vektoriprosessorissa, jonka seurauksena pakattua dataa ei tarvitsisi erikseen purkaa vähentäen merkittävästi datamuistin kulutusta. Työssä suunnitellaan ja arvioidaan kaksi erillistä käskyä: itsenäinen purkukäsky sekä yhdistetty purku- ja kertolasku-summauskäsky.

Tavoitteina on osoittaa pakatun datan suoran käsittelyn edut ja rajoitukset vektoriprosessorissa, ja verrata kuinka hyvin edellä mainitut käskyt saavuttavat asetetut tavoitteet.

Avainsanat 5G, 6G, BFP, ASIP, SIMD, VLIW

Preface

First, I would like to thank Markku Rouvala and Nokia for giving me the opportunity to work on an interesting topic. My gratitude also goes to my supervisor, Farhan Moin, for his helpful guidance throughout this project.

I also wish to thank my unofficial supervisors: Jaywant Kulkarni for this interesting topic and helpful insights, Heikki Kultala for helping me to understand technical details, and Matthew Goode for proofreading the thesis. My thanks extend to my colleagues at Nokia for creating a great work environment.

Finally, I am grateful to my significant other and family for their consistent support during this process.

Otaniemi, 30.7.2025

Antti Tarkka

Contents

Symbols and Abbreviations

Symbols

$\mathbf{A}$	I/Q data matrix
A_m	Amplitude modulation
f_c	Carrier frequency
f_{clk}	Clock frequency
f_s	Subcarrier spacing
$\mathbf{H}$	Channel matrix
N_b	number of beams
N_i	Instruction count
N_{prb}	Number of Physical Resource Blocks in frequency band
N_{rx}	Number of receiving antennas
N_{sc}	Number of subcarriers
N_{prb}^{sc}	Number of subcarriers in a Physical Resource Block
T_{ex}	Execution time
$\mathbf{W}$	Weight matrix
$\mathbf{B}$	Beamformed matrix
ϕ_m	Phase modulation

Abbreviations

1G	First-generation of telecommunication
3GPP	3rd Generation Partnership Project
5G	Fifth-generation mobile network
ALU	Arithmetic Logic Unit
ASIP	Application Specific Instruction-set Processor
BB	Baseband
BBU	Baseband Unit
BFP	Block Floating Point
BTS	Base Transceiver Station
CISC	Complex Instruction-set Computer
CMAC	Complex Multiply and Accumulate
CSI	Channel State Information
CP	Control Plane
CPI	Cycles Per Instruction
CU	Central unit
DAC	Digital-to-analog Converter
DFE	Digital Front-End
DM	Data memory
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
DU	Distributed Unit

EVM	Error Vector Magnitude
FFT	Fast Fourier Transform
gNB	gNodeB
GPP	General Purpose Processor (GPP)
HW	Hardware
IFFT	Inverse Fast Fourier Transform
IP	Intellectual Property
I/O	Input/Output
IPC	Instructions Per Cycle
I/Q	In-phase/Quadrature
ISA	Instruction-Set Architecture
ISS	Instruction-Set Simulator
L1	Layer 1 (physical layer)
MAC	Medium Access Control
MIMO	Multiple-input, Multiple-output
NOP	No Operation
OFDM	Orthogonal Frequency Division Multiplexing
OFDMA	Orthogonal Frequency Division Multiple Access
O-RAN	Open Radio Access Network
PC	Program Counter
PDCP	Packet Data Convergence Protocol
PM	Program Memory
PRB	Physical Resource Block
QAM	Quadrature Amplitude Modulation
RAN	Radio Access Network
RB	Resource Block
RF	Radio Frequency
RE	Resource Element
RISC	Reduced Instruction-Set Computer
RLC	Radio Link Control
RRC	Radio Resource Control
RTL	Register Transfer Level
RU	Radio Unit
SD	Spatial Diversity
SDAP	Service Data Adaptation Protocol
SIMD	Single Instruction, Multiple Data
SoC	System on Chip
SRAM	Static Random Access Memory
SW	Software
UE	User Equipment
UL	Uplink
UP	User Plane
VLIW	Very Long Instruction Word
VLSI	Very Large-Scale Integration

1 Introduction

Hardware acceleration of mobile network baseband processing has become an increasingly important research topic due to worldwide deployments of fifth-generation (5G) networks [1, 2, 3, 4]. Efficient baseband processing hardware is needed to meet the ever-increasing throughput and coverage requirements of the current and future mobile network generations. Going forward, applications such as smart cities, mixed realities, and high-resolution video streaming, will increase the demand for reliable and fast connections [5, 6].

Due to increased throughput requirements, 5G base stations are often deployed with massive multiple input, multiple output (MIMO) systems, that utilize large antenna arrays to transfer and receive large quantities of data. While this approach significantly enhances network capacity and performance, it also increases the challenges related to efficient movement, storage, and processing of the data [7].

In 5G network architecture, the fronthaul interface connects a radio unit (RU) containing antennas and lower-level processing components, to a baseband unit (BBU), which contains higher-level baseband processing hardware [8]. The amount of data that flows through the interface increases with the number of MIMO data streams that are processed.

Simplifying the RUs is beneficial, particularly because they are typically deployed in remote or space-constrained locations. The increased complexity results in higher size, weight, and power consumption which is not ideal for the deployment. However, the simplification comes at the cost of increased data traffic over the fronthaul interface, as more signal processing must be handled by the centralized BBU [9].

To mitigate the throughput requirements of the fronthaul interface Open Radio Access Network (O-RAN) alliance has proposed compression techniques for the in-phase/quadrature (I/Q) data streams. These include Block floating point (BFP), Block scaling, μ-Law, Beamspace, and Modulation compression [9]. While these techniques effectively reduce the data volume transmitted over the fronthaul interface, the I/Q data needs to be decompressed after fronthaul transmission. As a result, the challenge of storing large volumes of decompressed data persists, posing ongoing concerns for data management and system scalability [10]. Traditionally, decompression of fronthaul I/Q data is performed by dedicated hardware accelerators, after which the decompressed data is passed to other computational units for further processing, requiring significant memory resources [11].

This thesis explores an alternative approach by integrating BFP support into an Application Specific Instruction-set processor (ASIP), allowing compressed I/Q data to be directly consumed for calculation. The thesis evaluates the potential memory savings achieved through this approach and analyses the associated processing overhead.

BFP has been selected as the target compression method, due to its widespread adoption in fronthaul interfaces and its suitability for exploring compressed data handling on ASIPs. The ASIP under modification is designed for vector processing and its architecture combines features of Single Instruction Multiple Data (SIMD) and Very Long Instruction Word (VLIW) paradigms.

An ASIP was selected to realize the direct processing of BFP-compressed data, due to its programmability, which offers greater flexibility in large baseband processing systems compared to more rigid hardware accelerator-based solutions [12]. Two distinct instruction-level implementations are proposed:

1. BFP decompression instruction that is used in parallel with arithmetic instructions.

2. Merged decompress and Complex Multiply Accumulate (CMAC) instruction.

The benchmark application that is used to evaluate these implementations is uplink (UL) beamforming. Efficient beamforming for multiple-user equipment (UE) relies on leveraging large antenna arrays, which results in the transfer of substantial volumes of data over the fronthaul interface [2, 13]. To create realistic processing scenarios, the benchmark applications use test data corresponding to systems with 64 antennas, frequency resources split into 18 physical resource blocks (PRB), and 16 beams. The baseline application used in the comparison gets the data already in decompressed form.

The performance metrics evaluated through benchmarking include data memory consumption, program memory consumption and processing latency. In addition, hardware utilization will be calculated, and maximum clock frequency and impact silicon area will be derived from synthesis results of the proposed implementations.

The scope of this thesis is confined to the architectural modifications required to support BFP compression on the ASIP and their impact on memory consumption and computational latency. The accuracy loss resulting from compression is also assessed using test data with varying signal magnitudes. While some software-level optimizations are applied to enhance performance with the modified hardware, they are not the primary focus of this work. The focus will be on the receiving chain of the base station, commonly referred to as uplink (UL).

The rest of the thesis is structured as follows. Chapter 2 provides foundational background on 5G and 6G mobile networks relevant to the context of this thesis. Chapter 3 focuses on the processing aspects, introducing key concepts related to ASIPs, processor parallelism, and baseband signal processing. Chapter 4 describes the methodology used for benchmarking and the implementation details of the new instructions. Chapter 5 presents and analyzes the benchmark and synthesis results. Finally, Chapter 6 concludes the thesis, summarizing key findings and potential directions for future work.

2 Mobile networks

This chapter explains the key features of modern mobile networks relevant to this thesis. The focus will be on topics that help understand the basic physical layer functionality, multiple antenna systems, beamforming, and why data compression is needed for the fronthaul interface.

Mobile networks have advanced significantly over the years. As the world is increasingly becoming more wireless, the demand for higher speed, increased coverage, greater bandwidth, and cost efficiency continues to rise. The higher demands are a consequence of the increasing quantity of mobile devices and the expanding data consumption of their respective applications. The popularity of mobile networks increases due to their advancements, which creates a feedback loop of ever-increasing requirements [14].

Mobile network development started from 1G in the 1980s which refers to the first generation of telecommunication. Mobile networks started as analog circuit switch networks and evolved through the years to modern high-speed packet-switched digital networks [15]. Each generation from 1G to 5G has improved the quality, reliability, speed, and security of the network by adding new innovative features and technologies. A similar trend will continue with the future generations [14].

The 5G network infrastructure is defined and standardized by the 3rd Generation Partnership Project (3GPP) [16]. The standard defines the air interface, protocols, and network interfaces for 5G networks. The different network components and network architecture are also part of the standard [17].

The Open Radio Access Network (O-RAN) alliance works also on specifications which have a goal of revolutionizing mobile networks, by promoting open interfaces where network components from different vendors can be connected to each other [18]. The aim is to create standards that are consistent with 3GPP architecture, interface specification, and terminology, with the addition of O-RAN-specific features [19].

The O-RAN alliance was formed to combat the increasing maintenance costs of mobile networks for network operators. Nowadays Radio Access Network (RAN) components are still far from open, which limits network deployments that consist of equipment from multiple vendors [18]. In this thesis, the BFP support for the ASIP is implemented following the 9-bit encoding specified by O-RAN [9]. The following sections give context and motivate the need for data compression in the fronthaul interface.

2.1 Mobile network structure

Fig. 1 shows an example of 5G network deployment that contains the main components of the network. When a UE connects to the 5G network it first communicates with the base station, which in 5G is called gNodeB (gNB). The communication between the UE and core network is two-way. From the gNB perspective the path that handles data coming from the UEs is called uplink (UL) and the path that handles data sent to the UEs is called downlink (DL) [15].

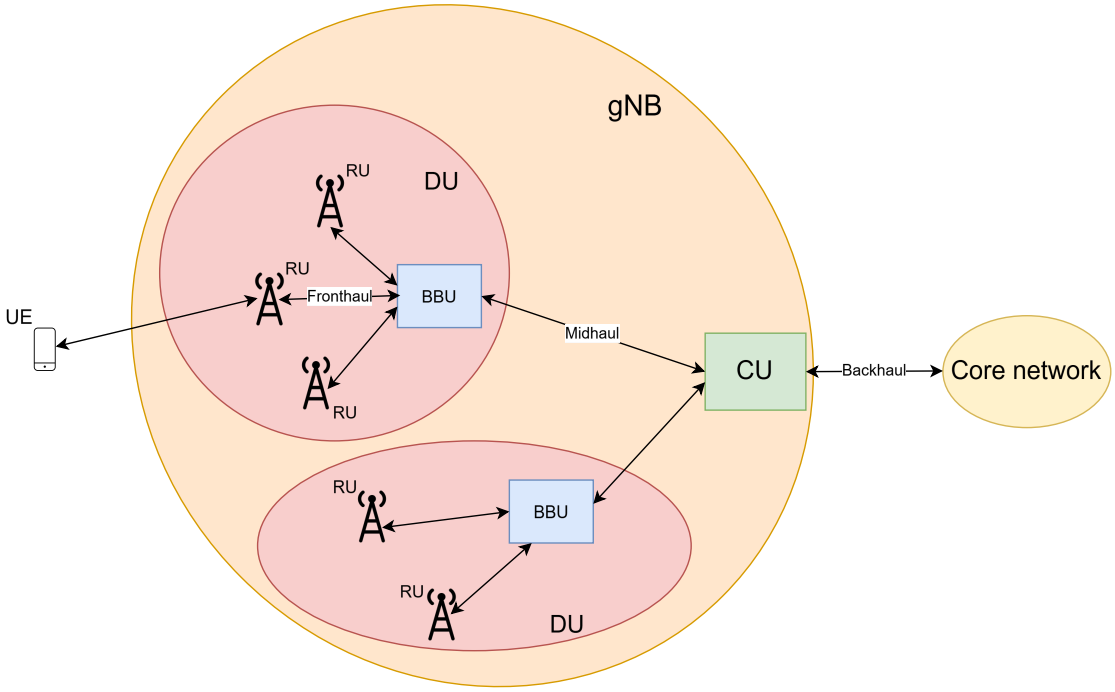

Figure 1: Example deployment of a 5G network.

3GPP has defined the 5G base station (gNB) to be split into two logical units. They are called the central unit (CU) and distributed unit (DU) [17]. The lowest level of functionality in the DU is in the physical layer, also known as Layer 1 (L1), which for example, handles conversions from radio frequencies (RF) to baseband (BB) frequencies and baseband signal processing. In the physical implementation the RF and digital front end (DFE) functionality is usually separated from the baseband processing functionality, which further divides the DU into two distinct units called radio unit (RU) and baseband unit (BBU).

The interface between RUs and BBUs is called the fronthaul interface [15]. Multiple RUs can be connected to a single BBU, and several DUs can be connected to a central unit (CU), which handles higher-layer network processing. The CU is then further connected to the core network via a backhaul interface.

The communication protocols used to transfer the data both ways consist of a user plane (UP) and a control plane (CP). The user plane handles the data packet transmission and the control plane is mainly responsible for connection establishment,

maintenance, mobility, and security.

The radio access network (RAN) protocol stack has three layers. Layer 3 consists of the Radio Resource Control (RRC) protocol and resides completely on the control plane. RRC configures the UE with the parameters required by the other protocol layers to establish and maintain connectivity between the UE and gNB [15]. Layer 2 consists of four protocols that operate both on the user and control plane. The protocols from highest to lowest level are as follows. Service Data Adaptation Protocol (SDAP), Packet Data Convergence Protocol (PDCP), Radio link Control (RLC), Medium Access Control (MAC). These protocols schedule and handle the data coming from and going to the physical layer [15].

The physical layer is the lowest in the RAN protocol stack. It is responsible for managing uplink timing control and synchronization, power control, channel coding, link adaptation, random access procedures, multi-antenna processing, and mapping, as well as beam management [20]. The focus of the later sections of this chapter will be on the physical layer.

2.2 5G wireless communication

The communication between a UE and a base station happens through a wireless channel. The physical signals on the wireless channel are electromagnetic waves propagating through the air. Antennas are used to send and receive wireless signals through the channel.

The three resources available for wireless communication are frequency band, transmit power, and transmission environment. The central challenge in wireless communication is how to utilize these resources efficiently. The wireless channel is almost always imperfect, introducing distortion, interference, and noise to the signal. These non-idealities need to be taken into account both when sending and receiving signals from the channel [21].

To understand the signal processing required to transmit data efficiently it is useful to look into how the data is sent through the wireless channel utilizing the available physical resources. Frame structure in the context of mobile networks describes how the data is organized in terms of the available physical resources in the uplink and downlink wireless channels between the UEs and the base stations [22].

The data frames propagating through the network consist of digital data, but when transmitted through the wireless channel the carrier signals are analog and operate in the radio frequency (RF) domain, which consists of considerably higher frequencies than the baseband frequencies that are used in the digital signal processing. The devices connected to the network use data that is in most cases fully digital. Therefore, there needs to be ways to convert the data to the RF domain and back, and also processing methods that enable efficient recovery of the digital data [15].

2.2.1 Single carrier modulation

To transfer the digital data efficiently through the wireless channel, digital modulation schemes need to be used to encode information to the RF signals. Digital modulation

is a technique, where digital data is mapped to a set of sinusoidal RF carrier waves. The smallest unit of time in sampling these RF signals is called a symbol. The digital data is encoded to the RF carrier signal by modulating the amplitude, phase, or frequency during the symbol duration.

One popular modulation scheme used in 5G networks is Quadrature Amplitude Modulation (QAM). In QAM a quadrature branch is added to the signal. By adding this branch the QAM signal is a sum of two sinusoidal waves with a phase difference of 90°. The two signals are also known as the in-phase (I) component and quadrature (Q) component and their sum can be represented as phasors on a 2-D complex plane due to their phase difference. The benefit of QAM is that it enables both amplitude and phase modulation [15].

Digital data can then be represented in the RF domain with a sinusoidal waveform where both the amplitude and phase are modulated.

$$s(t) = A_m \cos(2\pi f_c t + \phi_m), \tag{1}$$

where A_m is the modulated amplitude of the carrier signal, f_c is the carrier signal frequency, t is time and ϕ_m is the phase shift of the modulated signal [23]. Since the in-phase and quadrature signals are orthogonal the amplitude and phase information can be represented as a complex phasor.

Higher-order QAM systems are used in 5G networks to facilitate the desired spectral efficiency for high-speed data transmissions. They are defined to have 2^{2n} states, where $n \geq 2$ [15]. This set of states is also called a constellation. In the 3GPP standard higher-order modulation schemes are defined from 16QAM to 1024QAM [22]. Since each state represents a specific value, the number of bits in the QAM encoded constellation state is determined by $\log_2(n)$ where n is the amount of constellation points.

Fig. 2 shows a 16QAM constellation and a complex phasor which represents the amplitude and phase modulated signal pointing to a constellation state. From the figure, it can be seen that each of the constellation states has a unique binary value and by changing the amplitude and phase, the phasor can point to another constellation point [21]. A complex phasor $p \in \mathbb{C}$ can be represented as

$$p = A_m \cos(\phi_m) + jA_m \sin(\phi_m) = A_m e^{j\phi_m}. \tag{2}$$

From (2), it can be seen that by modulating the amplitude and phase, the RF signal complex phasor ends up close to a constellation point which represents a unique binary value. Therefore, the digital symbols can be encoded in the RF signal by modulating the amplitude and phase of the sinusoidal in-phase and quadrature (I/Q) signals. The modulated RF signals can then be transmitted to the wireless channel and converted back to digital symbols at the receiving end.

2.2.2 Multi-carrier methods

To reach the high data rate requirements of modern mobile networks, single carrier modulation by itself is not enough. One way to increase the data throughput is to use

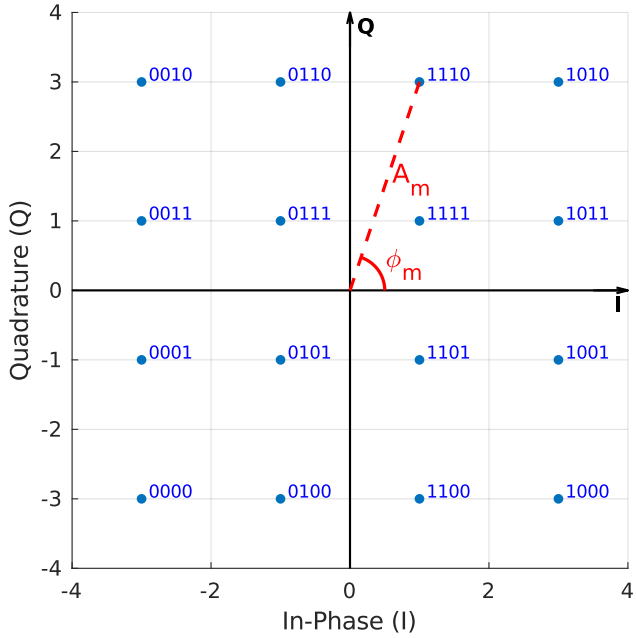

Figure 2: 16QAM constellation.

multi-carrier methods such as orthogonal frequency division multiplexing (OFDM). Multi-carrier methods use multiple RF carriers, also known as subcarriers, to transfer data. Each of these subcarriers are then modulated with a modulation scheme such as higher-order QAM [15].

The idea in OFDM is to have the subcarriers at frequencies that are mutually orthogonal to each other. This is achieved by having each subcarrier frequency as an integer multiple of the highest rate that the signals are captured from the wireless channel [15]. The difference in the frequency of the subcarriers is called subcarrier spacing.

Fig. 3 shows an example of four subcarrier signals with a spacing of 15 kHz in the frequency domain. The subcarriers are modulated with symbols in the form of rectangular pulses that have a period of $\tau = 1/f_s$, where the time is an inverse of the subcarrier spacing frequency. With this type of modulation, the subcarriers in the frequency domain are represented with $\sinh(x)$, which is the hyperbolic sinusoidal function [15].

The orthogonality properties of the subcarrier frequencies are also evident in Fig. 3. When each of the carrier frequencies are at their central frequency all other carrier frequencies have an amplitude of zero. This results in the signals being orthogonal to each other. OFDM takes advantage of this orthogonality in the demodulation process to extract the data from the orthogonal subcarriers. However, the multiple parallel subcarriers do not come for free if looked from the perspective of signal processing. To avoid large amounts of modulators at the transmitter and an equal number of demodulators at the receiver, modern OFDM systems do the modulation in the digital domain [15].

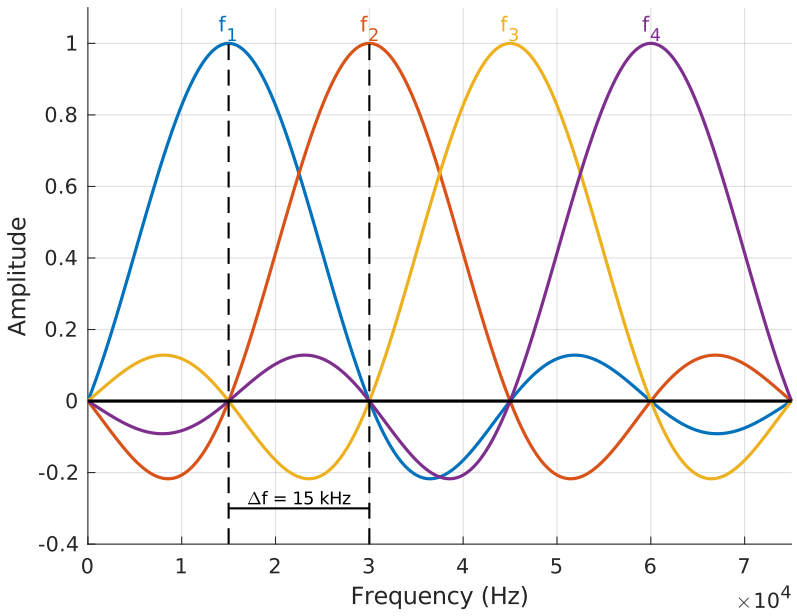

Figure 3: Example frequency domain plot of the 4 subcarriers.

Fig. 4 shows high-level block diagrams of typical OFDM transmitter and receiver chains. In the transmitter chain, the incoming serial data coming from heavier baseband processors is converted to parallel streams, where each set of parallel streams represents one subcarrier. These sets contain a subset of parallel streams and their amount depends on the modulation scheme.

For example, in 16QAM each subset of streams contains 4 streams, one for each bit in the symbol. The mapping stage then converts these substreams to complex numbers representing the amplitude and phase modulation of the subcarrier. Then the modulated symbols are forwarded to Inverse Fast Fourier Transform (IFFT) processing where the frequency domain representation is converted to time domain samples. These parallel samples are then combined into a serial signal that represents the OFDM-modulated signal. This serial signal is then converted to the analog domain with a Digital-to-Analog Converter (DAC) and finally up-converted to RF frequency. The receiver chain reverses this process to obtain the digital data from the carrier signal for further baseband signal processing [15].

ODFM by itself enables transmissions between only two points which is not ideal for multi-user mobile networks. OFDM is used as a basis for a multi-access technique called orthogonal frequency division multiple access (OFDMA), which in 5G is used both for uplink and downlink transmission.

2.3 5G frame structure

5G frame structure has been designed to be more flexible compared to its predecessors to handle a diverse range of mobile devices and applications. Frames are used to organize the data transfer in the uplink and downlink paths of the network. On a

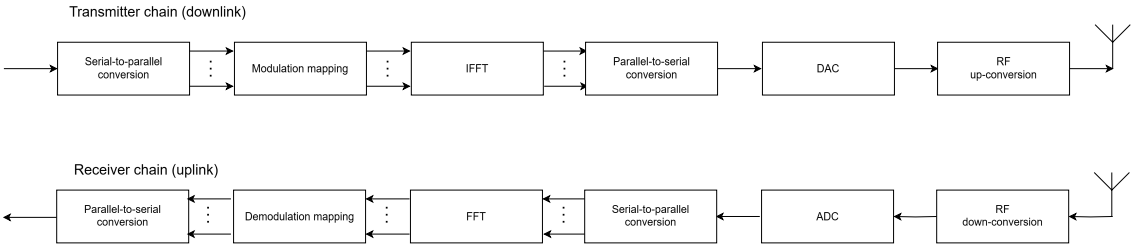

Figure 4: Simplified Block diagram of transmitter and receiver chains of an OFDM system.

carrier wave, there is one set of frames for uplink and another set for downlink [24].

In the 5G standard, downlink and uplink transmissions are organized and scheduled with frames that have a duration of 10 ms. These frames are then further divided to 10 subframes that each have a duration of 1 ms. The subframes are further divided into slots that consist of 12 or 14 OFDM symbols depending on the chosen cyclic prefix. Each of the symbols consists of subcarriers in the frequency domain [24].

The biggest difference in the 5G frame structure is the support for different OFDM numerologies. The numerology defines the OFDM symbol subcarrier spacing and the number of slots in a subframe [15]. The length of the slot also affects the symbol duration since the amount of symbols in a slot is always 12 or 14. As an example with 30 kHz subcarrier spacing the slot length is 0.5 ms (two slots in a subframe) so when there are 14 symbols in a slot the duration of a single symbol is 35.7 μs.

Fig. 5 illustrates the resource grid that is used to transfer the data through the wireless channel. On the vertical axis are the frequency resources that consist of the OFDM subcarriers that all together form the OFDM symbol. On the horizontal axis, the time resource allocation can be seen. The subcarriers in the frequency domain are divided into groups of 12 that are called resource blocks (RB). The amount of RBs depend on the channel bandwidth and subcarrier spacing for example with a 100 MHz bandwidth and 30 kHz subcarrier spacing there are 273 RBs [24].

The basic scheduling unit used is the physical resource block (PRB) which is defined as one RB over one OFDM symbol duration in the time domain [15]. Many of the baseband processing algorithms such as beamforming, are processed on PRB granularity since each of the subcarriers in a PRB share the parameters from the channel. For the same reason, the I/Q data compression methods also compress individual PRBs [9].

Each square of the grid represents a resource element [24]. The resource elements (RE) contain the actual data, for example when 16QAM modulation is used each RE contains 4 bits of the transmitted data. To summarize, the total amount of data that can be transferred within a frame depending on the RE modulation scheme, available bandwidth and the chosen OFDM numerology. These parameters are then used to modify the frame structure for different use cases.

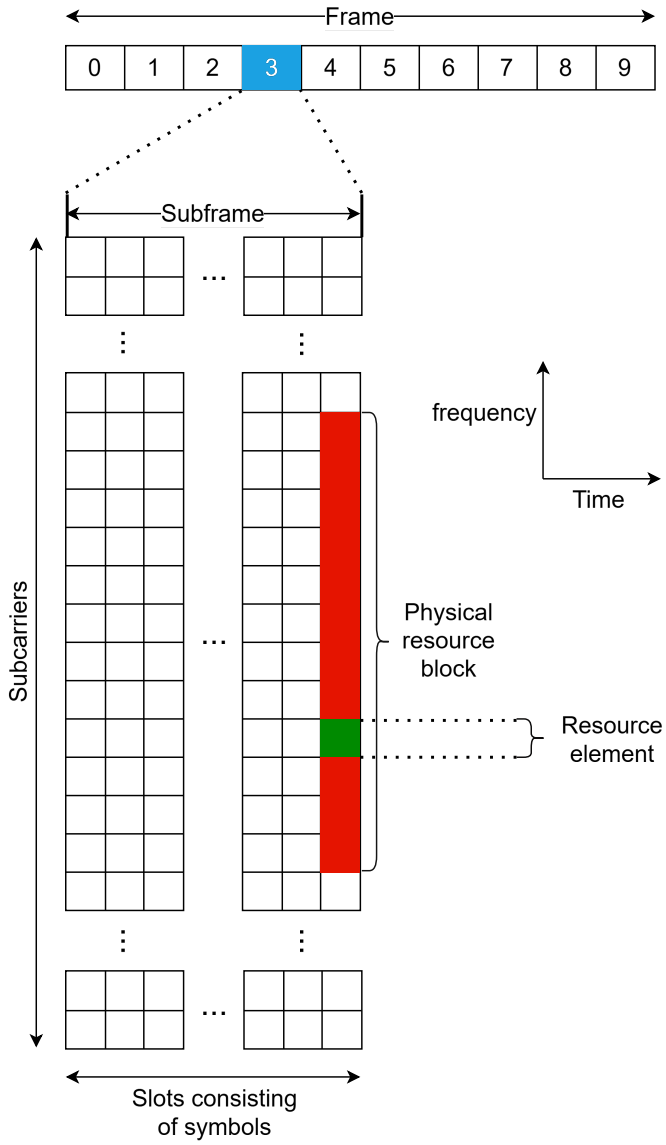

Figure 5: 5G time and frequency resource grid.

2.4 Multiple antenna systems

One of the key technologies to support the increasing capacity demands in mobile networks is the utilization of multiple antenna systems. These systems utilize multiple antennas, together with signal processing to transmit and receive data at high rates.

Multiple antenna techniques can be divided into three main categories [15]. Spatial Diversity (SD) techniques, provide protection against signal fading by combining signals that are unlikely to suffer fading simultaneously. Spatial multiplexing, uses multiple input, multiple output (MIMO) antenna arrays to enable transmission and receiving of multiple data streams using the same time or frequency resource to improve spectral efficiency. The third category is beamforming with multiple antennas. In beamforming at the transmitting end, the transmission power is increased in

specified directions, and at the receiving end, the antenna directivity is focused towards incoming traffic, increasing the gain of the incoming data.

This section briefly introduces beamforming and how it can be realized with MIMO systems. The focus will be on the receiver since the benchmark application introduced later is a part of the uplink beamforming. To understand beamforming it is useful to first look at how MIMO systems operate.

As an example let's consider the 3 x 3 MIMO channel illustrated in Fig. 6. At the transmitter side, the input data stream is divided into three substreams with a serial-to-parallel converter. These substreams are then encoded and used to modulate three different carrier waves. The carrier waves are then fed to the three transmitting antennas. At the receiver, each of the receiving antennas pick up all of the transmitted substreams. Each received substream has a different propagation path through the channel in relation to the receiving antennas, since each of them is transmitted from different points in space. The channel that is formed between the transmitter and receiver can be represented by a channel matrix $\mathbf{H} \in \mathbb{C}^{3\times3}$ [15].

The channel matrix $\mathbf{H} \in \mathbb{C}^{N\times M}$ is defined in general as

$$\mathbf{H} = \begin{bmatrix} h_{11} & \dots & h_{1N} \\ \vdots & \ddots & \vdots \\ h_{M1} & \dots & h_{MN} \end{bmatrix}, \tag{3}$$

where N is the amount of transmitting antennas and M is the amount of receiving antennas. The signal received at a single receiver is defined by the sum

$$R_i = \sum_{j=1}^{N} T_j H_{ij}, \tag{4}$$

where T_j represents the transmitted signal and H_{ij} is the channel coefficient between the receiving and transmitting antennas. The channel coefficients can be obtained for example by sending training sequences from one antenna at a time.

MIMO systems can be scaled up to use hundreds of antennas [25]. These large antenna array systems are called massive MIMO and can be used to do spatial-division multiplexing so that different data streams occupy the same time and frequency resources. A key element to perform efficient spatial-division multiplexing is to have a large array of independently controlled antennas.

The idea of beamforming is to transmit and receive the different data streams to and from the UEs with focused beams. By having more antennas at the base-station, individual data streams can be amplified by forming narrow beams with increased signal power [15]. With beamforming, the carrier waves can be phase-shifted to reinforce each other in wanted directions and destruct each other in unwanted directions. To do spatial-division multiplexing, the antenna array needs to be aware of the propagation channels to the surrounding UEs. The information about the propagation channels is also known as channel state information (CSI). Channel estimation is used to characterize the CSI [26].

At the receiving end of a MIMO device, the incoming signals from the UEs can be amplified by de-spreading weights that can be adjusted to coherently sum signals

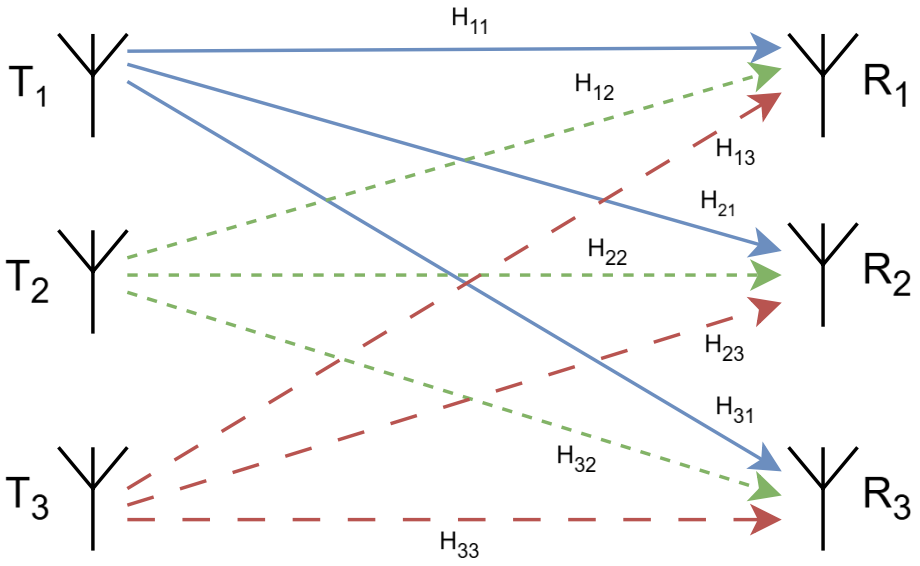

Figure 6: 3 x 3 MIMO channel.

coming from wanted directions. The weights modulate the phase and amplitude of the signals coming to the receiving antennas so that the ones coming from the selected channel directions are at the same phase when summed, which increases the gain of the signal. In modern base stations all antennas receive signals from active UEs simultaneously, and to extract the individual data streams, sophisticated digital signal processing is needed [15].

Massive MIMO system signal processing involves operations on large matrices since the number of antennas can be in the order of hundreds. The channel matrix **H** is used in matrix-matrix, matrix-vector, and matrix inversion calculations during the baseband processing. The goal of the UL baseband processing is to detect, decode, and separate the user data streams so that the data can be forwarded to higher network layers [2].

2.5 Fronthaul interface

The fronthaul interface is defined as the logical link connecting an RU to a BBU. The functionalities that the RUs and BBUs contain, vary depending on the functional split that is in use. The functional split defines the fronthaul interface characteristics such as the amount and what kind of data is sent over the interface [9].

When defining the fronthaul interface for a network deployment there are two competing interests. It is beneficial to keep the RU as simple as possible since a complex RU has higher power consumption and heavier weight, which makes the deployment process more difficult. However, it is also beneficial to have the interface at a higher-level, since with a lower level interface, the throughput increases.

Fig. 7 shows the functional UL split option 7-2 defined in the O-RAN specification [9]. In this thesis, we will be loosely following split option 7-2 in the UL direction,

where the fronthaul interface is between the lower and higher level functionality of the physical layer. The exception that is made is with beamforming. The beamforming will be carried out concurrently when decompressing the I/Q data, and this concurrent computation is the main premise of this study.

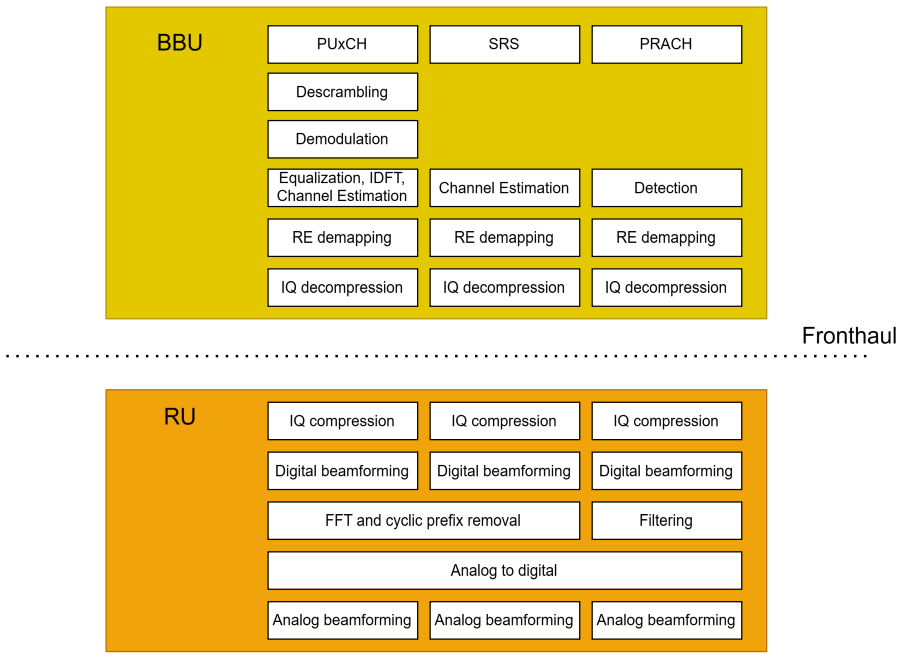

Figure 7: 7-2 functional split presented by O-RAN [9].

In general, the higher physical layer includes UL functionalities such as demodulation, channel estimation, detection, and resource element demapping. Examples of Lower physical layer UL functionalities are the time and frequency domain transformation with fast Fourier transform (FFT) and conversion between the analog and digital domain. However, the location of some physical layer functionalities varies depending on the network use case and component vendor preferences [9].

The fronthaul interface has high throughput requirements. As an example, let's calculate the amount of I/Q data flowing through the interface in the UL per second in the case where the subcarrier spacing is 30 kHz, for a single channel that has a bandwidth of 100 Mhz. The total throughput per second in this case can be calculated as 16(bits) $\times$ 24(I/Q samples) $\times$ 273(PRBs) $\times$ 14(symbols) $\times$ 2 (slots) $\times$ 10 (subframes) $\times$ 100(frames/s) $\approx$ 2.94 Gbps. In this case, assumed that the I/Q data is in an uncompressed format where each I/Q sample is represented with a 16-bit fixed point value [11].

The throughput requirements increase linearly with the number of channels. These multiple channels can be served simultaneously with the help of techniques enabled by mMIMO such as spatial multiplexing and beamforming.

Due to the high throughput of the fronthaul interface, it is usually implemented with optical fiber [27]. The fiber-based solutions might not be always enough to meet the throughput requirements. For example, if there are four channels between the UEs and the RU a 10 Gbps bandwidth optical fiber solution is not enough to

meet the over 11 Gbps throughput requirements. One way to meet the throughput requirements is to compress the I/Q samples. For example, when using the 9-bit BFP compression specified by O-RAN the throughput per second for a single channel reduces to 1.71 Gbps with a compression ratio of 58.33% [11].

3 Baseband processing with ASIPs

This chapter is an overview of Layer 1 (L1) baseband processing, especially on the role of Application-Specific Instruction set Processors (ASIP). The focus will be on topics related to ASIPs, general concepts of computer architecture, and vector processing. The ASIP modified in this thesis is implemented with very long instruction word (VLIW) and single instruction, multiple data (SIMD) architectures, which are introduced concurrently with different forms of computing parallelism. A more detailed description of the binary representation of the I/Q data especially with block floating point is presented. Finally, memory technologies and architectures important for baseband and especially vector processing are also introduced.

3.1 Application-specific instruction set processors

The complexity of digital very large-scale integration (VLSI) systems used to realize the baseband processing of mobile networks has increased significantly due to the realization of 5G networks with high requirements for baseband processing [12]. The baseband processing in mobile networks is realized with complex systems on chip (SoC) devices that can be further divided into intellectual property (IP) blocks that each have a specific purpose inside the chip [28].

To meet the requirements posed by the 5G and future standards, the modern baseband SoCs need to have processing IPs with high performance and low power consumption. One solution is to have hardware accelerators to do the computing. However, the problem with hardware accelerator-based solutions is that they are difficult to design to be adaptable to changes in the mobile network standards. Another problem is that the register transfer level (RTL) designs of hardware accelerators are more prone to sub-micron effects such as noise and quantum effects when fabricated with transistor technologies under 10 nm [12].

Fig. 8 illustrates the performance and flexibility comparison of different processing hardware (HW) implementation methods. On the most flexible side, there are software (SW) implementations for general-purpose processors (GPP) and digital signal processors (DSP). These options in most cases fall short of the performance and energy efficiency requirements of 5G network baseband processing due to their general purpose nature. The limits in performance come from the fact that they are designed to support wide range of applications which makes them non-optimal for processing specific workloads [12].

Application-specific instruction set processors (ASIP) are between the HW-based solutions and SW-based solutions both in flexibility and performance. The motivation behind designing ASIPs is to have a reasonable throughput and area efficiency compared to the HW accelerators without compromising programmability. For ASIPs, the HW and SW are usually designed in parallel, which allows to add more flexibility on the software side or more performance on the hardware side, depending on its use case [12].

The difference between ASIPs and GPPs is in their instructions-set architecture (ISA). ASIPs only contain instructions specific to their application domain, whereas

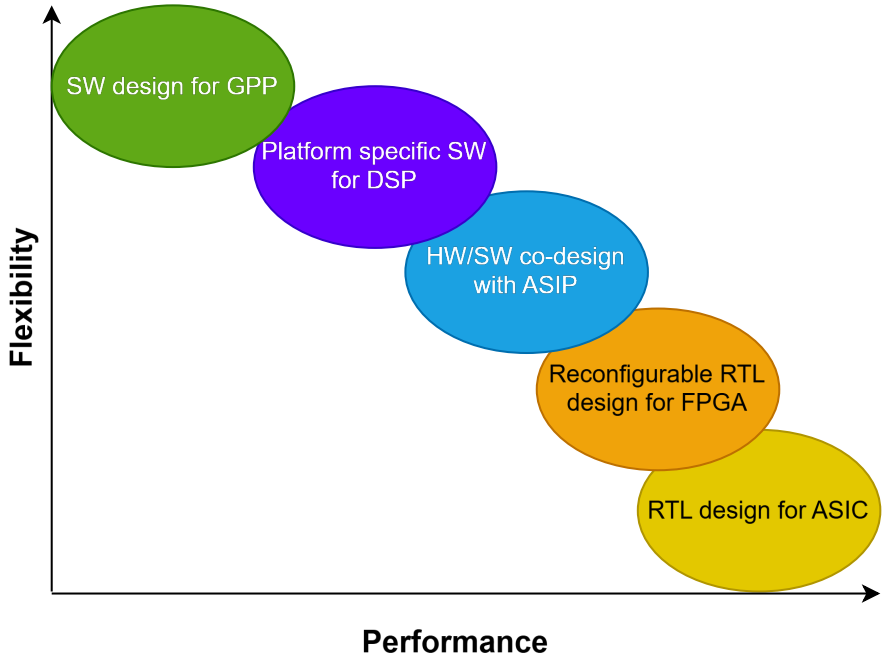

Figure 8: Performance compared to efficiency of different processing implementation methods [12].

GPPs and DSPs usually have more extensive instruction sets and can be used in multiple different domains. When designing hardware for baseband signal processing the functionality needed is mostly known during the design process. Therefore, two major benefits of ASIPs compared to GPPs and DSPs are that the implementation area and energy efficiency of the design can be improved by excluding functionalities that are not required in the application domain of the processor [3].

3.1.1 Instruction-set architecture

The ISA of the ASIPs is the key to its software programmability. An ISA defines the interface between the software and the hardware. Some examples of ISAs are x86, ARM, and RISC-V [29]. From the given examples RISC-V is the only open source one [30]. Instructions are sequences of bits that encode the wanted behavior of the hardware. The sequences are fed to an instruction decoder which extracts control signals from the instruction that tell the processor hardware what to execute.

Processors can be divided into two categories based on their instruction set architecture. Complex instruction set computers (CISC) have instructions with variable length. The benefit of variable instruction length is that common instruction can have short encoding, while the more complex special instructions have longer encoding [31]. The downside is that the instruction decoding becomes more difficult. The x86 is an example of CISC type ISA [31].

The other category is reduced instruction set computers, which have a more simplified instruction set. Instructions usually have a fixed length and simple encoding.

In addition, RISC instructions execute a single operation where as CISC instructions can contain multiple operations, that take several clock cycles to execute. The benefit of RISC is that it simplifies the processor design process due to having simpler instruction encoding and data paths. The ARM and RISC-V categorized as RISC-type ISAs when considering their base instruction set [31].

The RISC-V instruction set has a modular structure, which means that the base ISA is very simple, but there exist extensions that can be added on top of the base ISA. It is quite common to also create custom instructions that can be used to optimize performance in the application domain of the RISC-V processor [32].

The instruction decoder and the data paths of the processor are designed based on the ISA. The ISA defines the control signals and how they are extracted from the instruction encoding. The binary instructions also known as machine language are derived from a human-readable assembly language with the help of an assembler. The assembly language can be directly written by a programmer, but this is considered slow and tedious. Higher-level programming languages such as C can solve this problem, with the help of a compiler that generates the assembly language based on the higher-level description [29].

ASIPs are usually designed on top of a relatively simple processor template [12]. The instruction set is then expanded and modified to meet the requirements specific to the domain that it is used in. These special instructions can then be used to accelerate functionality common to the ASIPs application domain.

ASIP design relies a lot on the tools that are in use. Usually, the processor is described with a higher level processor description language. The instruction set often changes during the design process. Therefore, it is important to have a retargetable compiler, which can generate machine code based on the changes in the instruction set [12].

3.1.2 Processor Micro-architecture

When comparing ASIPs to hardware accelerators with state machine control logic, ASIPs introduce some overhead due to the control logic that fetches and decodes the different instructions. In this section, we have a quick overview of a simple RISC processor architecture as presented in [29], to illustrate the basic building blocks of software programmable processors.

Fig. 9 shows an example implementation of a basic RISC datapath. On the left there is a program counter (PC) register that stores the address of the next instruction that is executed. This address is then sent to the program memory (PM) to load the next instruction. After the instruction is fetched it is decoded. In the decoding process, the different binary fields are extracted from the instruction.

The instruction has an operation code that tells the decoder what fields the instruction encoding contains and what control signals can be extracted. The operation code and the different control signal fields in the encoding are defined by the ISA. The control signals are marked with red in the picture and can be used in different types of multiplexing, addressing, and operation selection.

Another part of the instruction decoding is the operand selection of the instruction.

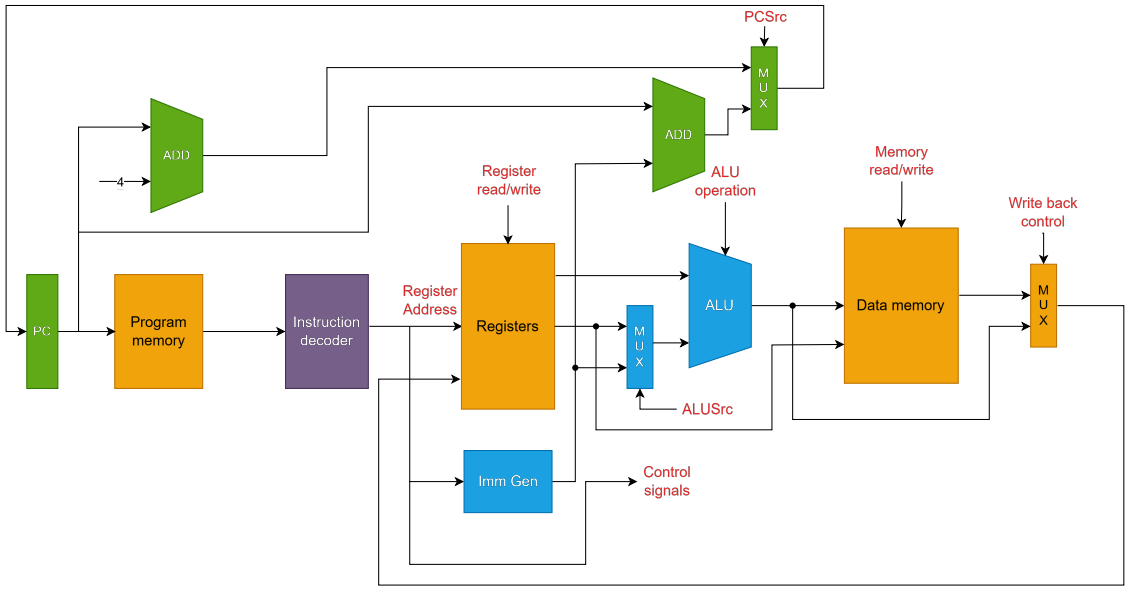

Figure 9: Datapath of a simple RISC processor [29].

The operands are fetched from the register file which contains memory elements that can be accessed quickly. Depending on the instruction an immediate value can be generated based on the instruction encoding. Immediate values are operands that are encoded to the instructions and can then be used either as an operand in calculation or to point to the next instruction address when having a jump or branch instruction. After the decoding process is over, the instruction is executed.

Almost all instructions require some arithmetic operations during their execution. The arithmetic operations are executed in an arithmetic logic unit (ALU). An ALU can execute various digital arithmetic operations which are written back to registers after the execution. An ALU can also be used to calculate an address that is used for storing or loading data to and from the data memory. After the address calculation and the following memory access, the results are written back to the register file so that they can be accessed fast in the following instructions.

When designing ASIPs, one approach is to have a simple RISC core like the one introduced above, to manage the control flow of the programs running on the ASIP and then add the more complex features as a separate functional units [2], [3], [33]. The scalar core can be used for general-purpose processing such as memory address calculations for loading and storing data or do branching address calculations for program memory when there are conditional statements in the software.

3.2 Computing performance analysis

The performance analysis of computing systems can be realized in several ways and the comparison can be performed between various performance metrics [34]. One example of a performance metric is execution time. The execution time is measured

in the seconds that it takes to execute a certain program. The program that is executed is usually a benchmark application that is made to emulate realistic usage of the hardware [34]. Execution time takes into account all the overhead that comes from memory accesses, input/output (I/O) activities, and an operating system if one is used [29].

Almost all digital processing units use a clock that defines when actions take place in the hardware. A clock period is the time it takes for a single clock cycle to complete. It is common to measure the clock performance as clock frequency, which is the inverse of a clock period. The clock frequency is not all that matters since the execution time also depends on the hardware and what kind of actions it does in a single clock period.

For processors, these actions rely on the ISA and how many instructions per clock cycle the hardware architecture allows to execute. The average clock cycles per instruction defines a metric called cycles per instruction (CPI). The inverse of the CPI is instruction per cycle (IPC) which is defined as the amount of instructions executed in a single clock cycle. If the IPC takes into account all the overhead in the processor, the program execution time can be represented with

$$T_{ex} = \frac{N_i}{f_{clk} \cdot IPC}, \tag{5}$$

where N_i is the number of instructions in the program and f_{clk} is the clock frequency. From (5), it can be seen that the execution time in processors depends both on the software and hardware. The instruction count can be reduced and the IPC can be increased by optimizing the software for the hardware in use.

The hardware architecture defines the instructions that can be used, the clock frequency and the amount of instructions that can be executed in a single cycle. In the performance analysis of computing systems the execution time is the standard performance metric since it takes into account both the clock frequency and how much processing is executed in a single clock cycle [29].

3.3 Parallel processing

ASIPs are processors that are tailored for specific application domains. If the domain consists of processing large quantities of data, it is necessary to utilize both instruction level and data-level parallelism to obtain optimal performance. This is the case for the vector ASIP modified in this thesis, since it is designed to process large quantities of data with minimal latency.

3.3.1 Instruction level parallelism

Executing instructions one after another is not efficient enough for modern processors. A single-cycle processor does all the processing related to an instruction in a single clock cycle, since it doesn't have intermediate registers to store the values between the different execution stages. When the whole instruction path is executed in a single clock cycle, there needs to be enough execution time in the cycle to execute

the slowest instruction. This reduces the maximum clock frequency of the processor significantly. The more complex and slow instructions the processor has the smaller the maximum clock frequency has to be for the single-cycle processors [29].

Nearly all modern processors include a pipeline to overlap instruction execution for improved performance [34]. A pipelined processor divides the instruction execution into separate stages, where the execution of each stage takes one clock cycle. Fig. 10 shows an example 5-stage pipeline of the simple RISC processor introduced previously. The dotted vertical lines between the stages represent pipeline registers that forward the necessary data to the next stage after each clock cycle. Different stages include instruction fetch, instruction decoding, execution stage, memory access, and finally the write-back stage.

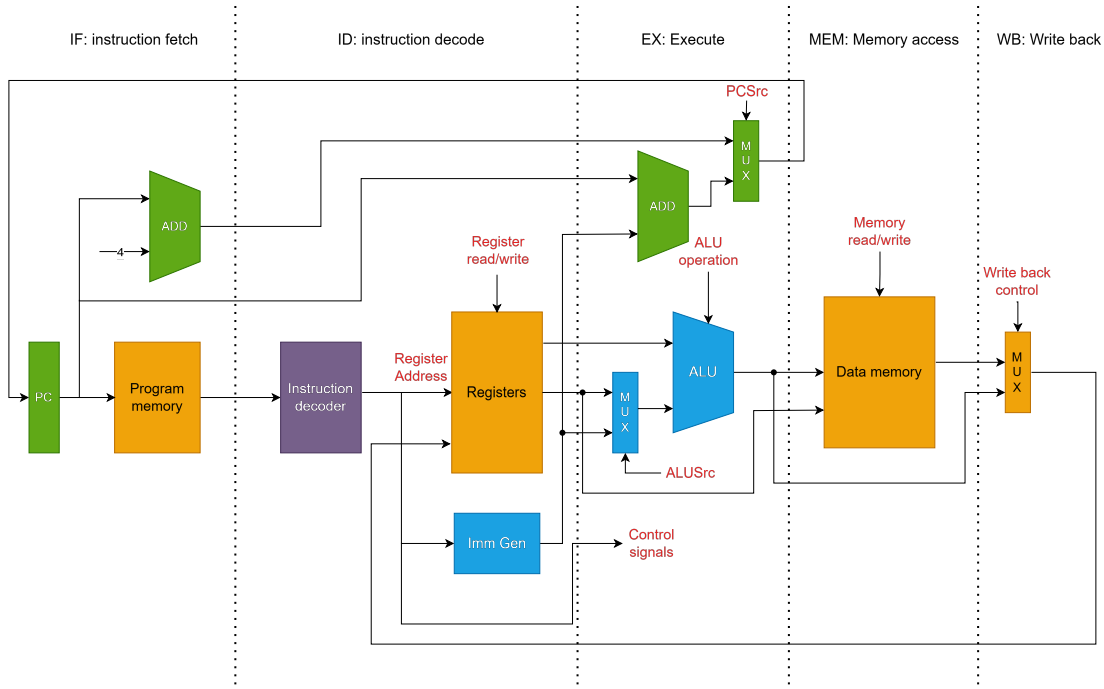

Figure 10: Pipelined datapath of a simple RISC processor [29].

Pipelining introduces some latency in the execution of a single instruction. The maximum clock frequency depends on the execution time of the slowest stage since all the stages need to be ready, for pipeline registers to forward valid data to the next stage. It is nearly impossible to divide the stages to have the same length, which then increases the total time it takes to execute a single instruction.

The benefit of pipelining comes from the increase in throughput. It enables instruction-level parallelism since there can be separate instructions executing in each stage simultaneously. There can be N instructions executing on a N-stage pipeline when the pipeline is fully utilized. However, during the execution of programs there exist three different hazard types that can prevent the full pipeline utilization. These hazard types include structural hazards, data hazards, and control hazards

from which the two first are most common when considering the vector processing application domain.

Structural hazards occur when the pipeline has two or more instructions that need the same resource [29]. The only way to combat this hazard is to design the ISA and hardware architecture so that these situations occur rarely. Data hazards occur when the data needed to execute an instruction is not yet available. One way to reduce data hazards is to introduce bypasses. Bypasses can pass the data from pipeline registers to previous stages rather than waiting for the data to appear in memory or programmer-visible registers [29].

Another approach to increase instruction-level parallelism is to issue multiple instructions in a single clock cycle. One way to do this is to design a processor that uses very long instruction word (VLIW) architecture. VLIW processors use multiple independent functional units. VLIW instructions contain multiple operations that can be issued to the independent functional units at the same time. The advantage of VLIW is that the instruction issue rate grows. However, to keep the functional units busy there needs to be enough parallelism in program code to fill the available operation slots [34].

Fig. 11 shows an example of a VLIW instruction with 5 operation slots. This instruction can issue two memory references, two floating point arithmetic operations, and one integer or branch operation in each clock cycle. This type of instruction architecture is beneficial if the application domain offers possibilities for parallelism. When designing a VLIW architecture, the functional units that enable the parallel execution need to be separate.

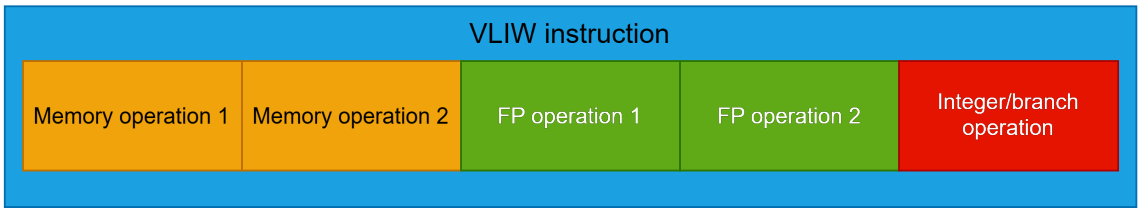

Figure 11: Example of a VLIW instruction with two parallel memory operation slots, two parallel floating point operation slots, and integer or branch operation slot.

There are often gaps in the operation slots even when the code is optimized for the VLIW architecture. This lowers the utilization of the hardware, the instruction throughput, and also increases the program code size if no operation (NOP) instructions are assigned in the place of empty slots. To reduce the amount of wasted bits variable length instructions need to be used. One way to implement variable length instructions is to compress the instructions in memory with a header that indicates how they are compressed and then expand them when they are read into a cache or decoded [34].

3.3.2 Data level parallelism

Data-level parallelism is an approach to increase the processing throughput by operating on multiple independent elements of data simultaneously. Two categories exist to implement data-level parallelism. These categories are multiple instruction, multiple data (MIMD), which is commonly used in multi-core processors, and single instruction, multiple data (SIMD), which is used often in vector processors [29]. Here we will be focusing on the SIMD approach since vector processing can be used to efficiently process the data streams of a massive MIMO system [2].

In SIMD all the parallel execution units are synchronized and respond to a single instruction. SIMD processor operates on vector registers that consist of vector lanes. Each of the lanes contains a data word, which often corresponds to a single value. The amount of lanes and the length of the words depend on the processor architecture. Fig. 12 shows an example of a SIMD data path. Compared to single lane processors there exists copies of the same functional units in the hardware so that the individual vector lanes can be processed in parallel.

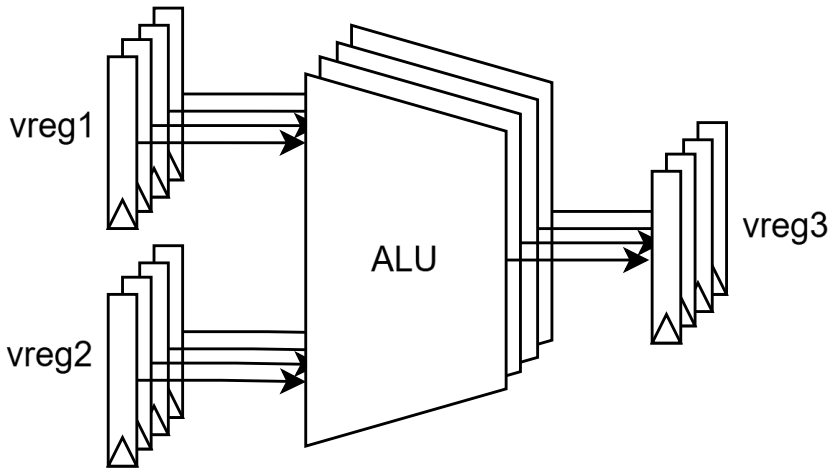

Figure 12: SIMD arithmetic logic unit with four parallel vector lanes.

The wider the vector registers are, the more data can be processed in parallel. Increasing the number of vector lanes also increases the area significantly since the functional units need to be the same on each lane. Also, the memory accesses get more difficult when adding more vector lanes since the memory bus needs to provide enough bandwidth to move the data to and from the vector registers. Designers need to find the optimal amount of lanes based on the application domain so that the utilization of each of the lanes would be as high as possible.

SIMD works best when there exists large arrays of data that need to be operated on. If the application contains a lot of conditional statements the efficiency of SIMD processing decreases significantly, since the calculating the conditional statement stalls the processing of more data elements when compared to scalar processors [29].

3.4 Numerical representations in baseband processing

In mobile network baseband processing the data that is operated on consists of complex I/Q values. Each of the complex values represents the amplitude modulation and phase shift of a single subcarrier wave. The I/Q samples measured by the receiver are continuous complex values, which cannot be represented in the digital domain perfectly. Therefore the I/Q samples need to be represented approximately and the accuracy of the approximation depends on how many bits and what kind of encoding is used.

In this section, we look at some examples of number formats used to represent complex I/Q values in baseband processing. The block floating point encoding, compression and decompression are introduced to give more insight into the architectural modifications implemented later for the vector processing ASIP.

3.4.1 Floating point and fixed point

Complex values can be represented in the digital domain with two fixed point or floating point values depending on the application domain. The complex numbers consist of two fields that are concatenated to one word, one representing the real and one representing the imaginary part. Floating point numbers offer higher dynamic range and accuracy per bit compared to fixed point numbers. However, in many digital signal processing applications fixed point numbers are preferred due to their lower overhead and implementation cost [35].

Floating point representation is a common way of representing real numbers in digital systems [36]. The most popular standard for floating point representations is the IEEE 754 [37]. Floating point calculation results might differ if the encoding format changes between different computing systems. The IEEE 754 standard was created to address this problem by guaranteeing the results of the same operations are identical between computing systems that follow it.

The standard defines floating point formats with 16, 32, 64, and 128 bits [37]. All of the formats can be represented as

$$x_{flp} = (-1)^s \cdot \beta^{e-e_b} * m, \tag{6}$$

where s is the sign bit which is either 0 or 1, e is the exponent, e_b is the exponent bias, m is the fraction that has a value $0 \leq m \leq \beta$, and $\beta = 2$ when the floating point number has a binary base. The exponent has a bias so that numbers close to zero can be represented with more accuracy. Floating point numbers might have several encodings that produce an equivalent result. For this reason, the numbers have to be normalized so that the fraction part is between $1 \leq m \leq \beta$. Due to normalization the first bit before the binary point is always 1 and can therefore be left out of the encoding [36].

Fig. 13 shows an example encoding of a 32-bit floating point number. The most significant bit contains the sign bit which is 0 in this case. The next eight bits are used to represent the exponent. For 32-bit format, the bias in the exponent is 127 so the final exponent is $124 - 127 = -3$. The rest of the bits represent the fraction

part which represents 0.25 in binary format. To that number the hidden 1 before the binary point is added so the mantissa represents the value 1.25. The result can be now calculated as $(-1)^0 \cdot 2^{-3} \cdot 1.25 = 0.15625$ with (6).

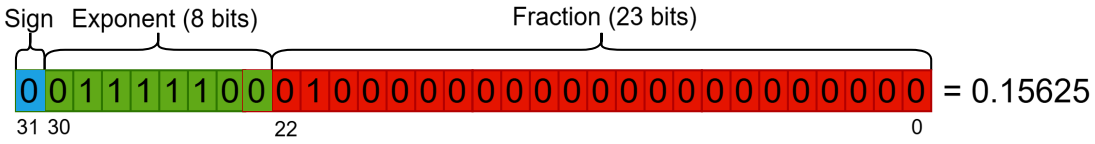

Figure 13: Example encoding of a 32-bit IEEE 754 floating point.

Since floating point numbers approximate real numbers, the results are often not exactly accurate. Due to the inaccuracies, the results would deviate further from a good approximation when operating multiple times on the floating point values, if no rounding is used. For this reason, the IEEE 754 standard also defines how the rounding should be performed [37].

Fixed point numbers are another method to represent real numbers. Fixed point arithmetic is especially favored in digital signal processing (DSP) systems due to them being more efficient in energy consumption, latency, and silicon area compared to floating point. The trade-off with fixed point is that it is less accurate compared to floating point [35].

Fixed-point representation encodes real numbers with a virtual binary point located between an integer and fractional part as demonstrated in fig. 14. The fixed point numbers can be converted to decimal representation with

$$x_{flp} = -2^{m-1} + \sum_{i=-n}^{m-2} b_i 2^i, \tag{7}$$

where m is the number of integer bits, n is the number of fractional bits, and b_i is the binary value of index i.

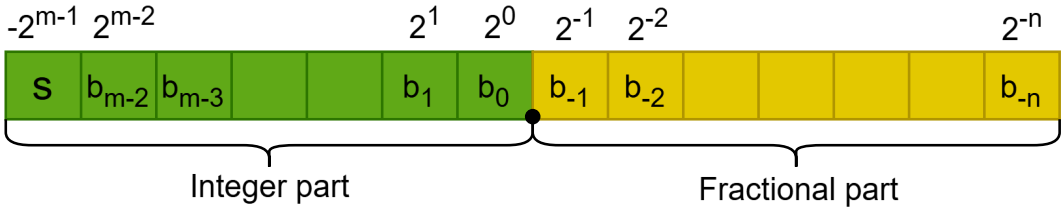

Figure 14: Fixed point encoding.

3.4.2 Block floating point

The O-RAN specification defines several compression methods that can be used to compress I/Q data sent through the fronthaul interface. All of the fronthaul

compression methods compress PRBs containing 12 complex I/Q samples [38]. This section introduces the block floating point (BFP) compression and decompression methods, since it will be used as the example compressed data type to investigate the viability of decompressing data while doing matrix multiplication in parallel.

In this thesis, the 9-bit BFP specified by O-RAN will be used. Fig. 15 shows the encoding for the 9-bit BFP, it consists of a PRB represented with 12 I/Q pairs and a common exponent. Each I/Q sample is represented with two 9-bit 2's complement mantissas. The least significant byte contains a parameter called `udCompParam`, which consists of the common exponent and a 4-bit field that tells which compression method is used for the PRB [9]. The padding also aligns the compressed PRB with bytes, which enables storing the compressed values in byte-aligned memories.

Figure 15: O-RAN 9-bit Block floating point encoding.

Algorithm 1 describes the BFP compression process. First, the maximum value of all imaginary and real samples is determined. Then the exponent is calculated based on the maximum value. This exponent will then be common to all of the I/Q samples in the compressed PRB. Next, all the samples are bitwise shifted to the right by the exponent. And finally, the samples are concatenated to their target length. The output consists of the compressed PRB and the common exponent. [38].

Algorithm 1 Block floating point compression

Input: $PRB[1, ..., 12],\ iqWidth$
Output: $cPRB[1, ..., 12],\ exp$

$maxValue \leftarrow \max(\max(|\Re(PRB)|), \max(|\Im(PRB)|))$
$msb \leftarrow \lfloor \log_2(maxValue) + 1 \rfloor$
$exp \leftarrow max(msb - iqWidth, 0)$
for i = 0 to 12 **do**
    $reShiftRight \leftarrow \Re(PRB[i]) \cdot 2^{-exp}$
    $imShiftRight \leftarrow \Im(PRB[i]) \cdot 2^{-exp}$
    $\Re(cPRB[i]) \leftarrow reShiftRight(iqWidth : 1)$
    $\Im(cPRB[i]) \leftarrow imShiftRight(iqWidth : 1)$
end for

Algorithm 2 shows the BFP decompression process. It is very simple when decompressing to 2's complement representation. The compressed values are bit-wise left shifted by the magnitude of the common exponent to produce the decompressed values [38].

The algorithm has a few extra steps if the decompression targets to sign-magnitude format, such as floating point numbers. For example, if the sign is negative the fraction bits have to be inverted. Also, floating point numbers have a biased exponent and

Algorithm 2 Block floating point decompression

Input: $cPRB[1, ..., 12]$, exp
Output: $dPRB[1, ..., 12]$

1: **for** i = 0 to 12 **do**
2: $\Re(dPRB[i]) \leftarrow \Re(PRB[i]) \cdot 2^{exp}$
3: $\Im(dPRB[i]) \leftarrow \Im(PRB[i]) \cdot 2^{exp}$
4: **end for**

when calculating the exponent, care needs to be taken so that the fraction represents a value between 1 and 2. In this thesis decompression implementations both to 2's complement and sign magnitude are explored.

3.5 Memory for vector processing

Vector processing requires high memory access bandwidth to supply large amounts of data to the vector registers. This section briefly introduces some basic memory technologies and how memory architecture can be designed to provide the needed bandwidth and latency for SIMD processors. In addition, the concept of memory alignment is introduced.

Memory architecture always has a hierarchy, due to fast memory elements being more expensive. The memory elements close to the processor, such as registers, are usually the smallest, most expensive, and fastest. Memory elements further away from the processor consist of cheaper and more inefficient memory elements that can be efficiently scaled to provide additional memory [34].

The fast memory elements are usually built with static random access memory (SRAM) technology. SRAMs typically use six transistors to store a single bit, preserving the stored infromation when it is read. This of course makes the implementation area of SRAM large, especially if millions of bytes need to be stored. SRAM is often used as fast on-chip memory to implement registers and caches, due to its low latency and power needed to retain the charge in standby mode [34].

Larger memory elements used in the processing are usually built with dynamic random access memory (DRAM) technology. To pack more bits per memory DRAMs use only a single transistor to store a bit [34]. Reading that bit destroys the information and therefore it must be restored. This is one reason why DRAM has a longer cycle time. In addition, DRAM needs to be refreshed periodically, which further decreases the latency of the access. DRAM is often used off-chip to facilitate gigabytes of memory, which can be used with the help of SRAM caches, to provide efficient memory hierarchy for complex applications [34].

In SIMD architectures it is common to have the memory divided into banks that correspond to the amount of vector lanes in the architecture. With banked structures, it is possible to fetch elements from different banks in parallel increasing the memory bandwidth significantly [39]. However, to achieve this high performance of banked memories, memory accesses needs to be aligned or there needs to be additional logic that handles unaligned memory accesses [40].

A multi-byte word is aligned when its memory address of the first byte is an integer multiple of the word length in bytes. For example, if a 32-bit word is aligned with addresses $0, 4, 8, ...$ in a byte-addressed memory. In a similar way, if a vector register with 4 elements each 32-bit wide is aligned with addresses that are multiples of $4 \cdot 4 = 16$.

4 Methodology

In this chapter, architectural modifications enabling computation with BFP compressed PRBs are introduced. These modifications are implemented on top of an existing application-specific instruction set processor (ASIP) designed for matrix-vector workloads. The main goal of the modifications is to investigate the viability of directly processing BFP-compressed PRBs without storing the decompressed values in memory.

First, the methodology used in the design process to benchmark and validate the implementations is outlined. Next, two different approaches to implement the BFP support are presented. The first one is a set of stand-alone decompression instructions and the second one is a merged decompress and multiply-accumulate instruction that does the decompression while doing vector lane shuffling. Both of the implementations are benchmarked and validated individually to assess their effectiveness and feasibility in incorporating BFP support into the processor architecture.

4.1 Design process

In this thesis, hardware modifications are implemented using a high-level processor description language provided by an ASIP tool. The processor description language generates a register transfer level (RTL) hardware description, a cycle-accurate instruction set simulator (ISS), and a custom C-compiler for the ASIP's ISA. The ISS simulates the instruction set of the ASIP using a processor model that contains a description of the processor pipeline. Utilizing the ISS is advantageous during the design phase because it significantly reduces simulation time compared to more detailed RTL simulations [41]. Subsequently, the RTL, containing a more comprehensive description of the processor, serves as the basis for synthesis and physical implementation that can be realized with hardware synthesis tools [42].

The processor description language requires a compiler interface description, that generates the custom C-compiler targeting the ASIP architecture, which serves as a critical component for efficient hardware-software co-development. Through this interface, custom compiler intrinsics that seamlessly map high-level software constructs directly to optimized hardware instructions can be defined. Such intrinsics, appearing as standard C functions from a software perspective, enable transparent access to hardware accelerating data paths inside the ASIP. Furthermore, the compiler interface supports the introduction of custom data types and the reassignment of standard C operators for them.

The two implementations are evaluated with a beamforming benchmark application to determine if the decompression introduces overhead in clock cycles and to quantify the data memory savings achieved by directly processing BFP-compressed data. In addition, the implementation is synthesized to evaluate the maximum clock frequency and impact on the silicon area. The performance metrics evaluated through benchmarking and synthesis results include:

1. **Data memory consumption:** Memory allocation for the data during processing.

2. **Processing latency:** Amount of clock cycles it takes to complete the program.
3. **Hardware utilization:** Percentage of available processing units utilized during program execution.
4. **Program memory consumption:** Memory needed to store the benchmark application instructions.
5. **Clock frequency:** Maximum clock frequency that the ASIP synthesized with after the modifications.
6. **Silicon area:** Impact the modifications have on the implementation area.

In this thesis, the ISS is utilized to execute the benchmark applications, and the results are used to evaluate the performance of the implementations. The benchmark applications are written in C and compiled with the custom C compiler provided by the tool. The RTL description is later used to further verify the correctness and to synthesize the processor.

4.1.1 Benchmark application

The application that is used to benchmark the hardware modifications is the matrix multiplication part of UL beamforming, since it is a possible use case for the block floating point processing on the ASIP. Nearly all of the physical layer algorithms needed for massive MIMO baseband processing have multiplications of large matrices in them, increasing the relevance of the application further [43].

The UL beamforming consists of algorithmically modifying the directivity pattern of an antenna array to point it toward a desired direction. In a MIMO communications system, multiple beams pointing in different directions can be achieved. The beams are I/Q data streams coming from the antenna. The directivity pattern of the antenna array can be modified by multiplying the signal streams with beamforming weights. The goal is to choose the weights so that the incoming signals are amplified in the desired directions and attenuated in other directions.

For a MIMO system, the weights can be represented with a matrix $\mathbf{W} \in \mathbb{C}^{N_b \times N_{rx}}$ where N_b is the number of beams and N_{rx} number of receiving antennas. The weight matrix can be calculated based on the channel matrix $\mathbf{H}$ with different methods such as maximum ratio combining (MRC), zero-forcing (ZF), and Minimum Mean-Squared Error (MMSE) [44]. In this thesis, we will focus on the part where the weight matrix is already calculated and multiplied with the I/Q data matrix.

When the MIMO system has N_{rx} receiving antennas and N_{sc} subcarriers the I/Q data matrix is represented by $\mathbf{A} \in \mathbb{C}^{N_{rx} \times N_{sc}}$. The amount of subcarriers is a product between the amount of PRBs the frequency bandwidth is divided to (N_{prb}), and how many subcarriers there are per PRB (N_{prb}^{sc}).

The beams are steered in the desired directions by multiplying each of the individual PRBs with the detection matrix. The detection matrix in the worst-case changes for each PRB, which increases the amount of needed memory to store the data and the amount of processing to determine the weight matrices. In most cases,

however, the weights stay the same for multiple PRBs. For each PRB a result matrix $\mathbf{B}_{prb} \in \mathbb{C}^{N_b \times N_{prb}^{sc}}$, representing the beamformed signals coming from each channel can be calculated with

$$\mathbf{B}_{prb} = \mathbf{W}\mathbf{A}_{prb} = [\mathbf{W}\mathbf{a}_0, \mathbf{W}\mathbf{a}_0, ..., \mathbf{W}\mathbf{a}_{12}], \tag{8}$$

where $\mathbf{W} \in \mathbb{C}^{N_b \times N_{rx}}$ is the weight matrix for the PRB and $\mathbf{A}_{prb} \in \mathbb{C}^{N_{rx} \times N_{prb}^{sc}}$ the I/Q data received from each antenna for a single PRB. The operation can be further split to multiple matrix-vector multiplications as seen in (8), where each of the vectors $\mathbf{a} \in \mathbb{C}^{N_{rx} \times 1}$ represent non-beamformed subcarrier measurements from each receiving antenna. This process is then repeated for each PRB to detect all the received data.

The benchmark applications used to measure the performance of the modifications contain matrices with the dimensions given in Table 3. Matrix $\mathbf{A}$ is split into N_{prb}, $\mathbf{A}_{prb}$ matrices that have dimensions of $\mathbb{C}^{N_{rx} \times N_{prb}^{sc}}$. In the case of the benchmark applications, there are 18 of these matrices and each of them is multiplied by the weight matrix $\mathbf{W} \in \mathbb{C}^{N_b \times N_{rx}}$.

The matrix multiplication consists of complex multiply-accumulate (CMAC) operations. The result of a single index (b_{ij}) of matrix B can be obtained as

$$b_{ij} = \sum_{k=1}^{N_{rx}} w_{ik} a_{kj}. \tag{9}$$

To calculate each index of the result matrix, N_{rx} CMAC operations are required. The result of a single $\mathbf{W}\mathbf{A}_{prb}$ matrix multiplication requires therefore $N_b \times N_{prb}^{sc} \times N_{rx}$ CMAC operations and since there are N_{prb} of them in a single symbol the total amount is calculated with $N_b \times N_{prb}^{sc} \times N_{rx} \times N_{prb}$. In the case of the benchmark application, the total amount of CMAC operations performed in a single symbol duration is $16 \times 12 \times 64 \times 18 = 221184$.

Table 3: Matrix dimensions of $\mathbf{W} \in \mathbb{C}^{N_b \times N_{rx}}$ and $\mathbf{A} \in \mathbb{C}^{N_{rx} \times N_{prb}^{sc} \times N_{prb}}$

Description	**Amount**
Receiving antennas (N_{rx})	64
PRBs (N_{prb})	18
Subcarriers in PRB (N_{prb}^{sc})	12
Total subcarriers (N_{sc})	216
Beams (N_b)	16

Due to the large number of operations that need to be performed with a tight schedule, there needs to exist hardware that offers possibilities for large-scale parallelism. When using an ASIP to do the processing the software is equally important since it needs to be optimized so that it utilizes the available hardware as well as possible. The optimization methods used in the benchmark application are data parallelism, vector lane shuffling, loop unrolling, and efficient memory layout of the data.

The data parallelism is achieved by loading 32-lane data vectors from the memory to the vector registers of the processor and utilizing the SIMD processing capabilities of the ASIP. Vector shuffling is performed to increase the locality of the operations so that the amount of vector loads can be reduced significantly. The goal with the shuffling is to do operations as much as possible with the data that has already been fetched to the vector registers.

The actual program is split into a loop that is iterated N_{prb} times. Each iteration consists of a nested loop that processes a single PRB coming from each of the receiving antennas. The loops are unrolled to ease the job of the scheduling executed by the compiler.

Fig. 16 shows an example of loop unrolling. Here the for-loop is unrolled 4 times [45]. This reduces the overhead caused by jumps in the program code and enables more efficient pipelining of the CMAC operations. In addition, loop unrolling helps to reduce the need for loading values multiple times to registers. The downside of loop unrolling is that it increases the program code size in memory. The goal of the loop unrolling in the benchmark application is to help the compiler schedule consecutive CMAC operations to the arithmetic lane pipeline.

Basic for loop	Unrolled for loop

for (int i=0; i < 64; i++)
{
a[i] = a[i] * b + c;
}
	for (int i=0; i < 64; i+=4)
	{
	a[i] = a[i] * b + c;
	a[i+1] = a[i+1] * b + c;
	a[i+2] = a[i+2] * b + c;
	a[i+3] = a[i+3] * b + c;
	}

Figure 16: A for loop can be unrolled to contain multiple copies of the original loop body [45].

The I/Q data matrix is stored in memory in row-major order which means that the subcarriers coming from a single RX antenna are stored consecutively. On the contrary, the weight and result matrices are stored in column-major order so that the weights for the different beams are stored in continuous addresses. The data is structured in this way to enable the accumulation of multiple values of the result matrix in parallel.

4.1.2 Instruction set simulator benchmark

The benchmarking process was iterated by optimizing the hardware and software until a correct and well-performing solution was found. The threshold for good performance in this case is that the cycle count and program memory consumption do not increase significantly and the data memory consumption reduces. A slight increase in cycle count and program memory consumption is tolerated due to the overhead that the BFP decompression might cause. The compressed data should on the other hand decrease the memory needed to store the I/Q data.

The benchmark performance metrics extracted from the ISS simulations are clock cycles needed to complete the program, data memory, and program memory allocation. In addition, the hardware utilization of the different VLIW lanes of the ASIP was calculated based on the compiled assembly code of the benchmark application.

Fig. 17 describes the process used to implement and benchmark the architectural modifications. First reference input and output data is generated with a custom MATLAB script, that utilizes models of the beamforming matrix multiplication and BFP-decompression. The models used are a part of a baseband processing modeling library. It is important to have a simple reference model that produces correct results, since software optimization and hardware implementations, can introduce unintended errors to the design.

The MATLAB script is parameterized so that it can generate test input and reference output vectors for different matrix sizes, enabling the effortless creation of different test vectors. The MATLAB model creates randomized input vectors first in BFP format and then decompresses them to a more accurate representation. Then the matrix multiplication is executed with decompressed data to get the reference output vectors.

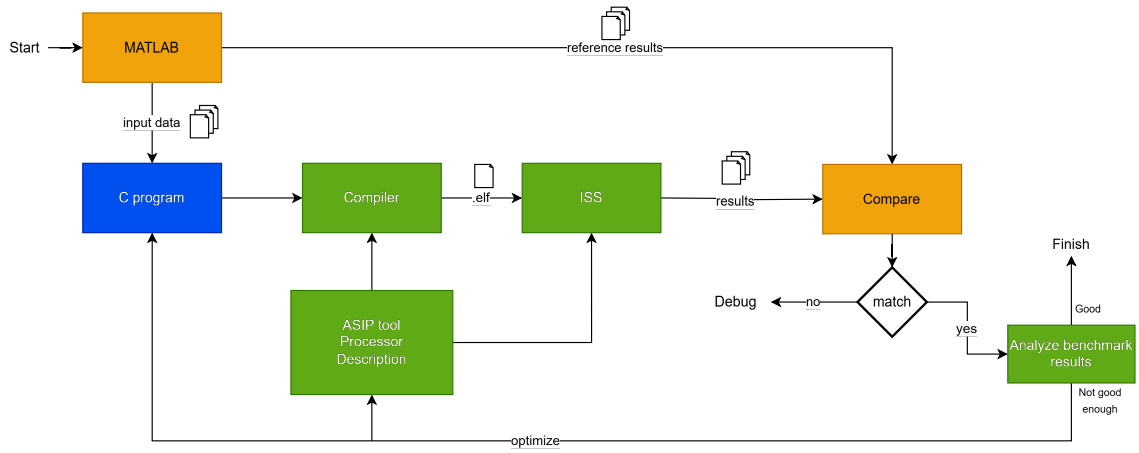

Figure 17: Benchmarking and verification process using MATLAB and the intruction set simulator provided by the ASIP tooling.

The hardware modifications needed for the new instructions and the benchmark

software are implemented in parallel with a hardware-software co-design methodology typical in ASIP development [12]. The processor description language provided by the ASIP tooling is used to implement the hardware modifications. The goal for the hardware modifications is to utilize the existing instruction level parallelism coming from VLIW architecture and pipelining, to seamlessly enable the decompression of the BFP data to be integrated into an optimized matrix multiplication algorithm.

The tool generates a retargetable compiler and an ISS based on the processor description. These are then used to simulate the execution of the benchmark application written in C. The compiler is used to create an executable linkable format of the application which the ISS then executes. The ISS contains a cycle-accurate instruction level description of the processor and can therefore simulate the pipeline scheduling correctly. The simulation can therefore be used to evaluate how many clock cycles it takes to run a certain program. However, the ISS does not measure how long a certain combinational path takes to execute. Therefore, it is important to take care that too long paths do not exist in the hardware design so that it can be later synthesized with an appropriate clock rate.

Software optimization is an important part of the benchmarking process. The software needs to be written so that it utilizes the hardware accelerating functional units as much as possible. When optimizing software it is important to check that the results are correct after each optimization. The reference output vectors produced by the MATLAB model are used to validate the correctness of the design. The correctness of the MATLAB model was also first cross-validated by running a simple scalar multiplication on the ISS of the ASIP and comparing the results. This was accomplished to verify that the memory layout of the input and output data was correct.

As previously mentioned, the baseline benchmark application that was modified to do the BFP decompression and matrix multiplication was already heavily optimized for the target architecture. The goal was to integrate the decompression into the optimized algorithm so that all of the previously mentioned optimization methods could still be utilized.

To have a simple software/hardware interface, the tool enables the definition of intrinsics to the compiler interface, that can be configured to use specific hardware accelerator blocks. These intrinsics make it easier for the compiler to determine what hardware acceleration blocks the program uses inside the ASIP. In this case, new intrinsics were defined so that the usage of decompressing hardware was easy to integrate into the applications. The intrinsics were then translated by the compiler to assembly instructions that utilized the dedicated hardware.

After running the benchmark program on the ISS the correctness of the results were checked. In the case they were wrong the system was debugged until a valid result was produced. When a valid result was obtained the benchmark results were analyzed. If the results were not desirable changes in the hardware and software were iterated until an adequate solution was found.

The MATLAB model calculated the beamforming results with complex double-precision floating point numbers, which have higher precision compared to complex floating point and fixed point numbers used in the ASIP ISS simulations. For this

reason error vector magnitude (EVM) method was used to quantify the accuracy loss caused by the BFP compression and calculation with less accurate data formats.

EVM is a standard metric in wireless and wireline communications [46]. It is widely used to test and evaluate the quality of modulated signals [11]. EVM can be calculated with

$$EVM = \frac{RMS(|IQ_{meas} - IQ_{ref}|)}{RMS(|IQ_{ref}|)} \times 100\%, \tag{10}$$

where RMS is the root mean square of a vector, IQ_{meas} is the measured I/Q vector, and IQ_{ref} is the reference I/Q vector. Fig. 18 illustrates how the error vector is formed in the complex I/Q plane for a single I/Q sample. In this thesis the reference vectors are the ones that were calculated with double precision format in MATLAB and the measured vectors were the ones calculated on the ISS of the ASIP.

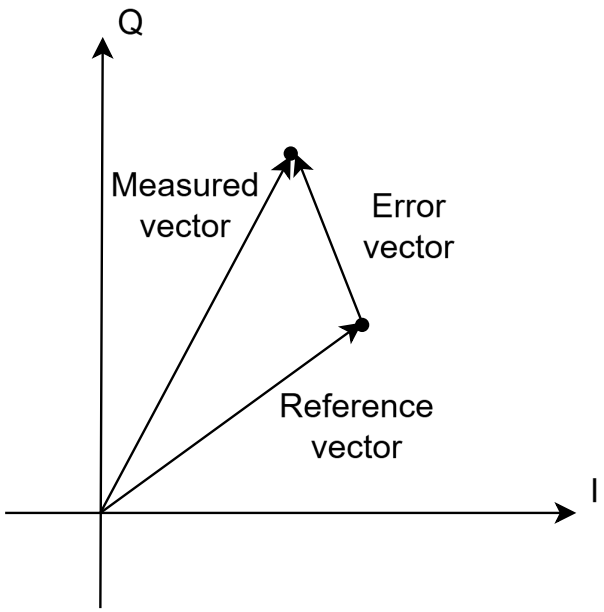

Figure 18: Error vector magnitude illustrated on I/Q complex plane.

The EVM was measured by first generating multiple input test vectors with different maximum exponent values to see if the signal magnitudes affect the accuracy. Then the ISS simulation results were compared to the MATLAB reference vector with (10), to obtain the total EVM of the simulations.

4.1.3 RTL simulations and synthesis

In addition to the ISS simulations, the benchmark applications were also run on the generated RTL description, to further verify the functionality of the modifications. Even though the RTL description and ISS generated by the tool should produce the same results, some internal flaws of the processor might be better detected with

the RTL simulations. For example, if some wires or registers have undefined values during the application execution they are easier to detect with RTL simulations.

In a digital design flow, logic synthesis is performed to the RTL description after the functionality of the design has been verified. The logic synthesis transforms the RTL design into an optimized gate-level netlist based on a given standard cell library and design constraints [47]. Standard cells are predesigned, pre-validated for functionality, and pre-characterized. The library usually consists of simple cells that implement logical functions or flip-flops, and more complex cells such as efficient multipliers. The synthesis tool takes as inputs the hardware description, design constraints, and the standard cell library [42]. Based on these inputs, the tool tries to find an optimal gate-level implementation by choosing and combining the different cells in the library.

Logic synthesis is an important part of the design since it allows timing estimation of the different combinational logic paths between registers [42]. For the synthesis tool, a desired clock frequency is given and the tool then tries to optimize the logical paths, so that they meet the timing requirements. In addition to timing analysis, an estimation of the silicon area can be obtained from the logic synthesis results, since each of the cells in the standard cell library has a defined area on silicon.

It is important to run the logic synthesis during the design process since the combinational paths that do not meet the timing constraints can be detected. Additionally, logic synthesis can reveal problems with the silicon area that the design requires. When flaws either with timing constraints or silicon area are found they can be optimized out before implementing new features to the processor.

4.2 Architectural modifications

This section presents the implementation details for two alternative architectural modifications introduced to an existing vector ASIP, enabling it to process BFP-compressed data. It begins with a high-level overview of the relevant processor architecture features regarding the modifications. Subsequently, challenges related to storing the compressed data are discussed, followed by more detailed descriptions of the proposed hardware modifications.

Fig. 19 illustrates the VLIW architecture of the processor that is modified. The vector processing unit of the ASIP consists of three parallel VLIW lanes, enabling the execution of orthogonal instructions in parallel. On top of the VLIW parallelism, the computational lanes are pipelined to increase the throughput of the processor. In the best-case scenario, the architecture allows the execution of a vector load, miscellaneous operation, and arithmetic operation to be performed in parallel every clock cycle. In reality, this kind of parallelism is rarely achieved every clock cycle, due to limitations caused by memory bandwidth, conditional statements, and data dependencies.

The ASIP architecture features vector registers with 32 parallel SIMD lanes, each lane consisting of 32-bit words. Each VLIW datapath is designed to process these SIMD vectors, offering significant potential for data-level parallelism. The ASIP is designed for application domains characterized by high data parallelism, such as

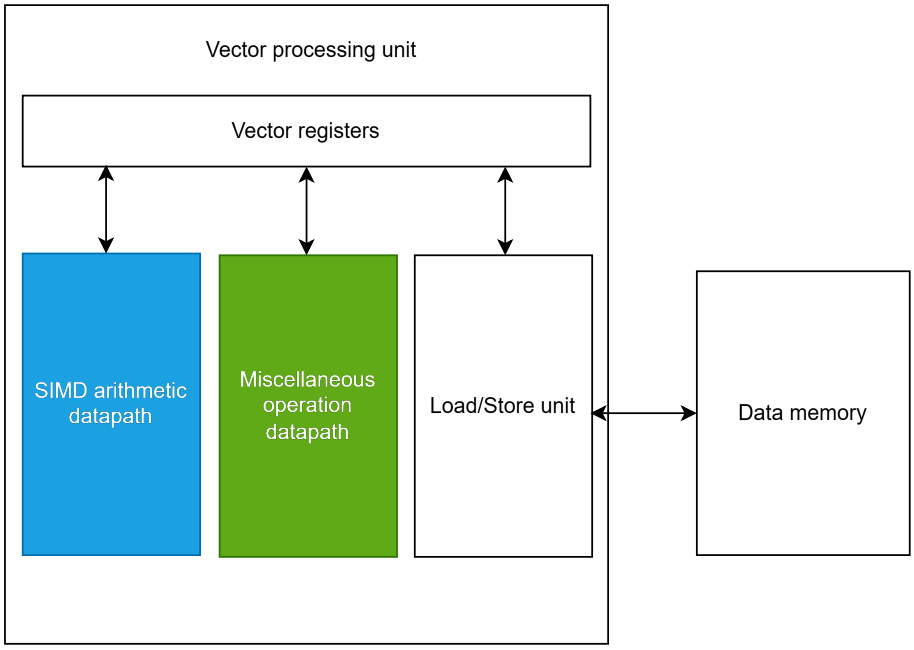

Figure 19: High-level Block diagram of the vector processing unit inside the ASIP.

large matrix and vector computations. However, to realize high hardware utilization, the software needs to be specifically optimized to exploit both the instruction-level parallelism and data-level parallelism.

The first modification is a set of standalone decompression instructions, which decompress BFP-compressed PRBs to complex floating point format. The second modification provides a merged decompress and multiply-accumulate instruction. This instruction utilizes the existing vector lane shuffling unit, located in the multiplier datapath, to perform the decompression to a fixed-point data format. For both output formats (complex floating-point from modification 1, complex fixed-point from modification 2), I/Q data pairs are represented with 32-bit words.

The implementation strategy for the BFP modifications considers the parallelism provided by the architecture. The standalone decompression instruction (Modification 1) is mapped to the miscellaneous instruction lane (highlighted green in Fig. 19). This placement enables parallel scheduling by the compiler, allowing the decompression to execute concurrently with arithmetic and vector load/store instructions. Modification 2 integrates the decompression functionality directly into the arithmetic datapath (highlighted blue in Fig. 19). While this approach frees the miscellaneous lane for other operations, it potentially increases the computational overhead within the arithmetic datapath.

4.2.1 Memory Interface

The representation of Block Floating Point (BFP) compressed I/Q data in memory is non-trivial, due to challenges posed by the compression format and hardware architecture. Fig. 20 illustrates how the BFP compressed I/Q data matrix can be

represented in memory. The I/Q data matrix $\mathbf{A} \in \mathbb{C}^{N_{rx} \times N_{sc}}$ is stored in the memory in row-major order so that each of the PRBs in a symbol, coming from a single antenna stream are stored sequentially.

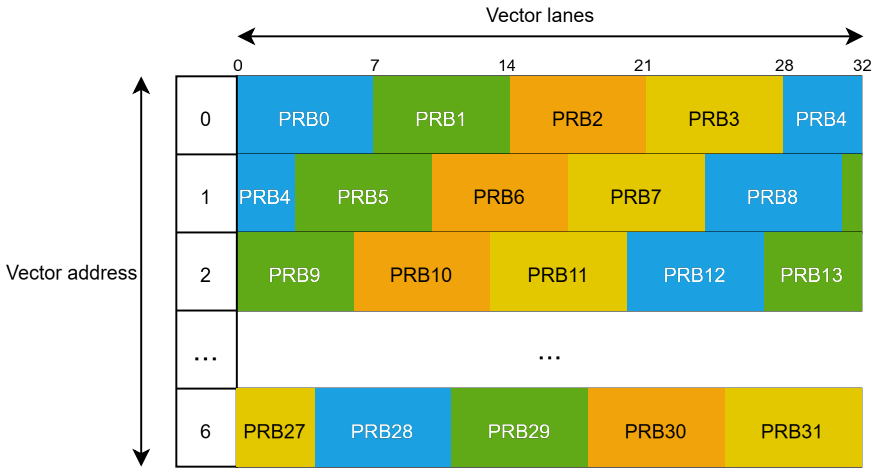

Figure 20: Data memory layout of 9-bit BFP compressed data.

The O-RAN 9-bit BFP format compresses each PRB, which consists of 12 I/Q pairs, into a fixed size of 224 bits. To interface with hardware that features 32-bit data words and 32-lane vector registers (totaling 1024 bits), a new vector data type was introduced. This data type is 1024 bits long and designed to hold four 9-bit BFP compressed PRBs. This means that $4 \times 224 = 896$ bits are useful, and the remaining $1024 - 896 = 128$ bits are unused, within the 1024-bit vector register. These unused bits are, in effect, the beginning of the subsequent PRB that doesn't fully fit into the current vector register.

The discrepancy between the BFP PRB size (224 bits, 28 bytes) and the vector register width (1024 bits, 128 bytes) presents a challenge for vector-aligned memory access. If memory loads were strictly aligned to vector addresses with 128-byte boundaries, individual PRBs would often be split across multiple vector registers, complicating PRB-wise processing.

To overcome this, the system utilizes misaligned memory loads. This capability, already supported by the processor, allows fetching data from memory starting at an arbitrary word-aligned address, ensuring that a complete 28-byte BFP-compressed PRB can be loaded into a vector register even if it doesn't start precisely at a 128-byte aligned address. This enables more efficient processing of the BFP-compressed data on a PRB basis

While it would theoretically be possible to avoid the unused bits by aligning PRBs differently, the least common multiple of 224 and 1024 is 7168 bits. This would require aligning PRBs across 7 vector lanes, adding significant complexity and overhead to the memory access and decompression processes. Therefore, the design accepts the presence of the 128 unused bits per BFP vector data type to simplify the data loading and processing.

Furthermore, the I/Q values within the BFP compressed data do not align with the boundaries of the 32-bit vector lanes. This necessitates either decompressing the BFP data before processing them in the existing arithmetic units or implementing specialized arithmetic units designed to process BFP-compressed data. The described implementations use the first approach, decompressing the data beforehand to leverage the existing arithmetic units. This choice is driven by considerations of silicon area, hardware utilization, and implementation complexity, making it a more cost-effective solution.

4.2.2 Modification 1: Decompression instruction

The first modification is a set of BFP decompression instructions that takes as an input the new vector data type containing 4 BFP compressed PRBs. Two different decompression instructions were created: the first one is called `dec4bfp`, which decompresses all 4 PRBs to two output vectors and the second one is called `dec2bfp`, which decompresses two PRBs into a single output vector. The data format chosen for the output was complex floating point which provides more accuracy to the calculations. Both of the instructions use the same hardware to decompress a single PRB, therefore there exists in total 4 parallel BFP decompression units to reduce the latency of the decompression.

Fig. 21 shows a high-level block diagram of the decompression instructions SIMD data path. First, the compressed data is fetched from the vector register file. From the software perspective, the data needs to be assigned to the BFP vector data format. Depending on the instruction either two or four PRBs are extracted for decompression. When in compressed format the PRBs are represented with $7 \times 32 = 224$ bits and after decompression the representation consists of $12 \times 32 = 384$ bits each 32-bit word representing an I/Q data pair complex floating point format. After the decompression, the lanes are shuffled based on the instruction mode, and finally, the results are written back to the vector registers.

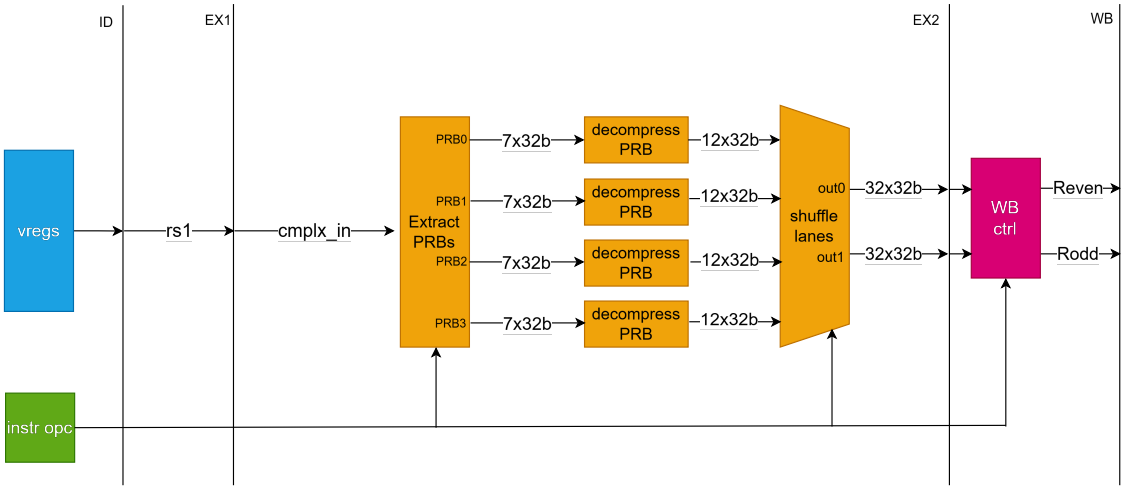

Figure 21: SIMD data path of the 9-bit BFP decompression instruction.

The extraction of PRBs from the vector depends upon the decompression instruction invoked. The 4-PRB instruction `dec4bfp` retrieves four 224-bit fields from the beginning of the vector and dispatches them to the respective decompression units. Correspondingly, the 2-PRB instruction extracts the first two PRBs and forwards them to two decompression units.

During the decompression of a BFP-compressed PRB, individual I/Q samples can be decompressed in parallel. Algorithm 3 outlines the steps involved in decompressing the I/Q samples. The mantissas, representing individual I/Q samples, are initially converted from 2's complement to sign-magnitude format. This conversion is achieved by extracting the sign bit and inverting the remaining mantissa bits if the sign is negative. Subsequently, the leading zeros are identified, and the mantissa is left-shifted based on this count. Normalization and overflow handling are also performed at this stage. The output of this stage consists of the amount of applied shift and the shifted mantissa in sign-magnitude format.

Following the mantissa normalization, the exponent is calculated individually for each I/Q sample. The common exponent within the BFP format does not incorporate a bias, which must therefore be added to the resulting exponent value. Furthermore, the mantissa value is normalized to lie within the range [1, 2). This normalization is typically achieved by an 8-bit left shift of the floating point, corresponding to an addition of 8 to the exponent. The number of left shifts applied to normalize the mantissa is subtracted from the calculated exponent. Finally, the resulting mantissa and exponent values are concatenated to form the decompressed resource element.

Algorithm 3 Block floating point decompression to complex floating point

Input: $cPRB,\ exp$
Output: $dPRB$

1: **for** i = 0 to 12 **do**
2: $\quad smantRe, shftRE \leftarrow normalize(\Re(cPRB[i]))$
3: $\quad smantIm, shftIm \leftarrow normalize(\Im(cPRB[i]))$
4: $\quad expRe \leftarrow exp + bias + 8 - shftRe$
5: $\quad expIm \leftarrow exp + bias + 8 - shftIm$
6: $\quad dPRB[i] \leftarrow concat(smantIm, expIm, smantRe, expRe)$
7: **end for**

There is also a small shuffling unit that implements three different align modes for the output vectors. These apply only with the `dec4bfp` instruction and are implemented to be used when the data is decompressed and stored back in memory.

Fig. 22 shows how the result vectors are aligned based on the alignment mode used. When decompressing 4 PRBs the result is in total 48 words long. Therefore there will be some unused vector lanes in the decompressed result vectors. Align mode 0 fills the first result vector completely, and stores the remaining values to the first half of the second result vector. Align mode 1 fills the end half of the first result vector and fills the result vector 2 with the remaining values.

With these two alignment modes, it is possible to use one vector shuffling instruction per two decompression instructions to store the data in memory to continuous

addresses. However, this is not optimal in terms of memory allocation. In the case the values are written back to memory there needs to be space allocated both for the compressed and decompressed values. Therefore, this solution is not applicable to the benchmark application, since one of the main goals is to save space from the local data memory of the ASIP.

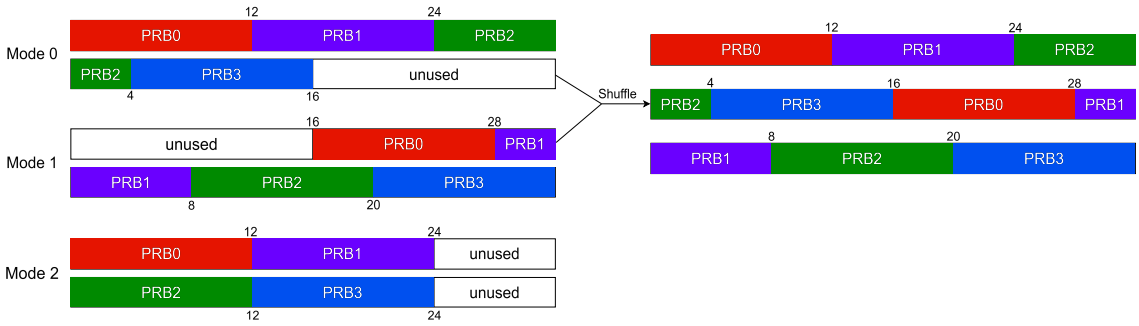

Figure 22: Result vector alignment modes after decompression.

Align mode 2 and the `dec2bfp` instruction are more suitable to be used when the result vectors are not written back to memory, which is the case in the benchmark application.

The efficiency of modification 1 depends a lot on the software optimizations used to utilize the implemented hardware as much as possible. In the optimized benchmark application only the `dec2bfp` instruction was used, therefore if the benchmark application is the only real use case of the application there exists a redundant shuffling unit for the output alignment and two unused PRB decompression units. However, adding these currently redundant units allows more flexibility for future improvements in algorithms and a wider set of use cases if needed.

Fig. 23 shows a simplified version of the vectorized I/Q data flow in the processor when the execution of the decompressing matrix multiplication starts. Each of the blocks represents a 32-lane vector. The pipeline lengths of the different VLIW lanes are not accurate to the actual implementation and they all have the same length for illustration purposes. First, the I/Q data is fetched from the memory to the vector register file utilizing the vector load/store pipeline. The loads and stores are pipelined so that more data can be loaded and stored each clock cycle.

When the data appears in the vector register file, it is forwarded for decompression happening in the miscellaneous operation lane. After the data is decompressed it is written back to the vector registers. After the write-back, the data is forwarded to the arithmetic unit where a complex multiply-accumulate operation is performed on it. The CMAC operation takes as inputs the weights, I/Q data, and a result accumulator value. When the operation is finished the result accumulator is written back to the memory.

The diagram of Fig. 23 does not include the overhead of loading the weights. The weights are needed in the multiply-accumulate process and are already in a decompressed format when arriving at the vector register file. In the actual application,

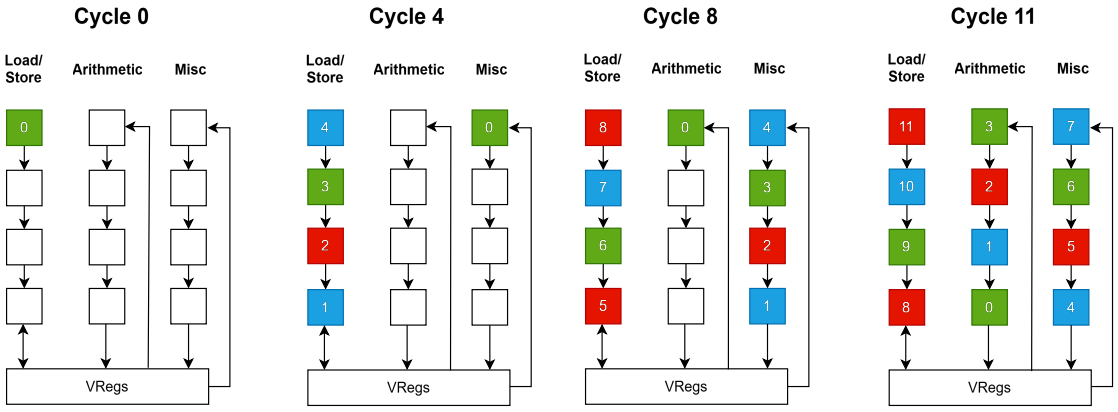

Figure 23: Simplified I/Q data flow in the ASIP pipeline when running the decompressing matrix multiply application for modification 1.

the shuffling units of the arithmetic lane are utilized to operate on the same data so that fewer memory loads are needed. The vector load/store unit also needs to facilitate the bandwidth for writing the results back in the memory, which further increases the importance of minimizing memory interface activity.

4.2.3 Modification 2: Merged decompress and multiply instruction

The second modification approached the BFP-compressed data computing by implementing the decompression in the arithmetic datapath. With this approach, the decompressed data was stored back to the vector registers only after it was already processed. This instruction enables consuming the same BFP vectors, as in modification 1, from the register file straight to the arithmetic VLIW-lane.

Fig. 24 illustrates the multiplier data path. After the respective arithmetic instruction is decoded the I/Q data is forwarded to a shuffling unit. The modification added a mode to the shuffling unit that also does the decompression to a fixed point format. The fixed point decompression has less overhead compared to decompressing to floating-point since the fractional value remains in 2's complement format. After the data has been decompressed and shuffled it is forwarded to the actual arithmetic units that perform the CMAC operation. Finally, after a few pipeline stages, the results are written back to the memory.

Before the modification, there was a shuffling unit that was only able to shuffle complete 32-bit words between the lanes. However, with 9-bit BFP the I/Q pairs do not in most cases follow the boundaries of the 32-bit words. Therefore a different shuffling unit that is able to extract and decompress the BFP-compressed data was created. Currently, the shuffling unit supports only a single mode (`bcast2`) where the two elements of the selected BFP block are decompressed and broadcasted to the 32 SIMD lanes. This shuffling mode was selected for trial since it was used in the benchmark application.

The block diagram of Fig. 25 illustrates the structure of the decompressing

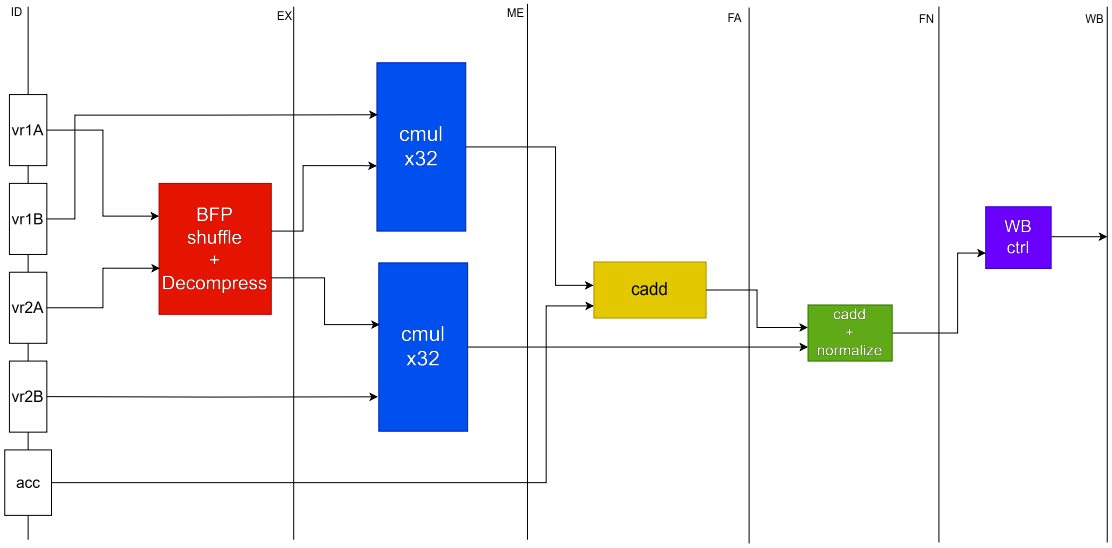

Figure 24: SIMD data path of the merged decompression and CMAC instruction.

shuffling unit. First, the BFP block is selected from the input vector that contains 4 BFP compressed PRBs. The selection is carried out with a multiplexer that is controlled by the instruction. After the correct block is selected, another multiplexer also controlled with the instruction, determines which two adjacent I/Q pairs are forwarded to the decompression block that decompresses them to a complex fixed point format. Finally, the two decompressed I/Q pairs are broadcasted to the vector lanes forming the output vector.

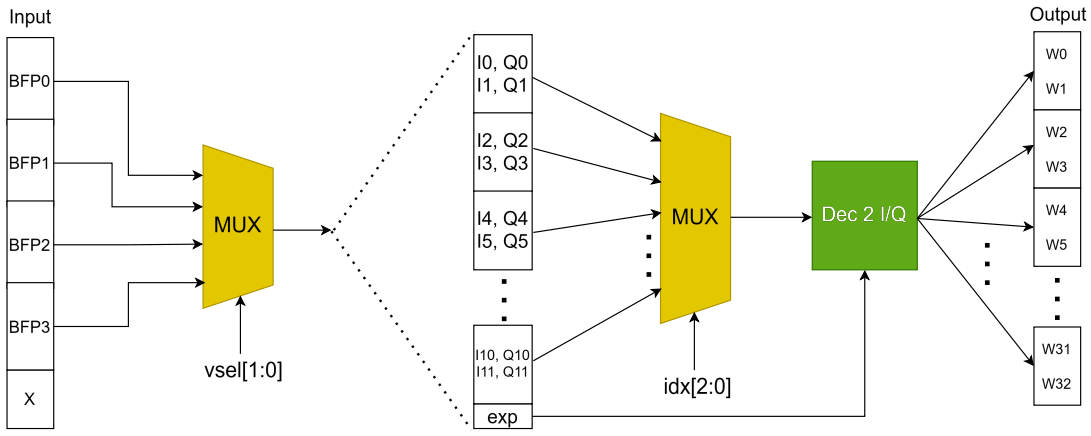

Figure 25: Block diagram of the decompressing shuffler.

All of the two adjacent words in the output vector receive the same copy of the two decompressed I/Q pairs. Therefore, for each input vector, only two values are decompressed. The decompression from BFP to fixed point has a small overhead as can be seen from algorithm 4.

Algorithm 4 Block floating point decompression to complex fixed point

Input: $cPRB,\ exp$
Output: $dPRB,\ offlineExp$

if $exp > 7$ **then**
  $offlineExp \leftarrow exp - 7$
  $shft \leftarrow 7$
else
  $offlineExp \leftarrow 0$
  $shft \leftarrow exp$
end if
for i = 0 to 12 **do**
  $smantRe \leftarrow \Re(cPRB[i]) \times 2^{shft}$
  $smantIm \leftarrow \Im(cPRB[i]) \times 2^{shft}$
  $dPRB[i] \leftarrow concat(smantIm, smantRe)$
end for

The algorithm starts by determining how much shift is needed to produce the fixed point value. An offline exponent can be used to increase the dynamic range of the output values. It stores the rest of the exponent that is not used to left shift the input value. After the amount of shift is determined the BFP mantissas are simply shifted left and concatenated to produce the decompressed values.

The software of the benchmark application for modification 2 is very similar to the baseline application. On the software side, only the vector load and shuffling intrinsics needed to be changed to utilize the newly implemented hardware. The currently implemented hardware is very specifically tailored for the benchmark application, but at the same time., it has the potential to be expanded to support other vector lane shuffling modes to increase its flexibility.

Fig. 26 shows an ideal case of the I/Q data propagation in the processor pipeline when the matrix multiplication starts. Contrary to modification 1 the miscellaneous VLIW lane is idle since all the decompression activity happens in the arithmetic lane. The benefit to this is that the miscellaneous lane is now free for other instructions during the processing if needed.

The pipeline needed to obtain result vectors is in effect shorter compared to modification 1, but as can be seen in the results section the impact of the shorter pipeline length is unnoticeable in the case of the benchmark applications.

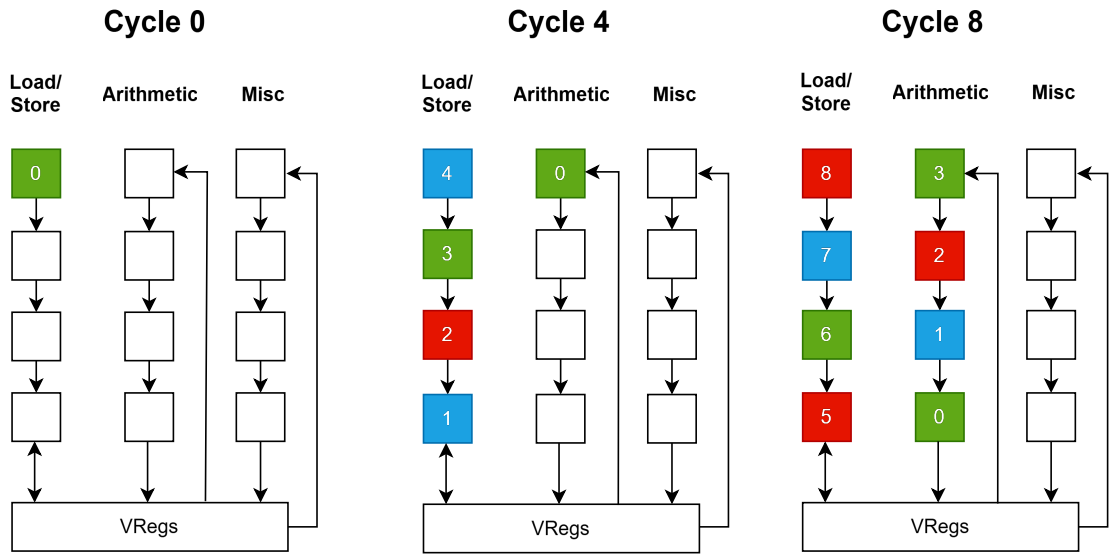

Figure 26: Simplified I/Q data flow in the ASIP pipeline when running the decompressing matrix multiply application for modification 2.

5 Results

This chapter presents and analyzes the results obtained from benchmarking and logic synthesis. Performance parameters, such as memory consumption and application latency, were quantified using a cycle-accurate ISS, with results validated against a MATLAB reference model. To further verify the hardware implementations, RTL-level simulations were also compared to the respective ISS simulations. Logic synthesis was carried out to provide data on the maximum clock frequency and silicon area for comparison purposes.

5.1 ISS Benchmark results

The main goal of the benchmarking was to determine how much memory can be saved, and how much overhead it causes if BFP-compressed data is directly consumed from the memory into the calculation units. Beamforming is a realistic use case where the ASIP could benefit from BFP-compressed data support. The dimensions of the input matrices were presented in Table 3.

Fig. 27 illustrates the cycle count and Fig. 28 shows the data memory consumption of the different benchmark applications. The cycle counts were normalized by the cycle count of the reference application. The reference application was run on an unmodified version of the ASIP and all other applications were run on the respective modified versions. `Mod1Slow` application illustrates the case where all of the BFP-compressed I/Q data is first decompressed using the decompression instructions and then stored back in the memory. Only after all the data was decompressed the matrix multiplication was exexuted.

This sequential approach performed significantly slower and consumed more data memory resources compared to the reference application. Therefore, in both of the optimized applications (`Mod1Opt`, `Mod2Opt`), the decompression was executed in parallel with the matrix multiplication, and the data was only stored back in the memory when the results were fully obtained from the CMAC calculations. With this approach, both of the modifications achieved the target of not increasing latency in clock cycles and reducing the data memory consumption.

Table 4 presents in more detail, the results obtained from the optimized benchmark applications. Both of the implementations fulfilled the requirements of not increasing the latency significantly while reducing the data memory consumption. The tool simulated the same latency in clock cycles for the baseline application and modification 2, which was to be expected since the instruction count in both applications was exactly the same due to merging the decompression instruction with the CMAC operation.

Modification 1 resulted in a marginal latency increase (0.34%) in clock cycles since there were more instructions to execute. The compiler was able to schedule the memory interface and arithmetic instructions efficiently with the decompression instructions. This can be seen from the Instructions Per Cycle (IPC) metric that increased when the decompression instructions were used. IPC metrics were calculated considering that each VLIW lane executed a distinct instruction. No-operation

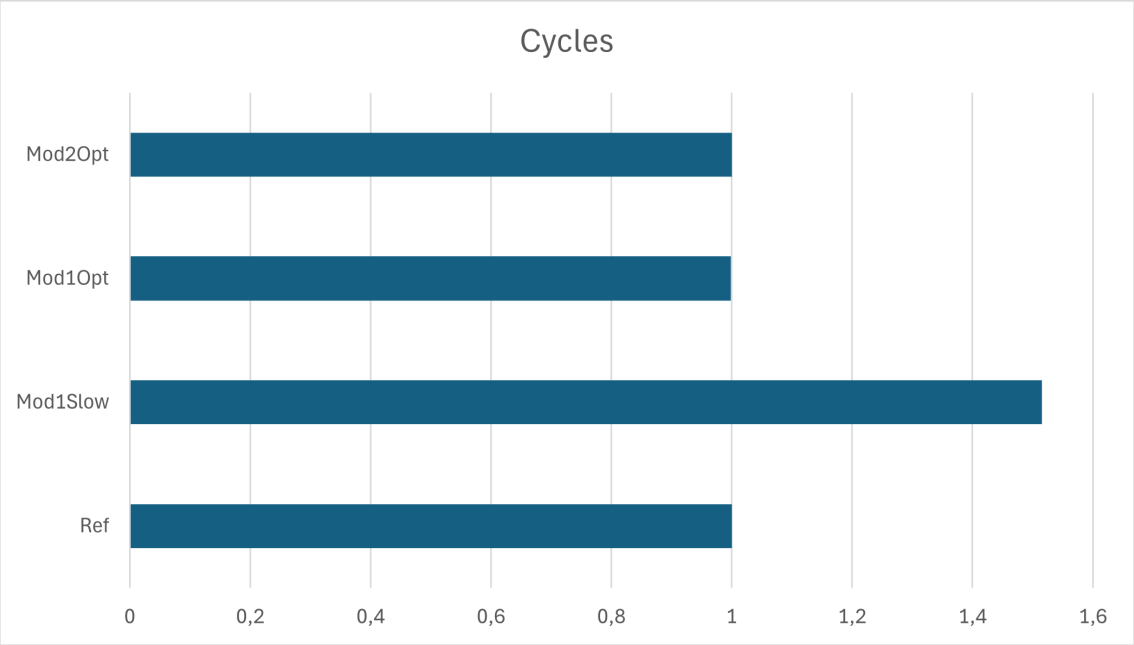

Figure 27: Normalized cycle counts for the different benchmark applications.

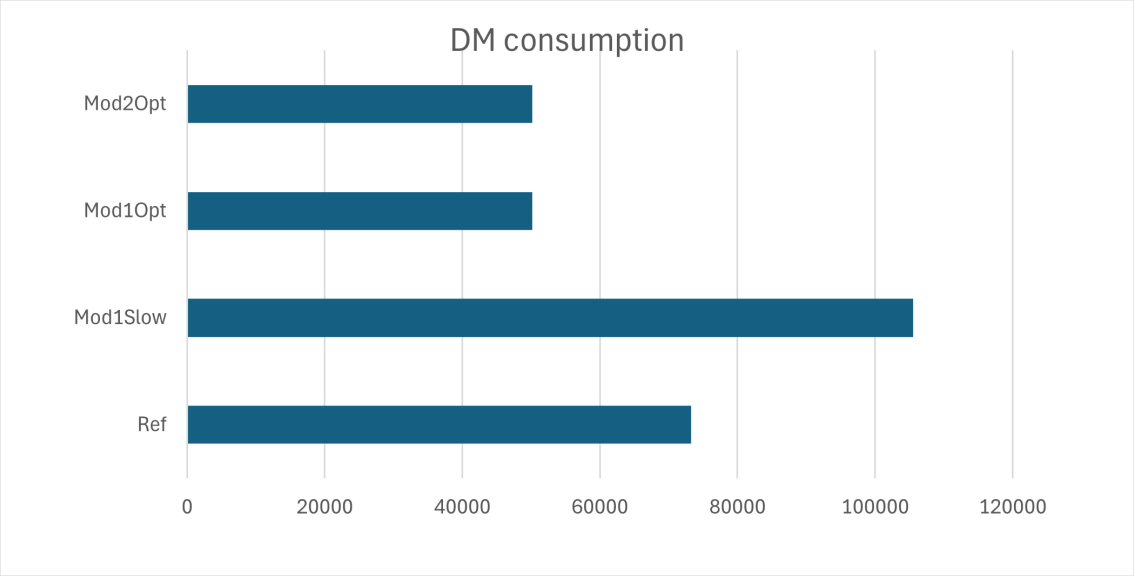

Figure 28: Data memory consumption in bytes for the different benchmark applications.

(NOP) instructions, which perform no useful computation, were excluded from these calculations.

The data memory consumption is the same for both of the modifications. The 9-bit BFP format decreases the I/Q data memory consumption by 41.6% but since

Table 4: Comparison of Performance for 16 beam 64 TRX applications

Performance metric	**Unit**	**Opt. baseline**	**Opt. mod 1**	**Opt. mod 2**
Clock Cycles	Δ%	0	+0.34%	0
IPC	Δ%	0	+14.7%	0
DM consumption	bytes	73216	50194 (-31.4%)	50194 (-31.4%)
PM consumption	bytes	534	588 (+10.1%)	534 (+0.0%)

there is memory allocation for the results and weights that are not compressed the maximum compression obtained is −31.4%. The maximum compression is in the case that all of the PRBs use the same weight matrix. In reality, this is not always possible and the rate of saved memory would decrease when the weight matrix changes between the PRBs.

The program memory consumption increased slightly for modification 1, which was expected due to the additional decompression instructions. For both modifications, the software complexity didn't increase significantly, since the decompression could be added to the software by changing the data type of the I/Q data and adding the intrinsics functions that used the decompressing hardware.

Table 5 shows the hardware utilization percentages for the beamforming matrix multiplication function. Modification 2 has the same percentages as the reference application. The utilization of the arithmetic lane is very high in each case but slightly reduces when using the stand-alone decompression instruction. However, the miscellaneous lane utilization increases significantly, which also affects the total utilization of the VLIW lanes when running the benchmark application.

Table 5: Hardware utilization of VLIW-lanes when running benchmark applications.

VLIW-lane	**Opt. baseline**	**Opt. mod 1**	**Opt. mod 2**
Artimetic	97.5%	97.0%	97.5%
Memory	52.3%	52.1%	52.3%
Misc.	9.3%	33.4%	9.3%
Total	53.0%	60.8%	53.0%

The only downside of increased hardware utilization is that the dynamic power consumption increases, due to logic gates toggling more frequently inside the core. Therefore, the decompression instruction might be less power efficient compared to the merged decompress and CMAC instruction. However, modification 2 still does the same decompression in the arithmetic lane which increases the activity there, which then increases the total power consumption.

Table 6 shows the results from the EVM simulations. The different maximum exponent values were used to see if the magnitude of the input signal causes inaccuracies in the results. The EVM was very small for both modifications when compared to the more accurate Matlab results. Therefore, it can be concluded that the accuracy of both implementations was good enough.

Table 6: EVM results for different signal magnitudes

Max exponent	EVM mod 1	EVM mod 2
4	0.044%	0.044%
8	0.045%	0.045%
15	0.044%	0.044%

5.2 Synthesis results

Logic synthesis was performed for both of the modified versions and the unmodified baseline version of the ASIP, to measure how much implementation overhead the modifications cause. Table 7 shows the relative change in area and maximum clock frequency for both of the modifications.

The area obtained from the logic synthesis acts as an estimation of the final area required for the physical implementation. Each cell has a defined area in the standard cell library. After the logic synthesis, the tool calculates the total area of all the cells used to synthesize the gate-level implementation of the design. The logic synthesis does not take into account the overhead that comes from placing and routing all the different cells to form the physical implementation. However, it gives a good enough estimation of the total area at this stage of the development.

Table 7: Comparison of Synthesis results

Performance metric	Unit	Baseline	Mod 1	Mod 2
Clock frequency (f_{clk})	Δ%	0	0	-7.69%
Silicon area	Δ%	0	+0.44%	+1.08%

The clock frequency degraded significantly in modification 2, due to the shuffling operation already being relatively slow even without the additional overhead from the decompression. The increase in area was also higher compared to modification 1. The increase in the area of modification 2 came from the additional vector lane shuffling unit which was significantly larger than the standalone decompression unit of modification 1.

5.3 Discussion

Both modifications achieved their target of reducing the data memory consumption. Modification 1 is a more efficient and flexible solution based on the simulation and logic synthesis results when compared to modification 2.

The efficiency comes from the fact that the maximum clock frequency remained the same as for the baseline ASIP. Table 8 shows the difference in the execution time of the benchmark applications. The execution time was calculated based on (5). The execution time of modification 2 increased significantly more when compared to modification 1, due to the reduced clock frequency.

Modification 1 was also a more flexible solution due to the instruction not being tied to a CMAC operation. Therefore, the standalone decompression instruction can

Table 8: Comparison of total execution time of the Benchmark applications

Performance metric	**Unit**	**Baseline**	**Mod 1**	**Mod 2**
Execution time (T_{ex})	$\Delta\%$	0	+0.51%	+7.82%

be used also in other similar algorithms where I/Q data is processed. To further increase the flexibility of modification 2, more vector shuffling schemes would need to be implemented in the decompressing shuffler. This would also mean that the area needed for modification 2 would increase. The only downside of modification 1 is that the dynamic power consumption might be higher due to the parallel execution of the decompression instruction.

Due to the shortcomings of modification 2, modification 1 would be a more preferable solution to add the BFP support. Furthermore, modification 1 has a higher potential to be expanded to support multiple compression data types, since the decompression is carried out in a separate lane without the overhead coming from the shuffling operation.

In addition to the two modifications discussed in this thesis, a third option to implement BFP support was briefly explored. In this approach, a new multiplier that processes BFP-compressed data would be added. However, adding a new multiplier would have complicated the structure of the ASIP significantly and increased the area too much. Therefore, the more area-efficient approaches presented in this thesis were chosen.

6 Conclusion

The objective of this thesis was to explore alternative approaches to operate on BFP-compressed I/Q data. The goal was to save data memory space without significant processing and silicon area overhead, by integrating BFP support into a vector processing ASIP. To successfully integrate the BFP functionality, while saving memory space, meant that the data would have to be decompressed and operated on without intermediately storing it back in the memory. The chosen application was uplink beamforming since it is a realistic processing scenario for the I/Q data after it is decompressed.

Two different approaches were implemented on top of an existing vector processing ASIP. The first approach (modification 1) added a standalone decompression instruction, which can be used in parallel to arithmetic instruction due to the VLIW architecture of the vector processing ASIP. The second approach (modification 2) added a merged decompress and CMAC operation where the vector shuffling unit of the arithmetic VLIW-lane was utilized to do the decompression.

Both modifications were benchmarked and evaluated with an uplink beamforming application. The main objective of the benchmarking was to evaluate how much data memory can be saved and how much processing overhead the parallel decompression of the BFP-compressed I/Q data causes. To have a realistic amount of data in the processing scenarios, the benchmark applications used input matrices dimensioned with 64 receiving antennas 18 PRBs, and 16 beams.

The baseline application used as a reference got the input data already decompressed and processed an optimized matrix multiplication algorithm. The benchmark applications used for both of the modifications got the same data but in a compressed format, and in parallel decompressed the data and calculated the matrix multiplication.

Modification 1, succeeded in reducing the data memory consumption of the I/Q data without adding significant processing latency, while running the benchmark application. The modified ASIP was synthesized successfully with the same clock frequency as the baseline version, with a slightly increased silicon area. Hardware utilization of the VLIW lanes increased due to the parallel decompression instructions. The increased hardware utilization is beneficial from the area efficiency perspective, but at the same time disadvantageous when considering dynamic power consumption.

In addition, modification 1 was a flexible approach for the decompression, since it could, in theory, be integrated as a part of other baseband processing algorithms. However, the decompression can be integrated only if the algorithm has free slots in the miscellaneous VLIW lane for BFP decompression.

Modification 2 was successful in reducing the data memory consumption of the I/Q data, but introduced some additional latency due to reduced clock frequency. The additional silicon area needed was also higher when compared to the first modification. Another downside of modification 2 is that it is currently implemented only for a single vector shuffling mode, which makes it an extremely application-specific solution. However, there are possibilities to increase the flexibility by adding different shuffling modes, but this comes with an additional overhead in the silicon area. One benefit

of this approach is that the miscellaneous VLIW lane can do other tasks if needed.

It is left for future work to investigate which modification scales better when other compression schemes than the 9-bit O-RAN BFP are supported. However, based on the current results modification 1 would scale better, due to not having problems with the latency aspect and providing more flexibility due to being separate from the arithmetic data path.

In conclusion, both of the modifications were successful in reducing the data memory consumption. Furthermore, they were easy to integrate from the software perspective into the optimized baseline matrix multiplication algorithm used in beamforming. From the two approaches modification 1 seems like the better approach due to its performance and flexibility.

sources

[1] L. Kundu, X. Lin, E. Agostini, V. Ditya, ja T. Martin, "Hardware acceleration for open radio access networks: A contemporary overview," *IEEE Communications Magazine*, vol. 62, no. 9, s. 160–167, 2023.

[2] M. Attari, L. Ferreira, L. Liu, ja S. Malkowsky, "An application specific vector processor for efficient massive mimo processing," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 69, no. 9, s. 3804–3815, 2022.

[3] J. Hoozemans, P. Jääskeläinen, K. Tervo, ja Z. Al-Ars, "Energy efficient multi-standard decompressor ASIP," *Proceedings of the 2021 7th International Conference on Computing and Data Engineering*, 2021.

[4] O. Castañeda, L. Benini, ja C. Studer, "A 283 pj/b 240 Mb/s floating-point baseband accelerator for massive MU-MIMO in 22FDX," Published: *ESSCIRC 2022-IEEE 48th European Solid State Circuits Conference (ESSCIRC)*. IEEE, 2022, s. 357–360.

[5] M. Sode, M. Ponschab, L. N. Ribeiro, S. Haesloop, E. Tohidi, M. Peter, S. Stańczak, B. H. Mohamed, W. Keusgen, H. Mellein, E. Yassin, ja B. Schroeder, "Reconfigurable Intelligent Surfaces for 6G Mobile Networks: An Industry R&D Perspective," *IEEE Access*, vol. 12, s. 163155–163171, 2024.

[6] S. Islam, Z. A. Atallah, A. K. Budati, M. K. Hasan, R. Kolandaisamy, ja S. Nurhizam, "Mobile Networks Toward 5G/6G: Network Architecture, Opportunities and Challenges in Smart City," *IEEE Open Journal of the Communications Society*, s. 1–1, 2024.

[7] Y. Liu, L. Liu, ja V. Öwall, "Architecture Design of a Memory Subsystem for Massive MIMO Baseband Processing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, s. 2976–2980, 2017.

[8] R. I. Rony, E. Lopez-Aguilera, ja E. Garcia-Villegas, "Cost Analysis of 5G Fronthaul Networks Through Functional Splits at the PHY Layer in a Capacity and Cost Limited Scenario," *IEEE Access*, vol. 9, s. 8733–8750, 2021.

[9] O-RAN Working Group 4, "Control, User and Synchronization Plane Specification," Technical Specification O-RAN.WG4.CUS.0-v07.00, 2024, [Online] Available: https://www.o-ran.org/specifications.

[10] M. Mahdavi, *Baseband Processing for 5G and Beyond: Algorithms, VLSI Architectures, and Co-design*. Lund University, 2021, no. 140.

[11] D. M. Thang, V. T. Van, N. Van Cuong, ja N. Q. Nhu Quynh, "Hardware Implementation for O-RAN Block Floating Point Compression and Decompression Methods in 5G Base Station," Published: *2024 International Conference on Advanced Technologies for Communications (ATC)*, 2024, s. 90–95.

[12] S. Shahabuddin, A. Mämmelä, M. Juntti, ja O. Silvén, "ASIP for 5G and Beyond: Opportunities and Vision," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 68, no. 3, s. 851–857, 2021.

[13] M. Attari, J. R. Sánchez, O. Edfors, ja L. Liu, "A 1095 pJ/b 219 Mb/s Application-specific Instruction-set Processor for Distributed Massive MIMO in 22FDX," Published: *2024 IEEE European Solid-State Electronics Research Conference (ESSERC)*, 2024, s. 257–260.

[14] S. Kaur, R. Dhand, ja P. Sharma, "Evolutionary Trends in Mobile Networks From 1G to 6G," Published: *2023 International Conference on Research Methodologies in Knowledge Management, Artificial Intelligence and Telecommunication Engineering (RMKMATE)*, 2023, s. 1–5.

[15] Douglas H. Morais, *Key 5G Physical Layer Technologies*, 2nd ed. Springer, 2022.

[16] 3GPP, "5G System Overview," 2022, [Online] Available: https://www.3gpp.org/technologies/5g-system-overview.

[17] 3GPP, "5G; NG-RAN; Architecture description (3GPP TS 38.401 version 17.10.0 Release 17)," ETSI Std., 2024.

[18] M. Polese, L. Bonati, S. D'Oro, S. Basagni, ja T. Melodia, "Understanding O-RAN: Architecture, interfaces, algorithms, security, and research challenges," *IEEE Communications Surveys & Tutorials*, vol. 25, no. 2, s. 1376–1411, 2023.

[19] O-RAN Working Group 1, "O-RAN Architecture Description," Technical Specification O-RAN.WG1.OAD.R003-v12.00, 2024, [Online] Available: https://www.o-ran.org/specifications.

[20] F. Rinaldi, R. Alessandro, ja S. Pizzi, "5G NR system design: a consise survey of key features and capabilities," *Wireless Networks*, vol. 27, s. 5173–5188, 2021.

[21] K. Q. T. Zhang, *Wireless Communications: Principles, Theory and Methodology*. John Wiley & Sons Ltd, 2016.

[22] 3GPP, "NR; Physical channels and modulation (3GPP TS 38.221 version 18.4.0 Release 18)," ETSI Std., 2024.

[23] D. Tse ja P. Viswanath, *Fundamentals of Wireless Communication*. Cambridge University Press, 2005.

[24] 3GPP, "5G; NR; Physical channels and modulation (3GPP TS 38.211 version 18.4.0 Release 18)," ETSI Std., 2024.

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

[26] T. L. Marzetta, “Massive MIMO: An Introduction,” *Bell Labs Technical Journal*, vol. 20, s. 11–22, 2015.

[27] M. Jiang, J. Cezanne, A. Sampath, O. Shental, Q. Wu, O. Koymen, A. Bedewy, ja J. Li, “Wireless Fronthaul for 5G and Future Radio Access Networks: Challenges and Enabling Technologies,” *IEEE Wireless Communications*, vol. 29, no. 2, s. 108–114, 2022.

[28] P. Coussy, A. Baganne, ja E. Martin, “IP cores integration in DSP System-on-chip designs,” Published: *2002 11th European Signal Processing Conference*, 2002, s. 1–4.

[29] D. A. Patterson ja J. L. Hennessy, *Computer organization and design : the hardware/software interface*, RISC-V edition ed. Cambridge, MA: Morgan Kaufmann, 2018.

[30] A. S. Waterman, *Design of the RISC-V instruction set architecture.* University of California, Berkeley, 2016.

[31] E. Blem, J. Menon, ja K. Sankaralingam, “Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures,” Published: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 2013, s. 1–12.

[32] E. Cui, T. Li, ja Q. Wei, “RISC-V Instruction Set Architecture Extensions: A Survey,” *IEEE Access*, vol. 11, s. 24696–24711, 2023.

[33] M. Ali ja D. Göhringer, “Application specific instruction-set processors for machine learning applications,” Published: *2022 International Conference on Field-Programmable Technology (ICFPT)*, 2022, s. 1–4.

[34] J. L. Hennessy ja D. A. Patterson, *Computer architecture : a quantitative approach*, 5th ed., ser. Morgan Kaufmann series in computer architecture and design. Amsterdam ;: Morgan Kaufmann/Elsevier, 2012.

[35] D. Menard, G. Caffarena, J. A. Lopez, D. Novo, ja O. Sentieys, “Fixed-point refinement of digital signal processing systems.” *Digitally Enhanced Mixed Signal Systems*, s. 1–37, 2019.

[36] J.-M. Muller, N. Brunie, F. De Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, ja S. Torres, *Handbook of floating-point arithmetic.* Springer, 2018, vol. 1.

[37] Microprocessor Standards Committee, “IEEE standard for floating-point arithmetic,” IEEE Std 754-2019, 2019.

[38] M. D. L. Silva, L. Ramalho, I. Almeida, E. Medeiros, M. Berg, ja A. Klautau, “A new o-ran compression approach for improved performance on uplink signals,” *Journal of Communication and Information Systems*, vol. 37, no. 1, s. 30–41, 2022.

[39] H. Chang, J. Cho, ja W. Sung, "Performance evaluation of an SIMD architecture with a multi-bank vector memory unit," *2006 IEEE Workshop on Signal Processing Systems Design and Implementation*, s. 71–76, 2006.

[40] S. Larsen, E. Witchel, ja S. Amarasinghe, "Techniques for increasing and detecting memory alignment," 2001.

[41] J. Espinosa, C. Hernandez, J. Abella, D. de Andres, ja J. C. Ruiz, "Analysis and RTL correlation of instruction set simulators for automotive microcontroller robustness verification," Published: *Proceedings of the 52nd Annual Design Automation Conference*, 2015, s. 1–6.

[42] V. S. Chakravarthi, *A Practical Approach to VLSI System on Chip (SoC) Design : A Comprehensive Guide*, 2nd ed. Cham: Springer International Publishing, 2022.

[43] Y. Xu, B. Wang, E. Song, Q. Shi, ja T.-H. Chang, "Low-Complexity Channel Estimation for Massive MIMO Systems With Decentralized Baseband Processing," *IEEE Transactions on Signal Processing*, vol. 71, s. 2728–2743, 2023.

[44] H. Q. Ngo, *Massive MIMO: Fundamentals and system designs.* Linkopings Universitet (Sweden), 2015.

[45] J. Huang ja T. Leng, "Generalized loop-unrolling: a method for program speedup," Published: *Proceedings 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology. ASSET'99 (Cat. No.PR00122)*, 1999, s. 244–248.

[46] R. Schmogrow, B. Nebendahl, M. Winter, A. Josten, D. Hillerkuss, S. Koenig, J. Meyer, M. Dreschmann, M. Huebner, C. Koos, J. Becker, W. Freude, ja J. Leuthold, "Error vector magnitude as a performance measure for advanced modulation formats," *IEEE Photonics Technology Letters*, vol. 24, no. 1, s. 61–63, 2012.

[47] S. Gayathri ja T. C. Taranath, "Rtl synthesis of case study using design compiler," Published: *2017 International Conference on Electrical, Electronics, Communication, Computer, and Optimization Techniques (ICEECCOT)*, 2017, s. 1–7.