Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Tommy André Opstad

FPGA Implementation of a Real-Time Direction Finding System

Master's thesis in Eletronic Systems Design
Supervisor: Per Gunnar Kjeldsberg
Co-supervisor: Karl Emil Sandvik Bohne
June 2023

NTNU
Norwegian University of
Science and Technology

Tommy André Opstad

FPGA Implementation of a Real-Time Direction Finding System

Master's thesis in Eletronic Systems Design
Supervisor: Per Gunnar Kjeldsberg
Co-supervisor: Karl Emil Sandvik Bohne
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU
Norwegian University of
Science and Technology

Problem Description

Topic: Hardware Acceleration of a Real-Time Direction Finding System

The thesis deals with hardware acceleration of parts of Angle of Arrival (AoA) calculations (preferably the MUltiple SIgnal Classification (MUSIC) algorithm) using an Field Programmable Gate Array (FPGA). This is done using an existing or new locator that connects to an existing FPGA development board. The thesis also partly builds on work performed during the student project where the "tag" and associated software were developed. The critical aspect will be comparing such an implementation against a conventional software-based solution.

Assignment given: 16 January 2023
Supervisor: Per Gunnar Kjeldsberg, NTNU
Co-supervisor: Karl Emil Sandvik Bohne, EmLogic

Abstract

Bluetooth Direction Finding (BDF) was introduced by the Bluetooth Special Interest Group (SIG) with the Bluetooth 5.1 core specification. This new feature brings enhanced direction-finding capabilities that allow smart devices to pinpoint physical locations down to centimeter accuracy while maintaining the low cost and power requirements associated with such devices. BDF has several advantages over traditional location tracking technologies such as Global Navigation Satellite System (GNSS), Received Signal Strength Indication (RSSI), and Time of Flight (ToF), which are only capable of estimating the proximity of other devices and therefore unable to find the direction.

This thesis aims to develop and research the feasibility of using a Field Programmable Gate Array (FPGA) to accelerate the calculations required to determine the Angle of Arrival (AoA) of a Bluetooth tag and compare it to a traditional software implementation. A software model of the system has been implemented in MATLAB and Python, while the hardware design is written in VHDL. The design is synthesized and implemented on the Zynq Z2 FPGA using Vivavdo-2022.2. The main focus of the thesis is to implement the MUltiple SIgnal Classification (MUSIC) algorithm in hardware. MUSIC is a type of super-resolution direction-finding algorithm, and it consists of the following: perform Covariance Matrix Calculation (CMC) on the incoming samples and then conduct Eigenvalue Decomposition (EVD) to obtain the eigenvectors. The eigenvectors are, in turn, used to estimate the direction of the incoming signal using Spectral Peak Search (SPS). A Real-valued Transformation (RVT) method has been implemented to speed up and reduce the required hardware for calculating the eigenvectors by transforming complex values into real values without reducing accuracy. This method reduces the area and computational time by approximately 50%. The eigenvectors are computed using a state-of-the-art parallel method for solving the eigenvalue problem, which reduces the calculation time from 190 µs to 2.88 µs compared to the serial method.

The hardware implementation of the MUSIC algorithm can compute the azimuth and elevation of an incoming Bluetooth package in 6 µs at a clock frequency of 100 MHz. An equivalent Python and MATLAB program has an average run time of 112 ms and 22 ms, respectively. The algorithm implemented in hardware is approximately four orders of magnitude faster, and it can find the direction with an accuracy of 1° in two dimensions. The design utilizes 61%, 15%, 89%, 90%, and 1.35% of the available look-up tables, registers, Digital Signal Processing (DSP) slices, block ram, and F7 multiplexers, respectively. The hardware implementation draws 1.4 W of power compared to 60 W when running on a general-purpose computer.

Sammendrag

(Norwegian translation of the abstract)

Bluetooth Direction Finding (BDF) ble introdusert av Bluetooth Special Interest Group (SIG) med Bluetooth 5.1-kjernespesifikasjonen. Den nye funksjonen gir forbedret retningssøk, noe som gjør det mulig for smarte enheter å lokalisere fysiske steder ned til en nøyaktighet målt i centimeter. Samtidig kan enhetene opprettholde prisnivået og strømkravet knyttet til slike enheter. BDF har flere fordeler sammenlignet med tradisjonelle posisjonssporingsteknologier som Global Navigation Satellite System (GNSS), Received Signal Strength Indication (RSSI) og Time of Flight (ToF), som bare er i stand til å estimere nærheten til andre enheter og derfor ikke finne retningen.

Denne avhandlingen tar sikte på å utvikle og undersøke muligheten for å bruke en Field Programmable Gate Array (FPGA) for å øke hastigheten på beregningene som kreves for å bestemme Angle of Arrival (AoA) til en Bluetooth-tag og sammenligne den med en tradisjonell programvareimplementering. En programvaremodell av systemet er implementert i MATLAB og Python, mens maskinvaredesignet er skrevet i VHDL. Designet er syntetisert og implementert på en Zynq Z2 FPGA ved hjelp av Vivavdo-2022.2. Hovedfokus i oppgaven er å implementere MUltiple SIgnal Classification (MUSIC) algoritmen i maskinvare. MUSIC er en type retningssøkingsalgoritme med svært høy oppløsning, og den består av de følgende delene: utfør Covariance Matrix Calculation (CMC) på de innkommende målingene og deretter utføre Eigenvalue Decomposition (EVD) for å finne egenvektorene. Egenvektorene brukes deretter til å estimere retningen til det innkommende signalet ved hjelp av Spectral Peak Search (SPS). En Real-valued Transformation (RVT)-metode er implementert for å øke hastigheten og redusere den nødvendige maskinvaren for beregning av egenvektorene ved å transformere komplekse verdier til reelle verdier uten en reduksjon i nøyaktighet. Denne metoden reduserer arealet og kjøretiden med omtrent 50%. Egenvektorene beregnes ved hjelp av en state-of-the-art parallell metode for å løse egenverdiproblemet, noe som reduserer beregningstiden fra 190 µs til 2.88 µs sammenlignet med en seriell metode.

Implementeringen av MUSIC algoritmen i maskinvare kan beregne asimut og høyde av en innkomende Bluetooth pakke i løpet av 6 µs ved en klokkefrekvens på 100 MHz. Et tilsvarende Python- og MATLAB-program har en gjennomsnittlig kjøretid på henholdsvis 112 ms og 22 ms. Algoritmen implementert i maskinvare er omtrent fire størrelsesordener raskere, og den kan finne retningen med en nøyaktighet på 1° i to dimensjoner. Designet benytter henholdsvis 61%, 15%, 89%, 90% og 1% av de tilgjengelige oppslagstabellene, registrene, Digital Signal Processing (DSP)-skiver, blokkminne og F7 multiplekserere. Maskinvare implementasjonen bruker totalt 1.4 W med strøm i motseting til en personlig datamaskin som bruker 60 W.

Preface

This thesis is the final requirement to be awarded a Master of Science (MSc) in Electronic Systems Design at the Norwegian University of Science and Technology. The thesis continues the specialization project carried out in the preceding semester. The research and work in this thesis have been carried out on behalf of EmLogic and under the supervision of Professor Per Gunnar Kjeldsberg at the Faculty of Information Technology and Electrical Engineering. EmLogic is a consultancy and design house located in Asker and Trondheim, and they specialize in embedded hardware and software design. In addition to my contributions, there has also been another student, Jacob August Rangnes, that have also contributed to the project's success.

I want to thank my supervisor from NTNU for his guidance on academic writing and methodology. In addition, I would like to thank the great people at EmLogic, especially my external supervisor Karl Emil Sandvik Bohne, for his aid and technical expertise. I also thank Espen Flo Eriksen for his theoretical and mathematical knowledge.

Tommy André Opstad, June 2023
Norwegian University of Science
and Technology

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Abbreviations

AoA Angle of Arrival

AoD Angle of Departure

BDF Bluetooth Direction Finding

CORDIC COordinate Rotation DIgital Computer

CTE Constant Tone Extension

CMC Covariance Matrix Calculation

DSP Digital Signal Processing

DPE Diagonal Processing Element

ESPRIT Estimation of Signal Parameters via Rotational Invariant Techniques

EVD Eigenvalue Decomposition

FPGA Field Programmable Gate Array

GNSS Global Navigation Satellite System

GS Gauss Siedel

GFSK Gaussian Frequency-Shifting Key

HDL Hardware Description Language

HLM High Level Model

ISM Industrial, Scientific, and Medical

IQ In-Phase and Quadrature

MUSIC MUltiple SIgnal Classification

OPE Off-diagonal Processing Element

PS Processing System

PL Programmable Logic

PLAN Positioning, Localization, and Navigation

PPA Performance, Power and Area

PE Processing Element

RVT Real-valued Transformation

RSSI Received Signal Strength Indication

SPS Spectral Peak Search

SIG Bluetooth Special Interest Group

SIG Bluetooth Special Interest Group

SOR Successive-over-Relaxation

ToF Time of Flight

VPE Vector Processing Element

Chapter 1

Introduction

1.1 Motivation

Positioning, Localization, and Navigation (PLAN) technology has been subject to heavy research and investment since the rising popularity of positioning systems such as Global Navigation Satellite System (GNSS) [1]. Today these technologies are found everywhere in applications ranging from self-driving cars, naval and aerial navigation to mobile cell phones. Traditional PLAN technology has focused on outdoor positioning, but recently focus and attention has been shifted towards applying the technology to indoor applications [2]. Accurate indoor tracking has extensive applications in factory warehouses, logistics warehouses[3], indoor navigation at airports[4], hospitals, and other large facilities. Figure 1.1 [5] demonstrates how AoA can be applied to tracking assets in a warehouse. The indoor PLAN market is estimated to reach $28.2 billion by 2024 at an annual growth rate of 38.2% [6]. This, combined with the ever-growing amount of smart connected devices, has attracted significant interest in academia and the industry.

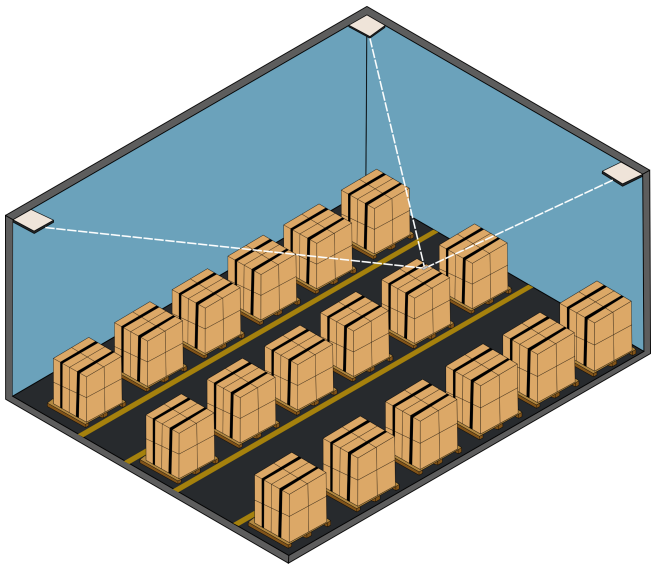


Figure 1.1: AoA applied to warehouse asset tracking

GNSS is a typical example of a PLAN technology that is very mature and in heavy use. Still, it has some significant drawbacks, such as the need for satellite reception and the lower accuracy over smaller distances, such as inside buildings and other indoor structures. Other methods for direction finding include Received Signal Strength Indication (RSSI) and Time of Flight (ToF) methods, but these methods can only estimate the general proximity of a device, and the accuracy is particularly poor indoors. Indoor positioning has proved difficult to implement despite the active research and development of new technologies [7]. Bluetooth Direction Finding (BDF) which was released in 2019 is an attempt from the Bluetooth Special Interest Group (SIG) to address these issues [8]. Figure 1.2 shows a typical architecture of an indoor Bluetooth Direction Finding system. The system consists of the following parts: a BDF transmitter, locator board, positioning engine, and user application. The BDF transmitter, which will be referred to as the tag from this point onward, transmits unique BDF signals, measured by a locator board with multiple antennas and a Bluetooth receiver. The receiver samples the antenna signals and sends the measurements to a positioning engine. The engine translates the measurements into the actual direction of one or more tags. The calculated direction can later be used inside a user application or stored in the cloud. This thesis will focus on data processing occurring inside the positioning engine.

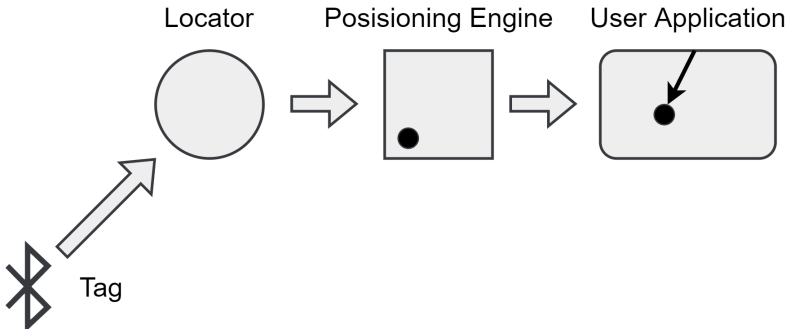


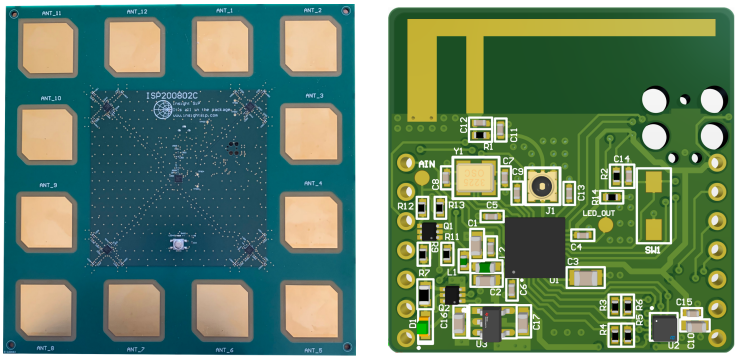
Figure 1.2: Typical architecture of a Bluetooth Direction Finding system

Real-time processing of a direction-finding system poses several challenges. These challenges include a large amount of processing power and the low latency needed for real-time performance. The large computational power required often prevents using microcontrollers, while general-purpose computers use a significant amount of power combined with non-deterministic execution behavior. A better solution could therefore be to employ dedicated hardware optimized for the task. The hardware acceleration of direction-finding can achieve faster and more predictable run-time than the other alternatives combined with low power consumption. This thesis will therefore investigate the feasibility of offloading the heavy calculation to an FPGA in order to achieve real-time performance.

1.2 Background

The Bluetooth Direction Finding technology is relatively new; hence, a limited amount of hardware and software is available. The technology has shown interest among prominent Bluetooth chip vendors like Nordic Semiconductor, Silicon Labs, and Texas Instruments. Still, few actors have been able to deliver a proper working system yet. This thesis continues the work completed during two specialization projects in the fall of 2022 [9, 10]. Both projects aimed to create a system capable of showing this new technology's possibilities.

During the preceding project thesis, a BDF tag was designed and tested with an existing locator board, as shown in Figure 1.3 [9]. In addition, another student, Jacob August Rangnes, worked on software development [10]. One project goal was to see if the movement read from an accelerometer could improve accuracy by combining the data with BDF technology. The data from the accelerometer was combined with an existing MUltiple SIgnal Classification (MUSIC) algorithm implemented in Python. The MUSIC algorithm is one of the more widely used algorithms in super-resolution direction finding, and it was introduced by Ralph O. Schmidt in 1986 [11]. The algorithm performs much better than traditional beam-forming algorithms when the incoming signals are not strongly correlated. Another good but less used algorithm is Estimation of Signal Parameters via Rotational Invariant Techniques (ESPRIT) [12], proposed initially as it is less compute-intensive than the MUSIC algorithm and does not require any search, making it faster than MUSIC. Still, MUSIC is more commonly used as it can achieve better accuracy, and it is more general considering the geometry of the locator board compared to ESPRIT. The project concluded that only using an accelerometer and the MUSIC algorithm gave little improvement due to the drift within the accelerometer. However, the accelerometer resulted in more efficient power usage as the system could stop transmitting data while staying still.



(a) ISP200802C Locator board [13] (b) Custom Bluetooth tag [9]

Figure 1.3: The locator board can be seen to the left, and the Bluetooth tag from the specialization project is on the right (not to scale)

Starting from the two specification projects, this thesis will attempt to improve the positioning engine by applying hardware acceleration. Hardware acceleration will allow the algorithm to run significantly faster as it is very computationally intensive, even for modern computers. The hardware acceleration of direction-finding algorithms is not new, and significant research has been done in the area [14]. Still, most research has been applied to applications other than BDF. Traditional applications of direction finding are sonar, astronomy, seismic event prediction, wireless communication systems, and radar [15, 16], but the same technology can be applied to BDF. In addition to faster computation time, utilizing hardware acceleration will also enable the system to increase the accuracy by using larger BDF sample sets which are essential for the system's accuracy since sample size and precision are highly related. This is especially true for indoor applications where signal reflections degrade performance compared to outdoor ones.

Several improvements have been introduced since the original MUSIC algorithm, such as Keh-Chirang Huarng and Chien-Chung Yeh [17], which in 1991 introduced a method for a real-valued MUSIC algorithm for Uniform Linear Arrays (ULA). The real-valued MUSIC algorithm transforms the complex samples into real-valued samples while still achieving

good accuracy [17]. The real-valued MUSIC algorithm has several benefits over the original algorithm, such as reduced storage requirements, routing complexity, and improved speed. A significant speedup can be gained in solving the eigenvalue problem using real matrices compared to complex [18]. This relates to the difficulty of finding the eigenvalue and eigenvector of complex matrices compared to real-valued matrices. Additional benefits are also found in the peak search, which is applied after the eigenvalue problem to find the desired direction. Wei Zhang et al. [19] introduced a method for real-valued transformation using Uniform Rectangular Arrays (URA) in 2014. The benefit of the URA is that it is a two-dimensional antenna array instead of the one-dimensional ULA, which enables the possibility of finding the direction in two dimensions instead of one.

1.3 Main Objectives

The overall goal of this thesis can be summarized in the following objectives:

1 Investigate the feasibility of the MUSIC algorithm for AoA calculations with a focus on accuracy, real-time performance, and power usage.

2 Implement selected parts of a suitable algorithm in a Hardware Description Language (HDL) such as VHDL and optimize the design concerning Performance, Power and Area (PPA).

3 Comparison between a hardware-accelerated and pure software implementation focusing on speedup and accuracy.

1.4 Project Scope and Limitations

Time has been a constant limitation for the project since the MUSIC algorithm is not trivial to implement in hardware. The project is divided among two students since the entire project's scope is relatively large. Due to the limited time, the data had to be modeled in software instead of the real-world data generated by the locator board. This also means the time it takes to move data to and from the FPGA was not considered. The selected FPGA also had limited hardware for implementing a digital processing system, resulting in having to deal with constant compromises between performance and feasibility.

1.5 Methodology

The research strategy is shown in Figure 1.4. The first step of the research is to complete a literature search where similar work is investigated. This work includes investigating existing algorithms, such as the MUSIC algorithm for AoA calculations. The architectural exploration phase is an important step where different methods are studied and tested. This step also involves searching for possible optimizations and exploitation. Architectural exploration can typically be done by modeling the desired system in a high-level language such as MATLAB or Python. These models are sometimes referred to as a High Level Model (HLM) as they attempt to describe the design in an abstract language. In addition to finding possible

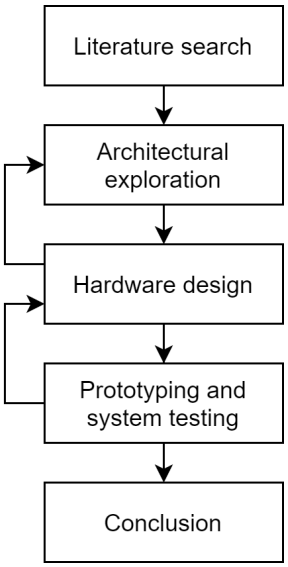


Figure 1.4: Research strategy

optimization, the model can also be used to generate test and verification data to ensure the correctness of the design. Microarchitectures can also be created during the architectural exploration that maps the HLM to the hardware. After the microarchitecture, the design can be implemented in hardware using HDL. The HDL can be simulated and synthesized using software tools such as Modelsim and Vivado. Finally, the system can be integrated and tested.

1.6 Thesis Outline

The thesis is laid out in the following chapters:

Chapter 2 - Background Theory presents some useful background theory.

Chapter 3 - Hardware Implementation of the MUSIC Algorithm presents the design and implementation of the MUSIC algorithm on hardware. Design choices and their respective advantages and disadvantages are also discussed.

Chapter 4 - Results presents the resource usage and performance of the system.

Chapter 5 - Discussion presents a discussion on different trade-offs. The results in the previous chapter are also discussed.

Chapter 6 - Conclusion summarizes key metrics and findings. Future work is also discussed.

Chapter 2

Background Theory

This chapter will present background theory that is helpful for later chapters. The chapter introduces the theory behind BDF, a brief introduction to linear algebra, the CORDIC and MUSIC algorithm, and finally some information about the FPGA used in this thesis.

2.1 Bluetooth Direction Finding

In 2019 the Bluetooth Special Interest Group (SIG) introduced the Bluetooth 5.1 specification [8], which added support for direction finding using antenna arrays. The new feature can provide highly accurate direction finding both indoors and outdoors. Bluetooth is a technical standard for short-range communication, and it utilizes the 2.4 GHz Industrial, Scientific, and Medical (ISM) frequency band [20]. The band operates over a frequency from 2400 to 2483.5 MHz. Figure 2.1 [9] shows how the frequency band is divided into 40 channels with a 2 MHz spacing. The advertising channels are indicated in black and are used for advertising. The advertising channels are only used for advertising, while the data channels can be used for data transmission and advertising. The advertising channels are placed to avoid overlapping with the WIFI channels 1, 6, and 11 of the IEEE 802.11 standard, indicated in a darker shade of gray in the figure. Placing the advertising channels outside the WIFI channels helps reduce congestion and ensure that advertisements succeed.

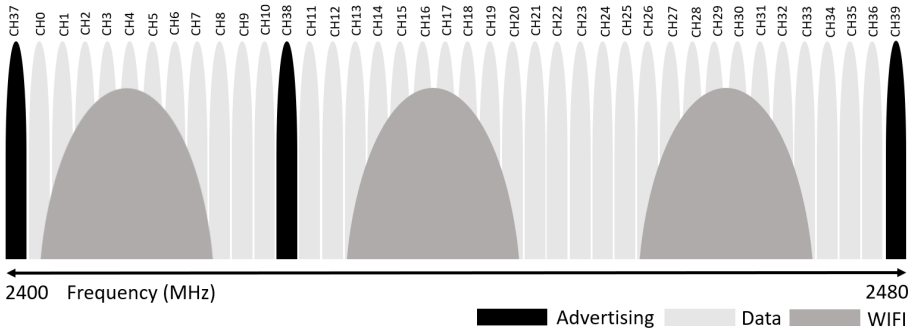


Figure 2.1: Bluetooth Low Energy frequency spectrum

Bluetooth uses Gaussian Frequency-Shifting Key (GFSK) to modulate the signal, which means that the frequency deviates with a positive offset for a binary 1 and a negative offset for a binary 0. With Bluetooth operating at 1 Mb/s, the nominal deviation is 250 KHz, and the average deviation should be between 225 KHz and 275 KHz.

The Bluetooth 5.1 core specification introduced direction-finding capabilities by appending a Constant Tone Extension (CTE) at the end of a traditional Bluetooth packet transmission, as shown in Figure 2.2 [8]. The CTE is a stream of binary ones that results in a sine wave at a fixed frequency. The antenna array and receivers can measure the phase of the incoming sine wave and later calculate the phase difference. The Bluetooth protocol usually employs a whitening technique to avoid constant binary ones being sent by replacing them with zeroes and encoding the data. However, this is naturally not applied to the CTE extension.

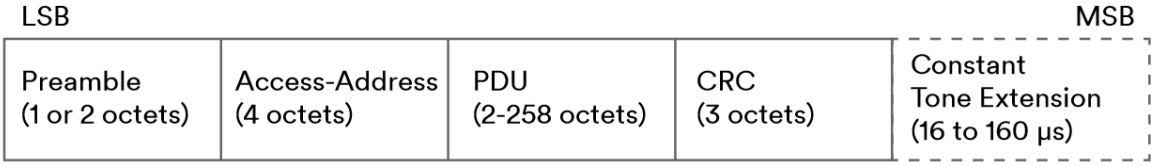


Figure 2.2: Bluetooth packet with CTE append at the end

There exist two types of Bluetooth Direction Finding. The first mode is called Angle of Arrival (AoA) and the second one Angle of Departure (AoD). In AoA, the moving tag that is tracked transmits the CTE signals while another device calculates the direction of the received signal using an antenna array. In AoD, the CTE is transmitted by the antenna array, while the moving tag measures the phase difference and calculates the direction. Each type has its respective advantages and drawbacks. In AoD, the transmitted CTE is switched between antenna elements which can draw significant power compared to AoA. The disadvantage of AoA is that multiple tags are to be located, requiring all tracked devices to transmit data to the receiving array. This might cause issues if multiple tags are transmitted simultaneously. The effect of this can be reduced by randomizing the time when each tag transmits, but the problem will still grow as the number of tags increases. On the other hand, AoD only requires that the array transmit and all devices receive, allowing an unlimited number of devices to estimate their direction from the array. However, if multiple arrays are to be used to track a single moving tag, AoA only requires the tag to transmit while all arrays sample simultaneously. For AoD, each antenna array would have to transmit separately instead of simultaneously. This would mean the direction estimates would be calculated with data separated in time.

2.1.1 Bluetooth Angle of Arrival Sampling

In BDF, the receiver takes several phase and amplitude measurements of the incoming sine wave at precise intervals in a process known as In-Phase and Quadrature (IQ) sampling. The phase and amplitude of the IQ sample are a set of Cartesian coordinates. When the receiver performs the IQ sampling, each sample has to be attributed to a specific antenna in the array. Figure 2.3 visually represents an IQ sample and its corresponding angle φ.

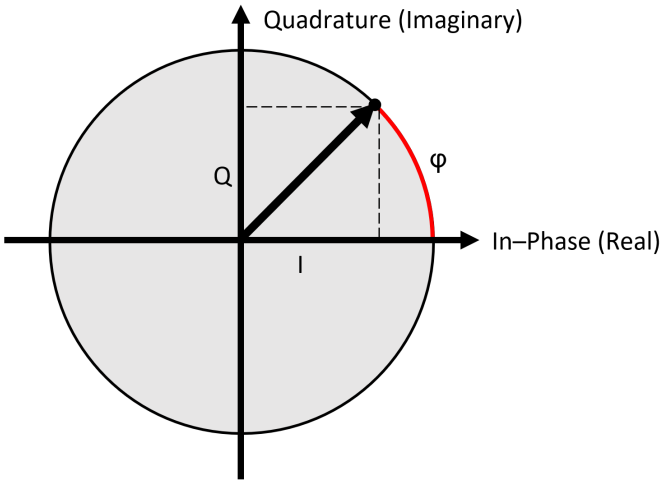


Figure 2.3: IQ sample in Cartesian coordinates

Figure 2.4 [8] shows an overview of the CTE field for the transmitter and receiver in AoA. The transmitter transmits a constant sine wave while the receiver sample the signal on the different antenna elements. The first 4 µs is designed to ensure that there is a gap between adjacent transmissions so that they do not overlap with each other. Eight IQ samples are taken from the first antenna at 1 µs intervals during the reference period. The receiver might use these samples to estimate the signal frequency and, in turn, calculate the wavelength. The sample and switch slots can be configured from 12 to 160 µs [8]. The duration of the CTE will, together with the sampling configuration, determine how many samples can be achieved during each transmission.

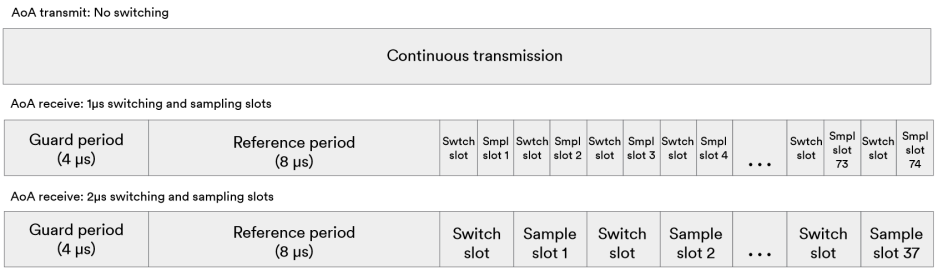


Figure 2.4: CTE timing rules for AoA [8]

Both the number of sampling slots and sampling duration can be configured. The sampling time for each antenna can be either 1 or 2 µs. Using 1 µs time slots would yield double the number of samples compared to using 2 µs for the same CTE length. However, using a 2 µs sampling slot does not increase the sampling window, as shown in Figure 2.5 [8]. The extra time in the sampling slot can be used to allow the signal to stabilize before sampling.

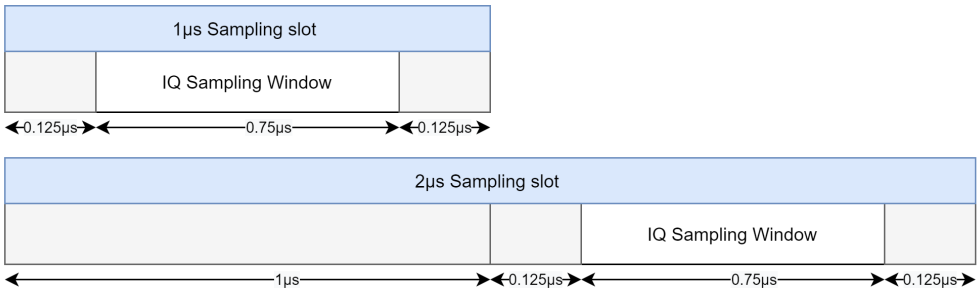


Figure 2.5: CTE sampling window

In an ideal world, every antenna would be sampled simultaneously, requiring multiple processors to read the samples. Most applications, therefore, allow for reduced accuracy to reduce system cost [21, 22]. A typical method for reading multiple antennas using one receiver is to utilize an RF switch that can switch between different antenna elements. One drawback of using an RF switch is that it takes time for the RF signal to become stable. Consider the array with elements numbered as shown in Figure 2.6. If we would sample each antenna element in a top-to-bottom order, Equation 2.1 shows that it would take 48 µs to sample each element once with a sampling period of 2 µs since each sampling slot requires a 2 µs switching slot.

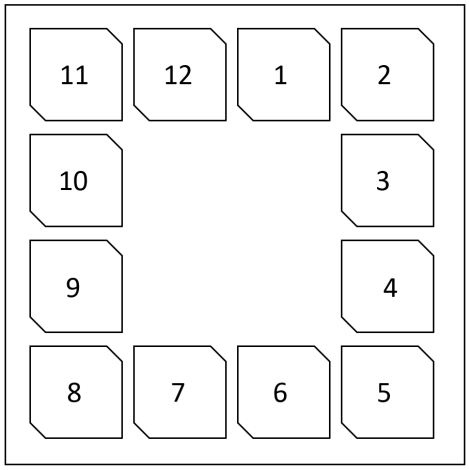


Figure 2.6: Locator board sampling slots with 12 antenna elements

$$t_{sampling} = (2\ \mu s + 2\ \mu s) \times 12 = 48\ \mu s \tag{2.1}$$

Figure 2.4 shows that the total CTE time is 160 µs where 12 µs is used for the 4 µs guard and 8 µs reference sampling periods. This leaves a 148 µs sample and switch period that can be freely used for sampling antenna elements, allowing each element to be sampled three times as shown in Equation 2.2. There are two possible orders to sample the antenna elements, and the first is to loop through each antenna element and then repeat the cycle. The advantage of this method is that each sample of different antenna elements will be measured closer together in time but at the cost of increased switching. The second method involves sampling each element multiple times, eliminating switching between antenna elements between samples of the same element. Suppose we sample the same element once in each sample slot and use two

successive sampling slots for the same element. In that case, we gain an additional measurement from the switching slot in-between, thus increasing the total number of measurements. The second method also has increased accuracy since less switching is required and therefore a more stable RF signal.

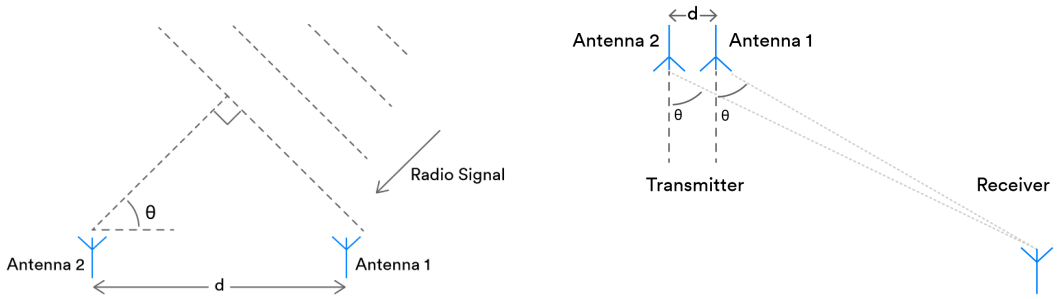
$$N = \frac{160\ \mu s - 12\ \mu s}{(2\ \mu s + 2\ \mu s) \times 12} = 3.08 \tag{2.2}$$

2.1.2 Calculating the Angle of Arrival Between Two Antennas

Figures 2.7(a) and 2.7(b) [8] show an example of AoA and AoD, respectively. In the AoA example, one radio signal will hit two receiving antennas. The radio signal will propagate toward the antennas, but the wave will hit each element with a different phase. IQ sampling on each antenna allows us to calculate the phase difference Ψ easily. Since the distance, d, is known, we can calculate the signal's angle θ using a trigonometric identity as shown in Equation 2.3 [8].

$$\theta = arctan(\frac{\Psi\lambda}{2\pi d}) \tag{2.3}$$

λ is the wavelength of the received signal where $\lambda = \frac{c}{f_c}$ and c is the speed of the light and f_c is the center frequency of the signal. Ψ is the phase difference between the two antennas, and the distance d is chosen to be relatively close to λ. A typical value of d is $\lambda/2$.



(a) Angle of Arrival (b) Angle of Departure

Figure 2.7: 2D example of using phase difference to derive angle of arrival

Using a one-dimensional antenna array such as the one in Figure 2.7(a) allows us to calculate the angle in one dimension, resulting in us being able to determine the direction in two dimensions. Expanding the antenna array to two dimensions allows us to calculate the angle in two dimensions. We refer to these angles as azimuth and elevation.

2.2 Introduction to Linear Algebra

This section introduces some basic matrix operations and notations that will be used later in the thesis. The majority of the theory in this chapter is from Linear Algebra and its Applications by David Lay et al. [23] and Notes on Kronecker Products by Louis Whitcomb [24] unless otherwise stated.

2.2.1 Matrix Multiplication

If A is an $m \times n$ matrix, and if B is an $n \times p$ matrix with columns $b_1, ..., b_p$, then the product AB is the $m \times p$ matrix whose columns are $Ab_1, ..., Ab_p$ as defined in [23, p. 113]. The formula for matrix multiplication is shown in Equation 2.4

$$AB = A[b_1 b_2 \cdots b_p] = [Ab_1 Ab_2 \cdots Ab_p] \tag{2.4}$$

A simple example is shown in Equation 2.5.

$$\begin{bmatrix} 2 & 3 \\ 2 & -5 \end{bmatrix} \times \begin{bmatrix} 4 & 3 & 7 \\ 1 & -2 & 3 \end{bmatrix} = \begin{bmatrix} 11 & 0 & 23 \\ 3 & 16 & -1 \end{bmatrix} \tag{2.5}$$

Note that the number of columns in A must match the number of rows in B in order to multiply the two matrices [23, p. 113].

2.2.2 Matrix Transpose

The matrix transpose of a matrix A is defined by [23, p. 117] as A^T where rows are exchanged for columns, and vice versa. This operation is shown with an example in Equation 2.6.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \tag{2.6}$$

2.2.3 Complex Conjugate

The complex conjugate of a complex number is formed by changing the sign of the imaginary component [23, A3]. Given a complex number of the form:

$$z = a + bi \tag{2.7}$$

where a is the real component and b is the imaginary component, the complex conjugate $\bar{z}$ is:

$$\bar{z} = a - bi \tag{2.8}$$

2.2.4 Hermitian matrix

A hermitian matrix is a complex square matrix that combines both operations from Equation 2.6 and 2.8; that is, the hermitian matrix is the conjugate transpose of the original matrix [24]. Equation 2.9 shows the general characteristics of a hermitian matrix.

$$\begin{bmatrix} 1 & 4+1i & 5-2i \\ 4-1i & 2 & 6+3i \\ 5+2i & 6-3i & 3 \end{bmatrix} \tag{2.9}$$

A hermitian matrix has some special characteristics, such as the diagonal, which always contains only real values, and the complex element located in the i-th row and the j-th column must be the complex conjugate of the element that is in the j-th row and i-th column [24]. Another characteristic is that the eigenvalues of a Hermitian matrix are always real [24].

2.2.5 Kronecker Product

The Kronecker product is a binary matrix operator that maps two arbitrarily dimensional matrices into a larger matrix with a special block structure [24]. Given the $n \times m$ matrix $A_{n\times m}$ and the $p \times q$ matrix $B_{p\times q}$:

$$A = \begin{bmatrix} a_{1,1} & \cdots & a_{1,m} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,m} \end{bmatrix}, B = \begin{bmatrix} b_{1,1} & \cdots & b_{1,q} \\ \vdots & \ddots & \vdots \\ b_{p,1} & \cdots & b_{p,q} \end{bmatrix} \tag{2.10}$$

The Kronecker product, denoted $A \otimes B$ is the $np \times mq$ matrix with the block structure:

$$A \otimes B \begin{bmatrix} a_{1,1}B & \cdots & a_{1,m}B \\ \vdots & \ddots & \vdots \\ a_{n,1}B & \cdots & a_{n,m}B \end{bmatrix} \tag{2.11}$$

For example, given:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, B = \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} \tag{2.12}$$

the Kronecker product $A \otimes B$ is:

$$A \otimes B = \begin{bmatrix} 1\begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} & 2\begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} \\ 3\begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} & 4\begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 0 & 5 & 0 & 10 \\ 6 & 7 & 12 & 14 \\ 0 & 15 & 0 & 20 \\ 18 & 21 & 24 & 28 \end{bmatrix} \tag{2.13}$$

2.2.6 Identity and Exchange Matrix

The identity I matrix is a square matrix of size $n \times n$ with ones on the main diagonal and zeroes everywhere [23, p. 55]. Equation 2.14 shows an identity matrix I with $n = 3$.

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.14}$$

We also have the exchange matrix J, which is similar to the identity matrix but has the ones on the anti-diagonal instead of the diagonal [23, p. 55], as shown in Equation 2.15

$$J = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \tag{2.15}$$

2.2.7 Persymmetric Matrix

A persymmetric matrix refers to a square matrix that is symmetric with respect to the northeast-to-southwest diagonal. Let $A = a_{ij}$ be an $n \times n$ matrix. A persymmetric matrix has the following requirement:

$$a_{ij} = a_{n-j+1,n-i+1} \quad for \quad all \quad i, j \tag{2.16}$$

An example of a persymmetric matrix:

$$\begin{bmatrix} 8 & 9 & 10 & 4 \\ 6 & 7 & 3 & 10 \\ 5 & 2 & 7 & 9 \\ 1 & 5 & 6 & 8 \end{bmatrix} \tag{2.17}$$

2.2.8 Unitary Matrix

A unitary matrix is a square matrix of complex numbers whose inverse is equal to its conjugate transpose [25]. In other words, the product of the unitary matrix and the conjugate transpose of a unitary matrix is equal to the identity matrix [25]. A matrix is said to be unitary if the following condition is true:

$$UU^H = I \tag{2.18}$$

An example of a unitary matrix:

$$U = \frac{1}{2}\begin{bmatrix} 1+i & 1-i \\ 1-i & 1+i \end{bmatrix}, U^H = \frac{1}{2}\begin{bmatrix} 1-i & 1+i \\ 1+i & 1-i \end{bmatrix} \tag{2.19}$$

$$UU^H = \frac{1}{2}\begin{bmatrix}1+i & 1-i\\ 1-i & 1+i\end{bmatrix}\frac{1}{2}\begin{bmatrix}1-i & 1+i\\ 1+i & 1-i\end{bmatrix} = \frac{1}{4}\begin{bmatrix}4 & 0\\ 0 & 4\end{bmatrix} = I \tag{2.20}$$

2.2.9 Eigenvectors and Eigenvalues

An eigenvector of an $n \times n$ matrix A is a nonzero vector x such that $Ax = \lambda x$ for some scalar λ [23, p. 285]. A scalar λ is called an eigenvalue of A if there is a nontrivial solution x of $Ax = \lambda x$; such an x is called an eigenvector corresponding to λ [23, p. 285]. Assume that we have a matrix A defined as:

$$A = \begin{bmatrix}1 & 6\\ 5 & 2\end{bmatrix} \tag{2.21}$$

The eigenvalues of A can be calculated as shown in Equation 2.22.

$$\det(\lambda I - A) = \begin{bmatrix}1-\lambda & 6\\ 5 & 2-\lambda\end{bmatrix} = \lambda^2 - 3\lambda - 28 = (\lambda - 7)(\lambda + 4) \tag{2.22}$$

$$\lambda_1 = 7, \lambda_2 = -4 \tag{2.23}$$

The eigenvalues of A are 7 and -4, as shown in Equation 2.23. The eigenvectors can now be calculated for each eigenvalue by solving the following equation:

$$(A - \lambda I)v = 0 \tag{2.24}$$

The eigenvector corresponding to the eigenvalue $\lambda = 7$ is:

$$\begin{bmatrix}1-\lambda & 6\\ 5 & 2-\lambda\end{bmatrix}\begin{bmatrix}v_1\\ v_2\end{bmatrix} = \begin{bmatrix}-6v_1 + 6v_2\\ 5v_1 - 5v_2\end{bmatrix} = \begin{bmatrix}0\\ 0\end{bmatrix} \Rightarrow \begin{bmatrix}v_1\\ v_2\end{bmatrix} = \begin{bmatrix}1\\ 1\end{bmatrix} \tag{2.25}$$

The eigenvector corresponding to the eigenvalue $\lambda = -4$ is:

$$\begin{bmatrix}1-\lambda & 6\\ 5 & 2-\lambda\end{bmatrix}\begin{bmatrix}v_1\\ v_2\end{bmatrix} = \begin{bmatrix}5v_1 + 6v_2\\ 5v_1 + 6v_2\end{bmatrix} = \begin{bmatrix}0\\ 0\end{bmatrix} \Rightarrow \begin{bmatrix}v_1\\ v_2\end{bmatrix} = \begin{bmatrix}-6\\ 5\end{bmatrix} \tag{2.26}$$

It is also possible to check if the eigenvectors are correct. Let us define u and v as the following:

$$u = \begin{bmatrix}6\\ -5\end{bmatrix}, v = \begin{bmatrix}3\\ -2\end{bmatrix} \tag{2.27}$$

We can check if u and v are eigenvectors of A:

$$Au = \begin{bmatrix} 1 & 6 \\ 5 & 2 \end{bmatrix} \begin{bmatrix} 6 \\ -5 \end{bmatrix} = \begin{bmatrix} -24 \\ 20 \end{bmatrix} = -4 \begin{bmatrix} 6 \\ -5 \end{bmatrix} = -4u \tag{2.28}$$

$$Av = \begin{bmatrix} 1 & 6 \\ 5 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ -2 \end{bmatrix} = \begin{bmatrix} -9 \\ 11 \end{bmatrix} \neq \lambda \begin{bmatrix} 3 \\ -2 \end{bmatrix} \tag{2.29}$$

We can see from Equation 2.28 and 2.29 that u is an eigenvector corresponding to an eigenvalue -4, but that v is not an eigenvector of A because Av is not a multiple of v.

2.3 The MUltiple SIgnal Classification Algorithm

The MUltiple SIgnal Classification (MUSIC) algorithm is a type of super-resolution subspace method. Subspace methods are based on the spectral decomposition of the covariance matrix of the received IQ samples [26]. Using an eigenvalue decomposition method, they separate the covariance matrix into signal and noise subspaces [26]. The MUSIC algorithm was first proposed by Schmidt R.O in 1986 [11] to estimate the direction of arrival of signals. The method can detect multiple incoming sources and performs better than the conventional beam-forming methods providing that these sources are uncorrelated or weakly correlated [26]. The MUSIC algorithm can estimate sources in both one- and two-dimensional space. The algorithm's capabilities depend on the antenna array's physical layout, which measures incoming signals. Two commonly used antenna array structures are Uniform Linear Array (ULA) which has a one-dimensional structure, and Uniform Rectangular Array (URA), which has a two-dimensional structure. The former can determine the direction in one dimension, while the latter is capable of two-dimensional direction finding. This thesis will use the latter, which can simultaneously accurately predict the azimuth (φ) angles and the elevation (θ) of multiple signals. The antenna array structure with an M number of antenna elements is shown in Figure 2.8. The MUSIC algorithm can estimate $D(0 < D < M)$ independent signal sources. The Angle of Arrival of each signal source is represented as $(\theta_k, \varphi_k), k = 1, 2, ..., D$, where $\theta_k(0 < \theta < \pi/2)$ and $\varphi_k(0 < \varphi < 2\pi)$.

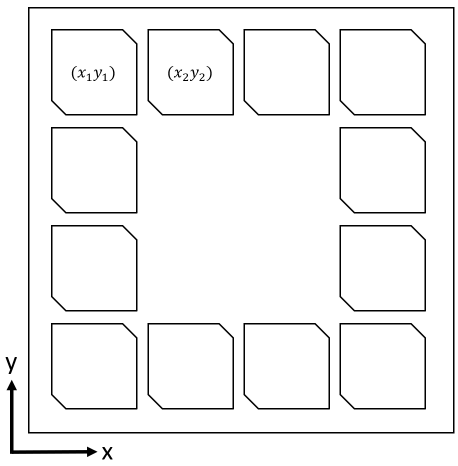


Figure 2.8: Antenna array coordinate frame where M_x and M_y are the x-coordinate and y-coordinate of each array element M, respectively.

Equation 2.30 shows a typical measurement model for estimating the direction based on phase measurements for a signal downconverted to baseband before sampling [14]:

$$x(t) = A(\theta, \varphi)s(t) + n(t) \tag{2.30}$$

where $A(\theta, \varphi) = [a(\theta_1, \varphi_1), a(\theta_2, \varphi_2), ..., a(\theta_D, \varphi_D)]$ is the matrix of steering vectors for each signal, $s(t)$ is the signal vector and $n(t)$ is the white Gaussian noise vector. For 2D-direction finding, the steering vector $a(\theta, \varphi)$ is defined in Equation 2.31 where λ is the wavelength of the signals.

$$a(\theta_k, \varphi_k) = \exp\left(\frac{2\pi j}{\lambda} x_n sin(\theta_k) cos(\varphi_k) + y_n sin(\theta) sin(\varphi)\right) \tag{2.31}$$

The covariance matrix of the received signals is computed as shown in Equation 2.32:

$$R_{XX} = E[X(t)X^H(t)] = AR_S A^H + \sigma^2 I \tag{2.32}$$

where $(\ \cdot\ )^H = Hermitian$, $E(\ \cdot\ ) = Expected\ value$ in the statistical average, R_S is the signal covariance matrix and σ^2 is the noise variance. In practice, we can estimate the covariance matrix from a finite number of temporal samples in the form shown in Equation 2.33. N refers to the number of snapshots taken, each containing a number of samples equal to the number of antenna elements.

$$R_{XX} = \frac{1}{N}\sum_{t=1}^{N} X(t)X(t)^H \tag{2.33}$$

Equation 2.34 shows an example of how the sample matrix X might look. In this example, there are four antenna elements and three snapshots taken.

$$X = \begin{bmatrix} \overbrace{sample\ 1}^{\text{snapshot 1}} & \overbrace{sample\ 1}^{\text{snapshot 2}} & \overbrace{sample\ 1}^{\text{snapshot 3}} \\ sample\ 2 & sample\ 2 & sample\ 2 \\ sample\ 3 & sample\ 3 & sample\ 3 \\ sample\ 4 & sample\ 4 & sample\ 4 \end{bmatrix} \begin{matrix} ANT1 \\ ANT2 \\ ANT3 \\ ANT4 \end{matrix} \tag{2.34}$$

The covariance matrix will have M number of eigenvalues $(\lambda_1, \lambda_2, ..., \lambda_M)$ and their corresponding eigenvectors are $E = e_1, e_2, ..., e_M$. If the eigenvalues are sorted from largest to smallest, we can divide the matrix E into two sub-spaces $[E_N E_S]$. The noise subspace E_N comprises $M - D$ eigenvectors associated with the noise. The signal subspace E_S contains D eigenvectors associated with the arriving signals. The noise subspace eigenvectors are orthogonal to the array steering vectors at the angles of arrival. The MUSIC algorithm work by searching for all arrival vectors that are orthogonal to the noise subspace. To search, MUSIC constructs the following pseudo-spectrum function:

$$P(\theta, \varphi) = \frac{1}{A^H(\theta, \varphi) E_N E_N^H a(\theta, \varphi)} \tag{2.35}$$

where $a(\theta, \varphi)$ is the steering vector. To find the AoA, evaluate $P(\theta, \varphi)$ for desired values of $\theta_k(0 < \theta < \pi/2)$ and $\varphi_k(0 < \varphi < 2\pi)$. This is done by substituting $\theta_1, ..., \theta_m$ and $\varphi_1, ..., \varphi_m$ in $P(\theta, \varphi)$. The resolution of the AoA estimation will directly depend on the number of angles for which $P(\theta, \varphi)$ will be evaluated. Find N largest peaks in $(\theta_k, \varphi_k), k = 1, 2, ..., D,$. The associated peaks will be the estimated AoA.

2.4 The Real-Valued MUSIC Algorithm

The covariance matrix is generated as shown in Equation 2.36. The covariance matrix naturally contains complex values since the IQ samples are a set of Cartesian coordinates. Complex values are not ideal for a hardware implementation as they complicate the calculation of eigenvalues, eigenvectors, and the peak search [17]. This will result in a larger area, routing usage, and increased complexity [17]. Keh-Chirang Huarng and Chien-Chung Yeh [17] have presented a method for transforming a complex covariance matrix into a real-valued matrix using a real-valued transformation.

$$R_{XX} = \frac{1}{N} \sum_{t=1}^{N} X(t) X(t)^H \tag{2.36}$$

A prerequisite for the real-value transformation is that the covariance matrix must be both hermitian and persymmetric. The matrix in Equation 2.36 is hermitian, but not persymmetric. The matrix can be converted into a hermitian and persymmetric matrix using forward-backward averaging as shown in Equation 2.37 [17]:

$$R_{FB} = \frac{1}{2}(R + JR^T J) \tag{2.37}$$

where R is the covariance matrix R_{XX}, R^T is transposed of R, and J is the exchange matrix with all unities on the secondary diagonal positions and zeroes elsewhere. If we apply Equation 2.37, it is possible to reduce the complex matrix into a real matrix. The method utilizes the fact that R_{FB} is both hermitian and persymmetric, in the way shown in Equation 2.38 where $\bar{R}$ is the conjugate of R and J is an exchange matrix.

$$J\bar{R}J = R \tag{2.38}$$

If we assume that M is even, we can let U be defined as an $M \times M$ matrix as:

$$U = \frac{1}{\sqrt{2}} \begin{bmatrix} I & J \\ iJ & -iI \end{bmatrix} \tag{2.39}$$

where I is the identity matrix and J is the exchange matrix respectively. It can be proved that for any $M \times M$ hermitian persymmetric matrix R, URU^H is real and symmetric since R is also Hermitian and symmetric [17]. Using this fact, we can create a symmetric real-valued covariance matrix $\hat{R}$. Since the covariance matrix is real and symmetric, it is a known fact that the eigenvalues and eigenvectors of the covariance matrix will also be real [17]. Equation 2.40 shows the premultiplication and postmultiplication of R by U and U^H. The real-valued

correlation matrix $\hat{R}$ can be obtained using Equation 2.40 where the imaginative component has been removed, and all the information is stored in the real component.

$$\hat{R} = Re\left\{URU^H\right\} \tag{2.40}$$

The steering vector used in the original MUSIC algorithm is no longer valid since the covariance matrix has been transformed into a real-valued matrix. Keh-Chirang Huarng and Chien-Chung Yeh [17] have shown that for ULA, the new steering vector becomes:

$$\tilde{a}_x(\omega) = U_x e^{-j(M_x-1)\alpha/2} a_x(\omega) \tag{2.41}$$

$$\tilde{a}_y(\psi) = U_y e^{-j(M_y-1)\beta/2} a_y(\psi) \tag{2.42}$$

for the steering vector in the x-direction and y-direction, respectively. We can define $\omega = sin(\theta)cos(\varphi)$, $\psi = sin(\theta)cos(\varphi)$, $\alpha = \exp(j\frac{2\pi d_x}{\lambda}\omega)$, and $\beta = \exp(j\frac{2\pi d_y}{\lambda}\psi)$. Wei Zhang et al. [19] have shown that we can obtain the real-valued steering vector for a URA-type antenna array by calculating the Kronecker product of $\tilde{a}_x(\omega)$ and $\tilde{a}_y(\psi)$:

$$\tilde{a}(\theta,\varphi) = \tilde{a}_x(\omega) \otimes \tilde{a}_y(\psi) \tag{2.43}$$

A new real-valued search function can be found by applying the new steering vector into the original MUSIC [11] search function by Schmidt [19]:

$$P(\theta,\varphi) = \frac{1}{[\tilde{a}_x(\omega) \otimes \tilde{a}_x(\psi)]E_N E_N^H [\tilde{a}_y(\omega) \otimes \tilde{a}_y(\psi)]} \tag{2.44}$$

2.5 The COordinate Rotation DIgital Computer Algorithm

This section presents the COordinate Rotation DIgital Computer (CORDIC) algorithm that can be used to calculate several trigonometric functions. The algorithm was first presented in an article by J.E. Volder in 1959 [27], and it was later improved upon by adding support for hyperbolic calculations in 1971 [28], and then the generalized hyperbolic CORDIC was introduced in 2019 [29]. CORDIC is suitable for hardware because it can be implemented using only a combination of additions, subtractions, shifting, and multiplication. CORDIC can perform a vector rotation by using a sequence of iterative micro-rotations of an elementary angle where the original rotation angle θ can be expressed by the sum of all elementary angles ϕ as shown in Equation 2.45 and Figure 2.9.

$$\theta = \sum_{i=0}^{\infty} \phi \tag{2.45}$$

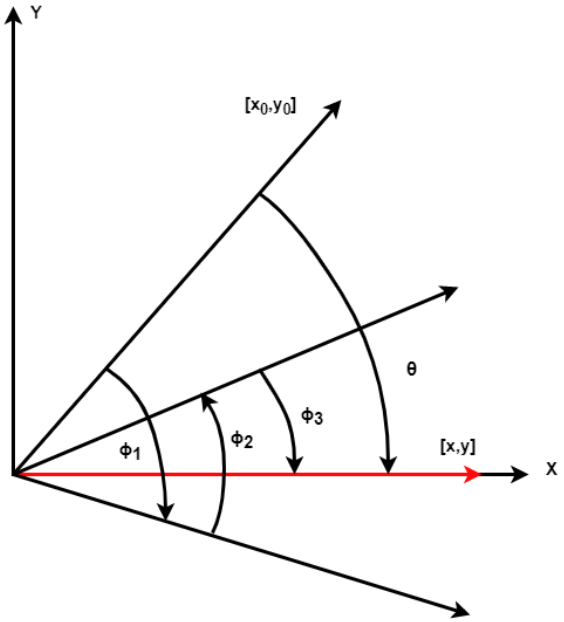


Figure 2.9: A vector $[x_0, y_0]$ rotated to $[x, y]$ using micro-rotations [30].

The following section is based on the survey by Andraka from 1998 [31]. The objective of the section is to explain how the CORDIC calculates trigonometric functions using simple arithmetics in an iterative process. Figure 2.9 presents two vectors of the same magnitude. The second vector is obtained by rotating the first vector by an angle θ. The new coordinates are computed using the formula shown in Equation 2.46.

$$\begin{aligned} x &= x_0 cos(\theta) - y_0 sin(\theta) \\ y &= y_0 cos(\theta) + x_0 sin(\theta) \end{aligned} \tag{2.46}$$

Equation 2.46 can be rearranged into the following [31]:

$$x = cos(\theta)[x_0 - y_0 tan(\theta)]$$
$$y = cos(\theta)[y_0 + x_0 tan(\theta)] \tag{2.47}$$

We can simplify the equation above if the rotation angles are restricted so that $tan(\theta) = \pm 2^{-i}$ and thus, the multiplication has been reduced to a simple shift operation[31]. Any rotation angle can be obtained by performing a series of successively smaller elementary rotations. We can also simplify the *cos* term by deciding which direction to rotate rather than whether or not to rotate and therefore make the $cos(\theta)$ term a constant since $cos(\theta) = cos(-\theta)$ [31]. The iterative rotation can now be expressed as:

$$x_n = K_i[x_0 - y_0 d_i 2^{-i}]$$
$$y_n = K_i[y_0 + x_0 d_i 2^{-i}] \tag{2.48}$$

where:

$$k_i = cos(tan^{-1}2^{-1}) = \frac{1}{\sqrt{1 + 2^{-2i}}} \tag{2.49}$$
$$d_i = \pm 1 \tag{2.50}$$

Removing the scale constant from the iterative equations yields an algorithm consisting of only shift and add operations. The product of K_i can be applied at the end instead. K_i reaches 0.6073 as the number of iterations reaches infinity, and the rotation algorithm, therefore, has a gain, A_n, of approximately 1.647 [31]. The exact number is dependent on the number of iterations as shown in Equation 2.51:

$$A_n = \prod_{i=0}^{n-1} \sqrt{1 + 2^{-2i}} \tag{2.51}$$

For some calculations, the CORDIC algorithm also uses a third term which keeps track of the progress done by the CORDIC [31]. An angle accumulator tracks the progress:

$$Z_{i+1} = [Z_i - d_i tan^{-1}(2^{-i})] \tag{2.52}$$

The $tan^{-1}(2^{-i})$ term can be implemented in hardware by storing a finite number of entries in a ROM. Table 2.1 shows the 10 first values of $tan^{-1}(2^{-i})$.

Table 2.1: 10 first values of $tan^{-1}(2^{-i})$

i	0	1	2	3	4	5	6	7	8	9
2^{-i}	1	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256	1/512
$tan^{-1}(2^{-i})$	45	26.6	14.0	7.1	3.6	1.8	0.9	0.4	0.2	0.1

The CORDIC algorithm is normally operating in one of two modes. The first mode is called rotation mode, and it rotates an input vector by a specific angle. When operating in rotation mode, the angle accumulator is initialized with the desired rotation angle. For rotation mode the CORDIC equations are:

$$\begin{aligned} x_{i+1} &= [x_i - y_i d_i 2^{-i}] \\ y_{i+1} &= [y_i + x_i d_i 2^{-i}] \\ z_{i+1} &= [z_i - d_i tan^{-1}(2^{-i})] \end{aligned} \tag{2.53}$$

where

$$d_i = \begin{cases} -1 & \text{if } z_i < 0. \\ +1 & \text{otherwise.} \end{cases} \tag{2.54}$$

After a finite number of iterations, the final result will be:

$$\begin{aligned} x_n &= A_n[x_0 cos(z_0) - y_0 sin(z_0)] \\ x_n &= A_n[y_0 sin(z_0) + x_0 cos(z_0)] \\ z_n &= 0 \end{aligned} \tag{2.55}$$

Table 2.2 shows an example of the CORDIC algorithm working in rotation mode.

Table 2.2: CORDIC rotating (x_{in}, y_{in}) by 90 deg using 10 micro-rotations. Initial values are $x_{in} = 1$ and $y_{in} = 0$. The (x, y) values shown in the table are always scaled by multiplying with $k = 0.6072$.

i	$z(i)$	d	$x(i)$	$y(i)$
0	0.47694	1	0.60725	0.60725
1	0.19538	1	0.30363	0.91088
2	0.04662	1	0.075907	0.98679
3	-0.028895	1	-0.047442	0.99627
4	0.009009	-1	0.014826	0.99924
5	-0.0099615	1	-0.016401	0.9997
6	-0.0004739	-1	-0.00078037	0.99996
7	0.0042702	-1	0.0070318	0.99997
8	0.0018981	1	0.0031257	0.99999
9	0.00071206	1	0.0011726	1

The other mode is the vectoring mode, which rotates the input vector to the x-axis while recording the angle required to rotate. In this mode, the CORDIC equations are:

$$\begin{aligned} x_{i+1} &= [x_i - y_i d_i 2^{-i}] \\ y_{i+1} &= [y_i + x_i d_i 2^{-i}] \\ z_{i+1} &= [z_i - d_i arctan(2^{-i})] \end{aligned} \tag{2.56}$$

where

$$d_i = \begin{cases} -1 & \text{if } y_i < 0. \\ +1 & \text{otherwise.} \end{cases} \tag{2.57}$$

After a finite number of iterations, the final result will be:

$$\begin{aligned} x_n &= A_n \times \sqrt{x_0^2 + y_0^2} \\ y_n &= 0 \\ z_n &= z_0 + tan^{-1}\left(\frac{y_0}{x_0}\right) \end{aligned} \tag{2.58}$$

If we initialize the input z_0 to zero then $z_n = tan^{-1}(\frac{Y_0}{X_0})$. This means that we can use the vector mode in order to calculate the inverse tangent function. Table 2.3 shows an example of the CORDIC algorithm in vectoring mode.

Table 2.3: CORDIC vectoring mode with $x_{in} = 0.7071$, $y_{in} = 0.7071$ and $z_{in} = 0$. After a 10 micro-rotations the result of $atan2(x, y)$ can be seen in the z output

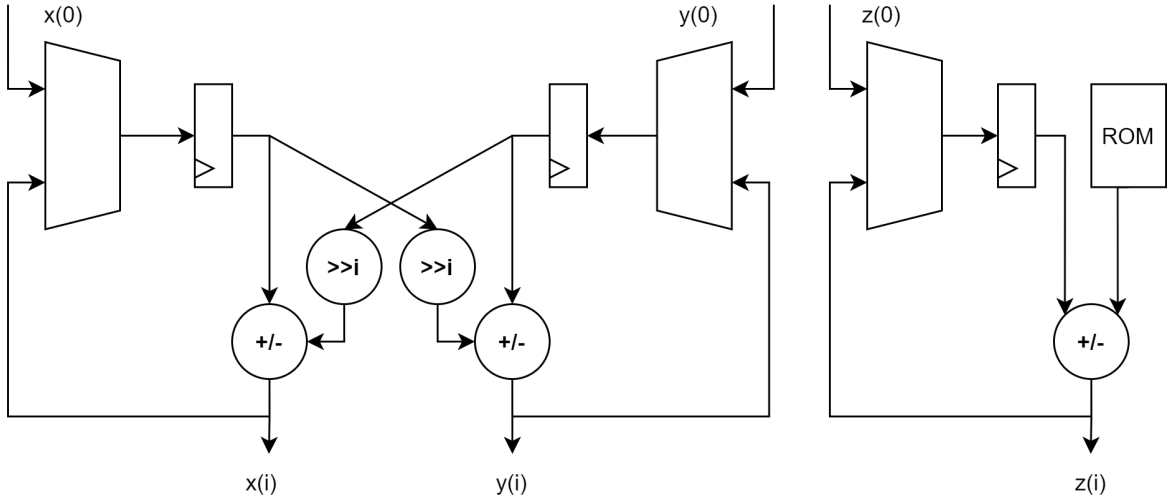
i	$z(i)$	d	$x(i)$	$y(i)$
0	0.7854	-1	1.4142	0
1	1.249	-1	1.4142	-0.70711
2	1.0041	1	1.591	-0.35355
3	0.87971	1	1.6352	-0.15468
4	0.81729	1	1.6449	-0.052481
5	0.78605	1	1.6465	-0.001079
6	0.77043	1	1.6465	0.024647
7	0.77824	-1	1.6467	0.011784
8	0.78215	-1	1.6467	0.0053517
9	0.7841	-1	1.6468	0.0021354

The current equation for the atan function is limited to $(-\frac{\pi}{2}, \frac{\pi}{2})$. During the Eigenvalue Decomposition (EVD), it might be necessary to calculate values outside this range, and the algorithm must be extended to support these cases. For the algorithm to support all angles, the input values must be conditioned with the following:

$$x, y, z = \begin{cases} -x, -y, z + \pi & \text{if } x < 0. \\ x, y, z & \text{otherwise.} \end{cases} \tag{2.59}$$

Figure 2.10 shows the hardware architecture for a bit-parallel iterative CORDIC. The module contains three bit-parallel registers to hold the current values of x, y, and z. In each iteration, the x and y values add or subtracted with the previous value shifted several times equal to the current iteration depending on the current sign. The z value is either subtracted or added with the current atan value corresponding to the current iteration. The atan values are typically stored in a ROM.

Figure 2.10: Iterative CORDIC architecture



2.6 Eigenvalue Analysis for Real Matrices

This section will introduce some current methods for solving the eigenvalue problem. The computation of eigenvalues for symmetric matrices is essential for many direction-finding techniques [14]. The number of antenna elements in a direction finding is highly related to the accuracy and the ability to track multiple concurrent signal sources [11], but when the number of elements increases, so does the size of the covariance matrix, and the computational power needed to calculate the eigenvalues and vectors increases drastically [32]. Some current methods for solving the eigenvalue problem are Gauss Siedel (GS) [33, p. 305], Successive-over-Relaxation (SOR) [34, p. 866], QR-decomposition [35, p. 98], and Jacobi methods [35, p. 463]. The Jacobi method is more favored for hardware acceleration even though it is significantly slower than the QR method [35, p. 463]. This is due to the fact that the Jacobi method is numerically stable for all real symmetrical matrices in addition to being simpler to implement [35, p. 463]. The Jacobi method can also avoid expensive operations such as square roots and divisions while delivering superior accuracy under certain conditions. The classical Jacobi method will be presented in Section 2.6.1, then the Cyclic Jacobi method is introduced 2.6.2, which is more suitable for computer systems. Finally, the parallel Jacobi method is presented in Section 2.6.3, which is currently one of the fastest methods for solving the eigenvalue problem [32].

2.6.1 The Classical Jacobi Method

Singular Value Decomposition (SVD) is an important method for factorizing a matrix in linear algebra [36]. The SVD of the original matrix $A \in R^{m \times n}$ is:

$$\text{SVD: } A = UDV^T \tag{2.60}$$

where $U \in R^{m \times m}$ and $V \in R^{n \times n}$ are orthogonal matrices and $D \in R^{m \times n}$ is diagonal. In SVD, the elements of D are the singular values of A, the columns of $U = u_1, ..., u_m$ are the left singular vectors of A, and the columns of $V = v_1, ..., v_m$ are the right singular vectors

of A [36]. The classic Jacobi method proposed by Carl G. J. Jacobi in 1846 approximates the Eigenvalue Decomposition utilizing a sequence of rotations or iterations for obtaining the diagonal D of an original matrix $A \in R^{m \times n}$:

$$A^{k+1} = J_{pq}^T A^k J_{pq}, \quad k = 0,1,2,.. \tag{2.61}$$

where $A^{k=0}$ is the original matrix and J_{pq} is the transformation matrix governing the Jacobi rotation as shown in Equation 2.62. It is defined by the parameters $c, s, -s, c$ in the pp, pq, qp, qq entries of an $n \times n$ identity matrix, where $p < q, c = cos(\theta)$, $s = sin(\theta)$, and θ is the rotation angle.

$$J(p,q,\theta) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \tag{2.62}$$

Equation 2.63 shows how the optimal rotation angle can be calculated.

$$\theta = \frac{1}{2} arctan2 \left[\frac{2a_{pq}^k}{a_{qq}^k - a_{pp}^k} \right] \tag{2.63}$$

Equation 2.64 shows how the optimal angle θ and the rotation matrix can be used to annihilate two entries of a 2×2 matrix.

$$\begin{bmatrix} cos(\theta) & sin(\theta) \\ -sin(\theta) & cos(\theta) \end{bmatrix}^T \begin{bmatrix} A_{pp}^k & A_{pq}^k \\ A_{qp}^k & A_{qq}^k \end{bmatrix} \begin{bmatrix} cos(\theta) & sin(\theta) \\ -sin(\theta) & cos(\theta) \end{bmatrix} = \begin{bmatrix} A_{pp}^{k+1} & 0 \\ 0 & A_{qq}^{k+1} \end{bmatrix} \tag{2.64}$$

The matrix values outside the diagonal are diminished through a series of iterations. It would take an infinite number of iterations to null the outer values completely, and it is therefore required to set a threshold value where the matrix is considered diagonalized. When an approximation is sufficient, it can be shown that the D is approximately diagonal after $O(Nlog(N))$ iterations [36].

2.6.2 The Cyclic Jacobi Method

The classic Jacobi method searches the upper triangle in each iteration and sets the largest off-diagonal element to zero. The search takes significant time and should be avoided if possible. The cyclic Jacobi method is a faster method that is more suitable for hardware implementation. The cyclic method traverses the upper triangle in a fixed order. For a 4×4 matrix, this would look like this:

$$(p,q) = (1,2) \rightarrow (1,3) \rightarrow (1,4) \rightarrow (2,3) \rightarrow (2,4) \rightarrow (3,4) \tag{2.65}$$

Each set shown in Equation 2.65 is referred to as a sweep, which takes $N(N-1)/2$ Jacobi rotations for each sweep. The diagonalization of the matrix will be finished after a few sweeps. Typically, three to six sweeps are enough.

2.6.3 The Parallel Jacobi Method

Additional improvements are possible by exploiting the fact that when performing Jacobi rotations around (p,q), only rows and columns p and q are altered. Instead of performing each rotation in a one-by-one cyclic order, they can be separated into multiple sub-problems and executed in parallel using a multiprocessor architecture. Ahmedsaid et al. [36] first presented a parallel array based on the Jacobi method. It involves dividing the original matrix into 2×2 sub-blocks. Each sub-block is calculated by a separate Processing Element (PE).

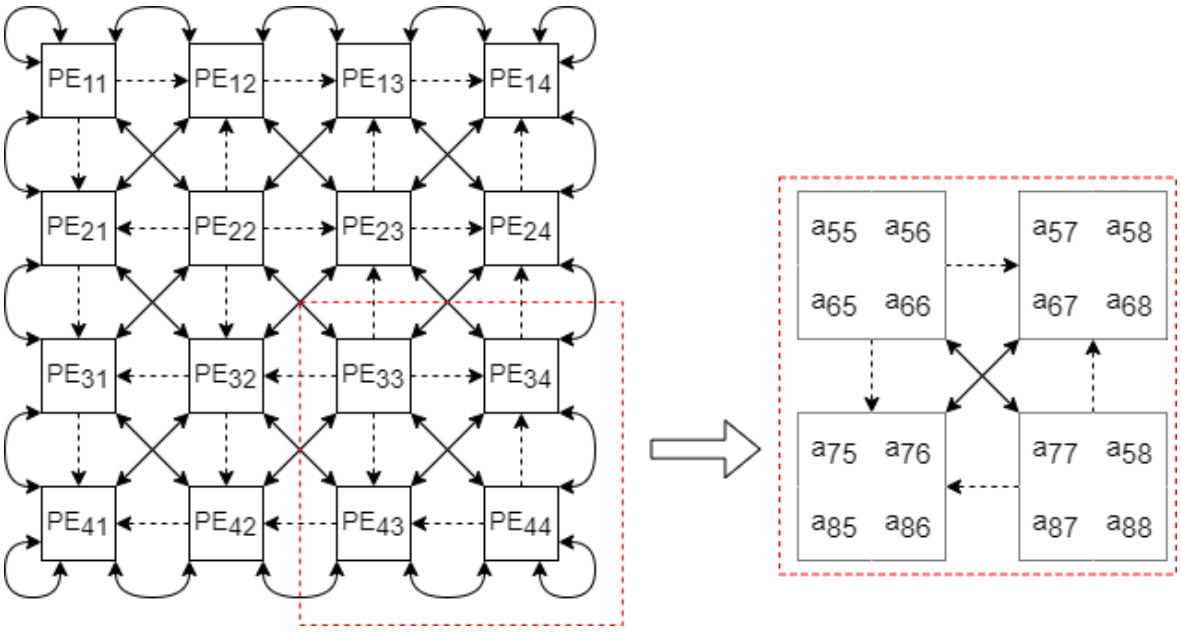


Figure 2.11: A 4×4 Brent-Luk-EVD array, where $n = 8$ for an 8×8 symmetric matrix

Figure 2.11 shows an example of 4×4 Brent-Luk-EVD array for an 8×8 symmetrical matrix. It is enough to calculate only the upper triangle of the matrix if the matrix is symmetrical along the diagonal. Exploiting this symmetry reduces the number of PEs from $N \times N$ to $N(N-1)/2$ resulting in approximately 50% reduction in area. The processing elements consist of two types, Diagonal Processing Element (DPE) and Off-diagonal Processing Element (OPE). At the start of the parallel Jacobi method, each sub-matrix is loaded into the corresponding PE. Then the diagonal PEs must compute the optimal rotation angle θ while the off-diagonal PEs wait. After the DPEs are finished, the angles are broadcast to the PE on the same row and column as indicated by the striped arrows. When every PE has received the optimal angle, each PE computes the left-hand side rotation followed by a right-hand rotation. After completing both rotations, the values are exchanged between adjacent processing elements, as the solid arrows indicate. After the exchange, the cycle is repeated several times before the matrix is fully diagonalized.

The following shows an example of finding the eigenvalue of a matrix using the parallel Jacobi method. The matrix in the example is 4×4 for simplicity, but the method can be extended to any size as long as the matrix is symmetrical and even-numbered in size. We define a symmetrical matrix along the diagonal because then it is enough only to solve the upper triangle, and we can ignore the lower triangle.

$$A^{k=0} = \begin{bmatrix} 4 & 4 & 2 & 2 \\ 4 & 4 & 2 & 2 \\ 2 & 2 & 1 & 1 \\ 2 & 2 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 4 & 4 & 2 & 2 \\ 4 & 4 & 2 & 2 \\ & & 1 & 1 \\ & & 1 & 1 \end{bmatrix} \tag{2.66}$$

We divide $A^{k=0}$ into 2×2 sub-matrices and define them as DPE_{11}, OPE_{12} and DPE_{22} respectively as shown in Equation 2.67

$$DPE_{11}^{k=0} = \begin{bmatrix} 4 & 4 \\ 4 & 4 \end{bmatrix}, OPE_{12}^{k=0} = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, DPE_{22}^{k=0} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \tag{2.67}$$

The optimal angle for each PE is found using Equation 2.63:

$$\theta_{11} = \frac{1}{2} arctan2\,(2 \times 4{,}4 - 4) = \frac{1}{2} arctan2\,(8{,}0) = 0.7854 \tag{2.68}$$

$$\theta_{22} = \frac{1}{2} arctan2\,(2 \times 1{,}1 - 1) = \frac{1}{2} arctan2\,(2{,}0) = 0.7854 \tag{2.69}$$

Substitute the angles θ_{11}, θ_{22} into the rotation matrices. Diagonal PEs use their own angle, while the off-diagonal PE rotates using the angle on the same row and column for the left-hand and right-hand side, respectively.

$$DPE_{11}^{k=1} = \begin{bmatrix} cos(\theta_{11}) & sin(\theta_{11}) \\ -sin(\theta_{11}) & cos(\theta_{11}) \end{bmatrix}^T \begin{bmatrix} 4 & 4 \\ 4 & 4 \end{bmatrix} \begin{bmatrix} cos(\theta_{11}) & sin(\theta_{11}) \\ -sin(\theta_{11}) & cos(\theta_{11}) \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 8 \end{bmatrix} \tag{2.70}$$

$$OPE_{12}^{k=1} = \begin{bmatrix} cos(\theta_{11}) & sin(\theta_{11}) \\ -sin(\theta_{11}) & cos(\theta_{11}) \end{bmatrix}^T \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} \begin{bmatrix} cos(\theta_{22}) & sin(\theta_{22}) \\ -sin(\theta_{22}) & cos(\theta_{22}) \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 4 \end{bmatrix} \tag{2.71}$$

$$DPE_{22}^{k=1} = \begin{bmatrix} cos(\theta_{22}) & sin(\theta_{22}) \\ -sin(\theta_{22}) & cos(\theta_{22}) \end{bmatrix}^T \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} cos(\theta_{22}) & sin(\theta_{22}) \\ -sin(\theta_{22}) & cos(\theta_{22}) \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 2 \end{bmatrix} \tag{2.72}$$

After rotating the matrix, each edge value is exchanged to an adjacent sub-matrix, as shown in Figure 2.11. After one iteration of the algorithm, the resulting matrix is shown in Equation 2.73. The new matrix has collected all the values in the lower triangle, but it is still not diagonal.

$$A^{k=1} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ & & 8 & 4 \\ & & 4 & 2 \end{bmatrix} \tag{2.73}$$

Calculate the new rotation angles:

$$\theta_{11} = \frac{1}{2} arctan2\,(2 \times 0{,}0 - 0) = \frac{1}{2} arctan2\,(0{,}0) = 0 \tag{2.74}$$

$$\theta_{22} = \frac{1}{2} arctan2\,(2 \times 4{,}8 - 2) = \frac{1}{2} arctan2\,(8{,}6) = 1.1071 \tag{2.75}$$

Substitute the angle into the rotation matrix a second time:

$$DPE_{11}^{k=2} = \begin{bmatrix} cos(\theta_{11}) & sin(\theta_{11}) \\ -sin(\theta_{11}) & cos(\theta_{11}) \end{bmatrix}^T \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} cos(\theta_{11}) & sin(\theta_{11}) \\ -sin(\theta_{11}) & cos(\theta_{11}) \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \tag{2.76}$$

$$OPE_{12}^{k=2} = \begin{bmatrix} cos(\theta_{11}) & sin(\theta_{11}) \\ -sin(\theta_{11}) & cos(\theta_{11}) \end{bmatrix}^T \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} cos(\theta_{22}) & sin(\theta_{22}) \\ -sin(\theta_{22}) & cos(\theta_{22}) \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \tag{2.77}$$

$$DPE_{22}^{k=2} = \begin{bmatrix} cos(\theta_{22}) & sin(\theta_{22}) \\ -sin(\theta_{22}) & cos(\theta_{22}) \end{bmatrix}^T \begin{bmatrix} 8 & 4 \\ 4 & 2 \end{bmatrix} \begin{bmatrix} cos(\theta_{22}) & sin(\theta_{22}) \\ -sin(\theta_{22}) & cos(\theta_{22}) \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 10 \end{bmatrix} \tag{2.78}$$

Exchange the edge values a second time and recreate the original matrix:

$$A^{k=2} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 \\ & & 0 & 0 \\ & & & 10 \end{bmatrix} \tag{2.79}$$

Equation 2.79 shows the resulting matrix after two iterations. The eigenvalues of A can be found along the diagonal in ascending order towards the lower right corner. In our simple example, A only contains one non-zero eigenvalue, but the method can find $M-1$ eigenvalues where M is the number of rows or columns in a symmetrical matrix. The eigenvectors can also be easily found using the same method but with slight modifications.

2.7 Vivado

Vivado is a tool developed by Xilinx [37], and it is used to synthesize and implement HDL on an FPGA. Synthesis involves transforming the HDL written in either VHDL or Verilog into a gate-level representation, while implementation involves placement, routing, and bitstream generation. After synthesis and implementation, the design has been transformed into a bitstream that can be loaded on the target FPGA.

Vivado also has tools for generating detailed reports about resource usage, timing, and power information. A utilization report gives insight into the number of look-up tables, registers, block ram, and digital signal processing DSP blocks utilized by the design. The timing report provides the designer with information about setup and hold violations, slack values, and critical paths in the design. Finally, Vivado can estimate the power usage of the design and report the static and dynamic power. The power usage is estimated using constraints such as the clock frequency provided by the designer.

2.8 Zynq Z2 System on a Chip

The Zynq Z2 is a part of the Zynq 7000 System on a Chip (SoC) family from Xilinx [38]. The Zynq contains the XC7Z020 SoC, which combines a dual-core ARM Cortex A9 processor with FPGA fabric on the same chip. The system is designed to provide high performance, low power consumption, and a large degree of flexibility for a wide range of applications. The Zynq has two main components: the Processing System (PS) and the Programmable Logic (PL).

The PS includes a hard Arm processor and various peripherals such as UART, USB, Ethernet, SPI, and other standard interfaces. It is designed to handle high-level software tasks such as running an operating system, communicating with external devices, and running application software. The PL includes programmable logic that can be configured to implement custom hardware functions. The PL has programmable logic such as look-up tables (LUT), flip-flops, block ram (BRAM), and digital signal processing (DSP) blocks. The PS and PL can communicate with each other using the AXI interface. The AXI bus can deliver high-speed and efficient data transfers between the two components. Table 2.4 shows an overview of the hardware resources available on the FPGA fabric.

Table 2.4: XC7Z020 Programmable Logic overview

Resource	Amount
Logic cells	85 000
Look-Up Tables	53 200
Flip-Flops	106 400
Block ram (number of 36 Kb Blocks)	4.9 Mb (140)
DSP slices (18x25 MACCs)	220

2.9 DSP48E1 DSP Slice

The Zynq Z2 FPGA from Xilinx [39] utilized in this thesis contains several dedicated Digital Signal Processing (DSP) slices throughout the chip. The DSP slice is optimized for performing high-speed and low-latency arithmetic operations. The DSP48E1 slice consists of dedicated arithmetic circuits that perform multiplication, addition, subtraction, and accumulation operations. Figure 2.12 shows how the DSP48E1 is structured. The inputs A and B are fed into a multiplier circuit that performs a 25x18 multiplication operation. Note that an optional pre-adder can add or subtract A and D before the multiplication. The multiplication results are fed into an adder circuit that can perform 48-bit addition or subtraction operations. The output of the adder is fed into the accumulator, which can perform 48-bit accumulation operations. Each of these operations can be completed in a single clock cycle.

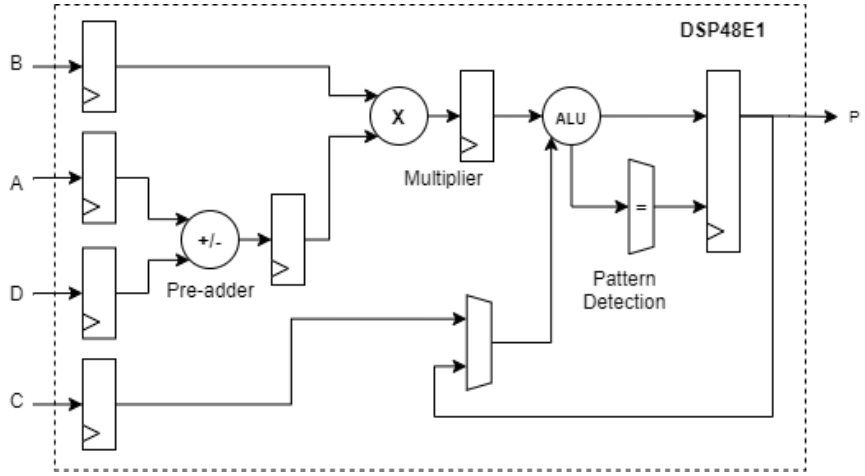


Figure 2.12: Basic Xilinx DSP48E1 Slice Functionality

The DSP48E1 slice also includes other features such as pipelining, saturation logic, rounding logic, and input/output registers. The pipeline registers allow the circuit to run at a very high clock frequency by breaking down arithmetic operations such as multiplication and addition into multiple stages. The traditional approach for implementing DSP algorithms involves using look-up tables (LUTs) to perform arithmetic operations. While LUT-based approaches are flexible and can be used for a wide range of DSP algorithms, they suffer from certain limitations regarding speed, area, and power consumption [39]. The DSP slice offers significant performance improvements over LUT-based designs. It uses less hardware and power and is often preferred over LUT-based designs, provided they are available [39].

Chapter 3

Hardware Implementation of the MUSIC Algorithm

Figure 3.1 illustrates the proposed hardware architecture. Recall from Section 2.4 that the real-valued MUSIC algorithm consists of the four following steps: Covariance Matrix Calculation (CMC), Real-valued Transformation (RVT), Eigenvalue Decomposition (EVD) and Spectral Peak Search (SPS). These steps correspond to the four components in Figure 3.1. The first component calculates the complex covariance matrix of the input samples. The second component performs a real-valued transformation of the complex covariance matrix into a real-valued covariance matrix. This method was chosen to save significant hardware resources and time during the Eigenvalue Decomposition (EVD) and the Spectral Peak Search (SPS). Working with real numbers results in approximately a 50% reduction in the amount of routing and resources such as look-up tables, registers, and block ram. This is due to the real values only containing the real component as opposed to the complex numbers, which also have a complex component. Significant time savings are also achieved since each iteration in the Jacobi method only requires two rotations when working with real matrices instead of four rotations when working with complex matrices [18]. After the matrix has been transformed, the eigenvalues and corresponding eigenvectors can be calculated, which are used in the spectral peak search to find the azimuth and elevation. This thesis will implement components one, two, and three, while another student will implement the final component [5].

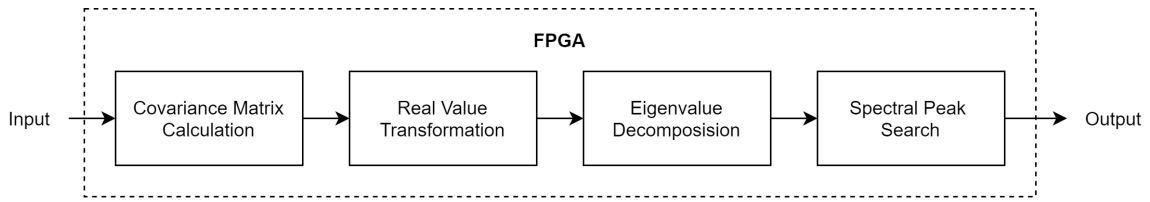


Figure 3.1: Proposed hardware architecture

The input IQ samples measured on the locator board or generated in software are represented as decimal values ranging from 1 to -1 since they are always placed within the unit circle. Instead of using decimal values, they can be represented by integers, ignoring the decimal point. This simplifies the hardware design, as all the logic can be implemented using a signed

value. This can either be done prior to transferring the data to the FPGA or by simply treating the data as integers and then truncating some of the digits to avoid overflow.

3.1 MUSIC Core

The MUSIC core contains the four components shown in Figure 3.1. The top-level entity is shown in Figure 3.2. The module has several parameters highlighted in green that can configure the operation of the core. The advantage of these parameters is that they allow flexibility to modify parts of the design by changing a few parameters instead of changing the design itself.

One example is the **width** parameter that decides how many bits the signed values within the core should use. This parameter can be easily changed to find the best trade-off between Performance, Power and Area (PPA). Typical bit width is between 16 to 20 bits depending on the desired accuracy and area requirements. Recall from Section 2.3 that the matrix size depends on the number of antenna elements and that the number of snapshots corresponds to how many times each antenna has been sampled. The **array_size** and **num_of_snapshots** select the number of samples and snapshots, respectively. The **iterations** parameter controls the number of iterations used by the CORDIC modules, and the **sweeps** parameter the number of Jacobi sweeps. The final parameter **num_vecmul** configures the number of parallel multipliers in the SPS module. The MUSIC core is initiated by setting the **enable** signal high for at least one clock cycle, and the **fetch** signal will be set high by the core each time a new snapshot is required. The samples are sent to the core as complex values. The **dout_valid** signal is set high when the final azimuth and elevation angle are ready at the output.

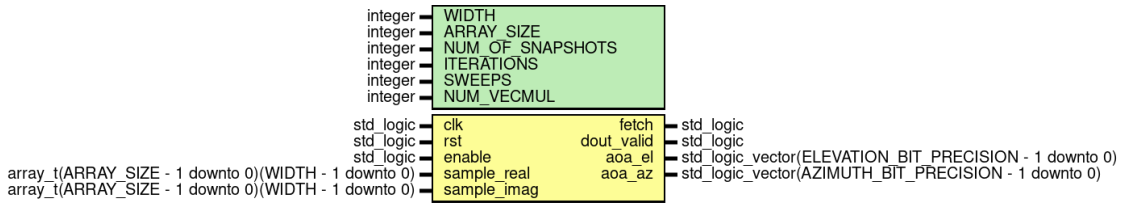


Figure 3.2: MUSIC Core top-level entity

3.2 Covariance Matrix Computation

The first step of the MUSIC algorithm is to calculate the covariance matrix as shown in Equation 3.1. The covariance matrix is a symmetrical matrix where each row and column has the same size as the number of samples in each snapshot. The formula can be simplified by removing the division as shown in Equation 3.1, which is desirable as division is time-consuming and takes considerable hardware resources. This simplification is possible since the averaging operation has no effect on subsequent calculation [40], and it is valid as long as we can ensure that no overflow is possible. If overflow is an issue, a hardware-efficient solution is to scale the numbers by a factor that is a power of two, as such an operation can be implemented using a shift operation.

$$R_{XX} = \frac{1}{T}\sum XX^H \Rightarrow \sum XX^H \tag{3.1}$$

Equation 3.2 shows how the covariance matrix is calculated. Each entry in the covariance matrix is calculated by one multiplication, and these multiplications together calculate $X \times X^H$. The summation $\sum$ is calculated by accumulating each entry in the covariance matrix between each snapshot. Combining these two operations will give the same result as Equation 3.1. The covariance matrix can be realized in hardware using a number of Complex Number Multiply and Accumulate (CMAC) units.

$$XX^H = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \cdot \end{bmatrix} \cdot \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix} = \begin{bmatrix} x_1^2 & x_1x_2 & \cdots & x_1x_n \\ x_1x_2 & x_2^2 & \cdots & x_2x_n \\ \vdots & \vdots & \ddots & \vdots \\ x_nx_1 & x_nx_2 & \cdots & x_n^2 \end{bmatrix} \tag{3.2}$$

An important design choice for the CMC is how parallel the design should be. Table 3.1 shows three different possible implementations. The table considers two key metrics for choosing the optimal solution, time complexity and hardware usage.

Table 3.1: Time and resource comparison between two CMC implementations

	Fully parallel (A)	Partially Parallel (B)	Iterative (C)
Time complexity	$O(M)$	$O(M \times N)$	$O(M \times N^2)$
CMAC usage	$N \times N$	N	1

Note: $N \times N$: matrix size, and M: number of snapshots

Implementation (A) is the fastest method because it multiplies and accumulates each element in the covariance matrix every cycle, and it is, therefore, $O(M)$, but this also requires one CMAC for each element resulting in $N \times N$ CMAC units. The iterative implementation multiplies and accumulate one entry each cycle, and it is, therefore, $O(M \times N^2)$, but it only requires CMAC. Implementation B is a compromise between A and B, and it operates by calculating one row each cycle resulting in a time complexity of $O(M \times N)$, and it uses N CMAC units. The following shows the estimated clock cycles required assuming that $N = 12$ and $M = 4$:

$$\text{Delay(A)} : 4 \tag{3.3}$$
$$\text{Delay(B)} : 4 \times 12 = 48 \tag{3.4}$$
$$\text{Delay(C)} : 4 \times 12^2 = 576 \tag{3.5}$$

If we assume that the design can operate at 100 MHz, then the calculation time would be:

$$\text{Time(A)} = 4 \times 0.01\mu s = 0.04\mu s \tag{3.6}$$
$$\text{Time(B)} = 48 \times 0.01\mu s = 0.48\mu s \tag{3.7}$$
$$\text{Time(C)} = 576 \times 0.01\mu s = 5.76\mu s \tag{3.8}$$

From Equation 3.6, we can see that implementation A is one order of magnitude faster than B and two orders of magnitude faster than C. The serial implementation is typically used in software and is not very suited for a hardware solution because of the high time complexity. The fully parallel implementation can achieve maximum speed and performance, but the drawback is that it is expensive and has a high data throughput. The high data throughput might result in a bottleneck in the system, and speed gains using this implementation might not be fully realized. One bottleneck is retrieving the input samples from the block ram, which can only read one value simultaneously for single-port and two for dual-port. The design would therefore require at least N block ram to fully utilize the parallel implementation since it has to retrieve one vector of size N each clock cycle. Assuming that the parallel version is configured with $N = 12$, the number of required DSP slices is:

$$\text{Area(A)} = 12 \times 12 \times 4 = 576 \tag{3.9}$$

$$\text{Area(B)} = 12 \times 4 = 48 \tag{3.10}$$

$$\text{Area(C)} = 1 \times 4 = 4 \tag{3.11}$$

Implementation (A) uses 576 DSP slices which is a relatively high number, and it would most likely require a high-end FPGA. This makes fully parallel implementation undesirable as the FPGA must either be large and therefore costly, or some of the CMAC units must be implemented using LUTs and registers, decreasing the system's performance and increasing power consumption. The partially parallel design is a compromise between the two others, and it trades some speed for vastly lower hardware usage. The partially parallel design calculates the covariance matrix by multiplying X^H with each element in X in a row-by-row order. This method significantly reduces the amount of data and calculations required because it does not need to multiply and accumulate every entry in the matrix each clock cycle. Another benefit of the partially parallel design is that it takes 12 clock cycles to multiply one matrix, making it possible to read one complex sample from a dual-port block ram each cycle or two single-port block ram.

The following sections will detail the design of the partially parallel CMC module. It will follow a bottom-up approach where each building block will be explained before being glued together in the finished module. First, the Complex Multiplier (CMUL) is introduced as a building block in the Complex Multiply and Accumulate module (CMAC). Next, the buffer and the conjugate transpose modules are detailed. Finally, the blocks are put together to form the CMC module.

3.2.1 Complex Multiplication Module

A series of complex multiplications are necessary to compute the covariance matrix since each sample contains complex values. Complex numbers present an issue since they typically require four multiplications for each complex multiplication, which will result in high usage of DSP slices inside the FPGA. Equation 3.12 shows that one complex multiplication requires four normal multiplications and three adder operations.

$$(A + Bi)(C + Di) = (AC - BD) + (AD + BC)i \tag{3.12}$$

Equation 3.16 shows an optimized method using the Karatsuba algorithm [41]. The optimized method only requires three normal multiplications and five adder operations. If we use our example of $N = 12$ this would decrease the number of required DSP slices from 48 down to 36, which is a 33% improvement.

$$K1 = C(A + B) \tag{3.13}$$
$$K2 = A(D - C) \tag{3.14}$$
$$K3 = B(C + D) \tag{3.15}$$
$$(A + Bi)(C + Di) = (K1 - K3) + (K1 + K2)i \tag{3.16}$$

We can conclude that the Karatsuba algorithm is better suited for hardware implementations assuming that multiplications are more expensive than additions or subtractions. The DSP48E1 slice also contains adders in the front and the back of the multiplier, which can be used to implement the additions and subtractions needed for the Karatsuba algorithm. Figure 3.3 shows the top-level entity of the Complex Multiplication (CMUL) module. The module accepts new data when **din_valid** is held high, and the **dout_valid** signal indicates when the data at the output is valid.

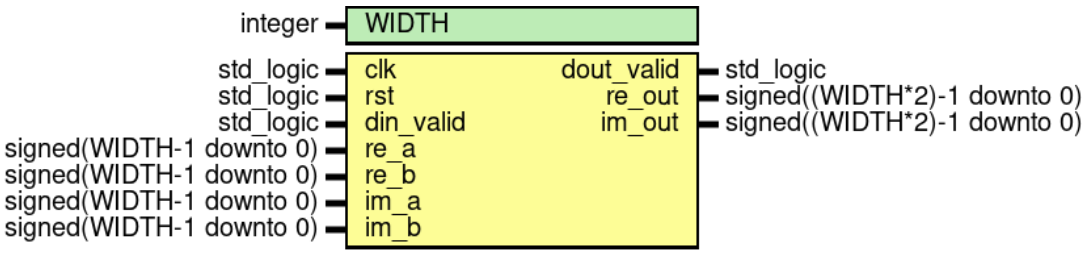


Figure 3.3: CMUL entity

Figure 3.4 shows the micro-architecture for the CMUL module. The design is fully pipelined to achieve high throughput and to decrease the critical path through the multiplier. The unit is designed to fully utilize the pre- and post-adders and the built-in register in the DSP slice and therefore optimize the hardware usage. The **dout_valid** signal is delayed by four registers to be synchronous with the data.

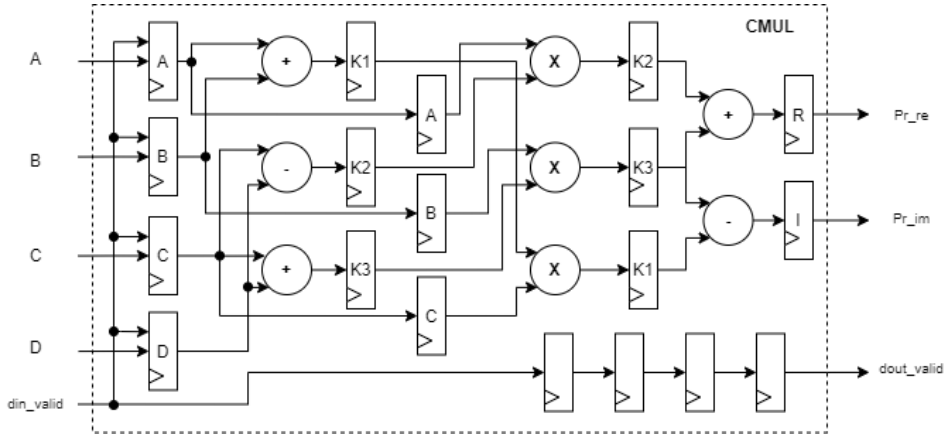


Figure 3.4: Architecture of the CMUL module

Table 3.2 shows the synthesis report generated by Vivavdo. We can see that the design can fully utilize the three DSP slices required without using any LUTs or registers outside the four connections between the **din_valid** signal and **dout_valid** signal. The multiplier is able to meet timing constraints with a frequency of up to 285 MHz.

Table 3.2: Synthesis report for the CMUL module

Parameter	Value	Utilization (%)
Look-up tables	0	0
Registers	4	≤ 1
DSP slices	3	1.36
Max clock frequency [MHz]	285	

Note: $W_L = 16$

3.2.2 Complex Multiply and Accumulate Module

The covariance matrix calculation can be divided into many parallel multiply and accumulate operations, and it is, therefore, necessary to design a unit capable of doing complex multiply and accumulate operations. A single operation is shown in Equation 3.17. The accumulation operation can be implemented by adding the previous value to the new value. Complex addition is done by adding the real and complex components of each number separately, as shown in Equation 3.18.

$$C = (A \times B) + C \tag{3.17}$$

$$(A + Bi) + (C + Di) = (A + C) + (B + D)i \tag{3.18}$$

Figure 3.5 shows the top-level entity of the CMAC module. The **din_valid** and **dout_valid** signals are used to accept new data and to indicate that the output data is valid. The output data and feedback data C are double the width of A and B to account for the growth of bits due to the multiplication operation.

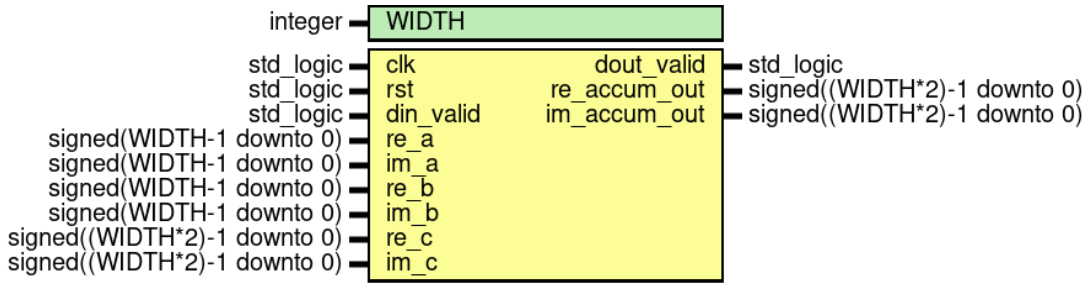


Figure 3.5: CMAC entity

Figure 3.6 shows the architecture of the CMAC. It utilizes the CMUL unit designed in Section 3.2.1. Four bit-parallel registers are inserted in series to synchronize the data between the three data inputs to the CMAC since the CMUL has four pipeline stages. The data from the CMUL unit and the pipelined data from C are fed into an adder and a register that

accumulates the data. A **dout_valid** signal from the CMUL unit is delayed by one cycle using a register to indicate when the data is ready on the output.

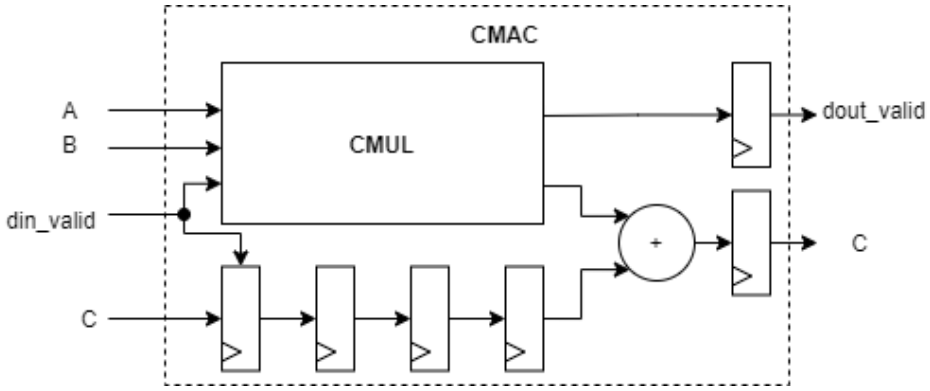


Figure 3.6: Architecture for the Complex Multiply and Accumulate Unit (CMAC).

Table 3.3 shows the synthesis report of the CMAC module. The add operation is implemented using LUTs, resulting in the unit using no additional DSP slices. Note that the maximum clock frequency has been reduced to 250 MHz. This is most likely because the addition is implemented using LUTs instead of DSP slices.

Table 3.3: Synthesis report for the CMAC module

Parameter	Value	Utilization (%)
Look-up tables	104	≤ 1
Registers	142	≤ 1
DSP slices	3	1.36
Max clock frequency [MHz]	250	

Note: $W_L = 16$

3.2.3 Conjugate Transpose Module

The conjugate transpose module takes one array of signed values at a time and negates the value of the complex component of each value. The complex conjugate transpose operation can be realized by implementing the operations shown in Equation 3.19 for each value in the array.

$$A + Bi = A - Bi \tag{3.19}$$

The top-level entity is shown in Figure 3.7, and the number of rows or columns can be configured by setting the **ARRAY_SIZE** parameter.

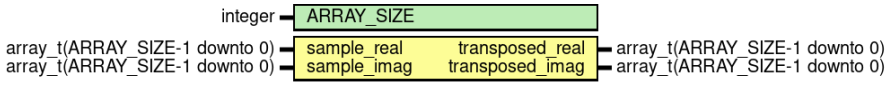


Figure 3.7: Conjugate Transpose entity

Table 3.4 shows the synthesis report of the conjugate transpose module. The module uses 132 look-up tables and can meet timing constraints of up to 300 MHz.

Table 3.4: Synthesis report for the conjugate transpose module

Parameter	Value	Utilization (%)
Look-up tables	132	≤ 1
Registers	0	0
DSP slices	0	0
Max clock frequency [MHz]	300	

Note: $W_L = 16$ and $N = 12$

3.2.4 Complex Shift Register Module

The Complex Shift Register (CSREG) stores intermediate results between each snapshot. Each CMAC is connected to its own CSREG and must be able to store $N - 5$ values since the remaining values are stored inside the pipeline of the CMAC and the accumulator. The CSREG might be realized in hardware as a bit-parallel shift register or block ram. The latter would be the most suitable when N gets very large as the number of registers needed for the shift register would grow. For small sizes of N, the controller controlling the block ram would most likely make the design more complex and take more space. Therefore, the bit-parallel shift register is chosen as it is simple to implement and control. Figure 3.8 shows the top-level entity of the complex shift register (CSREG). The **shift_en** is used to send **din** from one register to the next, while **clear** can be used to reset the values in all registers.

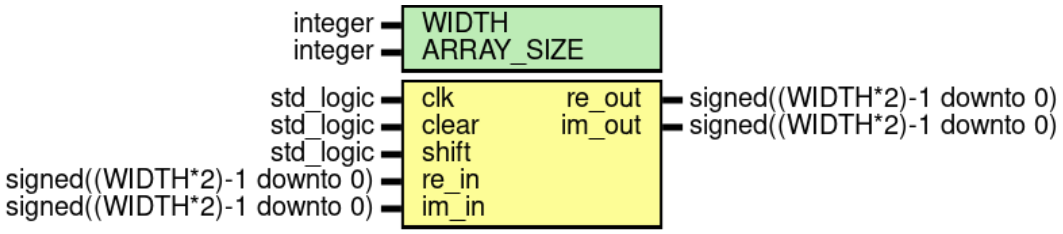


Figure 3.8: Complex Shift Register entity

Figure 3.9 shows how the CSREG might be implemented in hardware.

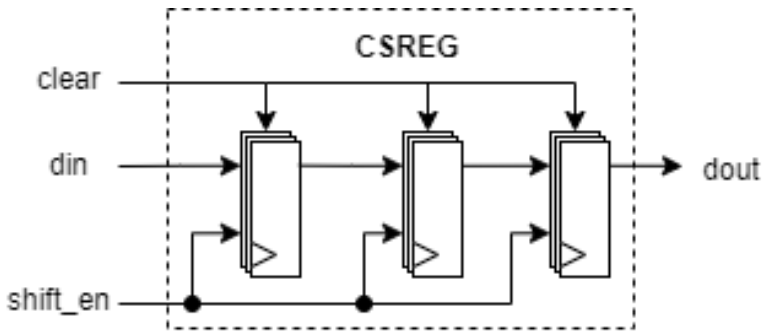


Figure 3.9: Architecture of the Complex Shift-Register (CSREG)

Table 3.5 shows the synthesis report of the conjugate transpose module. The module uses 224 registers and can meet timing constraints of up to 300 MHz.

Table 3.5: Synthesis report for the CSREG module

Parameter	Usage	Utilization (%)
Look-up tables	0	0
Registers	224	≤ 1
DSP slices	0	0
Max clock frequency [MHz]	300	

Note: $W_L = 16$ and $N = 12$

3.2.5 Covariance Matrix Calculation Module

Figure 3.10 shows the top-level entity of the CMC module. The **width** field configures the number of bits used to represent each integer value while **array_size** and **num_of_snapshots** configure the size of the input vector and the number of snapshots, respectively. The module is initiated by setting the **enable** signal high for one clock cycle, and the result is valid when **dout_valid** is held high. The **fetch** signal indicates when the module is ready to accept a new snapshot at the input.

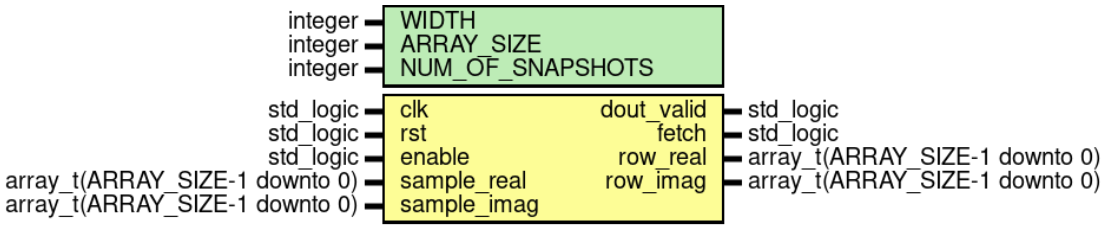


Figure 3.10: CMC entity

Figure 3.11 shows the architecture of the CMC module and how the module is partitioned. The CMC module is divided into a data path and a controller partition. The data path handles the data moving through the component using combinational logic and registers, while the controller includes a finite state machine (FSM) used to control the data flow in the data path and external control signals. The controller and datapath are connected together will control signals as shown in Figure 3.11. The data path consists of four components. The conjugate transpose computes $(\cdot)^H$ while the row multiplexer selects one value from the input data depending on the **row_sel** signal from the CMC controller. In each clock cycle of the same snapshot, every value from the conjugate transpose module is fed into its own CMAC together with the current value from row Mux. The results of the CMAC modules are fed into the CSREG, which stores intermediate values used in the next snapshot.

Figure 3.12 shows a simplified state machine diagram. The diagram shows how transitions between states are controlled. After a reset, the state machine will be in the **st_idle** state. In this state, the CMAC will be disabled while waiting for the **enable** signal to be asserted. When **enable** is set high, the state machine will transition to the **st_fill_pipe** state, which accounts for the delay while waiting for the pipeline to fill up. The **cmac_valid** signal is set high in this state to enable the CMAC unit and the row counter to start counting. When

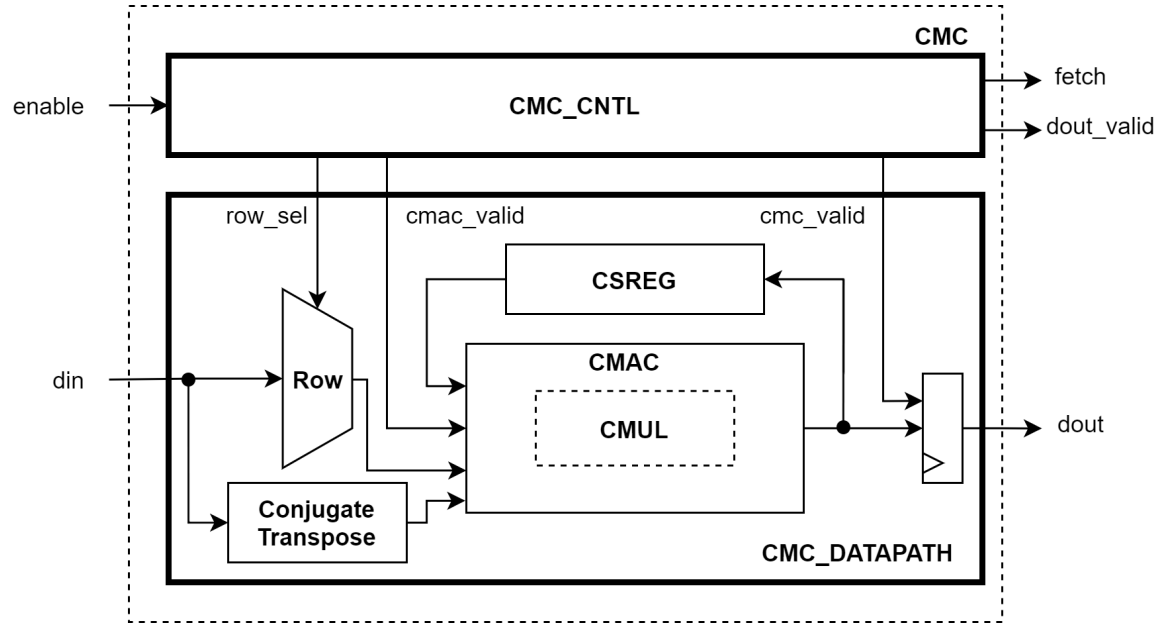


Figure 3.11: Architecture for the CMC module

the pipeline is filled, the machine will reach its final state, where it will stay as long as the snapshot counter and row counter are lower than the preset values before the machine goes back to idle. The row counter will count up to the value of **ARRAY_SIZE** before it returns to zero and starts counting again. This cycle will continue if the snapshot count is lower than **NUM_SNAPSHOTS**.

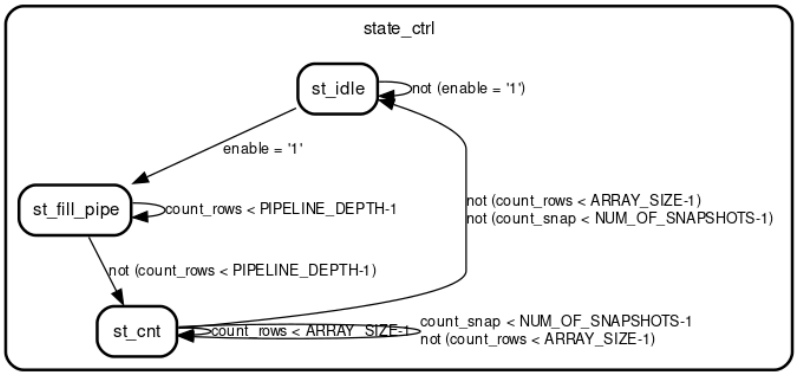


Figure 3.12: State machine of CMC controller

A separate state machine shown in Figure 3.13 keeps track of when the output data is valid. The state machine waits for the snapshot counter to reach the maximum value, and then it will transition to the **st_wait** state to account for the pipeline delay. After four clock cycles, the machine transitions to the **st_ready** state, and the **cmc_valid** and **dout_valid** signal is set high to indicate that the output is valid.

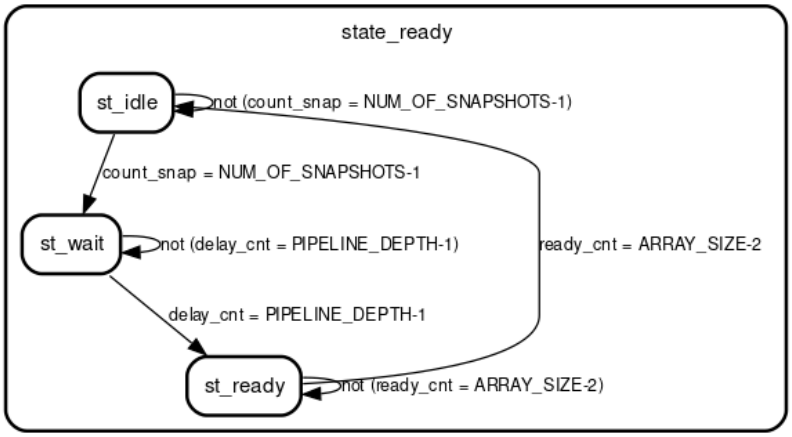


Figure 3.13: State machine of CMC ready controller

3.2.6 CMC Synthesis Report

Figure 3.6 shows the synthesis report of the CMC module. The module uses 6.4 % of the available look-up tables, 2.7 % of the registers, 16.4 % of the DSP slices, and no block ram. The module can meet all timing constraints up to 160 MHz.

Table 3.6: Synthesis report for the CMC module

Parameter	Usage	Utilization (%)
Look-up tables	3424	6.44
Registers	2816	2.65
Block ram	0	0
DSP slices	36	16.36
Max clock frequency [MHz]	160	

Note: $W_L = 16$, $N = 12$ and $M = 4$

3.3 Real-Value Transformation

A real-value transformation technique has been utilized to reduce the complexity and decrease the computational time of solving the eigenvalue problem and the Spectral Peak Search (SPS). As mentioned in the introduction to Chapter 3, solving the eigenvalue problem for complex matrices requires four rotations instead of two rotations for real matrices [18]. Table 3.7 shows a computational time and resource usage comparison for the Eigenvalue Decomposition (EVD) between using a real-value transformation and complex values [18]. The computation time depends on the number of rotations, CORDIC iterations, and Jacobi sweeps. The hardware usage is estimated based on the number of required CORDIC cores since the EVD module consists primarily of said cores. The estimated cores for the complex implementation are found in [18].

Table 3.7: Time and resource comparison

	Real-value Transformation (A)	Complex value (B)
Time complexity	$3C_I \times J_S$	$5C_I \times J_S$
Hardware usage	$2C_U \times N + 2C_U \times \frac{M}{(M-1)}$	$3C_U \times N + 4C_U \times \frac{M}{(M-1)}$

Note: C_I: CORDIC iterations, J_S: Jacobi sweeps, C_U: CORDIC units and $M = \frac{N}{2}$, $N \times N$: matrix size

The number of required clock cycles for solving the eigenvalue problem using the parallel Jacobi method for both real and complex matrices assuming $C_I = 16$ and $J_S = 6$ are:

$$\text{Delay(A)} : 3 \times 16 \times 6 = 288 \tag{3.20}$$

$$\text{Delay(B)} : 5 \times 16 \times 6 = 480 \tag{3.21}$$

Assuming a clock frequency of 100 MHz this would result in the following computational time:

$$\text{Time(A)} = 288 \times 0.01\mu s = 2.8\mu s \tag{3.22}$$

$$\text{Time(B)} = 480 \times 0.01\mu s = 4.8\mu s \tag{3.23}$$

The hardware used by the parallel Jacobi method of the real- and complex-valued EVD would be:

$$\text{Area(A)} = 2 \times 12 + 2 \times 6(6-1) = 84C_U \tag{3.24}$$

$$\text{Area(B)} = 3 \times 12 + 4 \times 6(6-1) = 156C_U \tag{3.25}$$

The RVT method reduces the resources used by the EVD by 46% and the computational time by 48%. The RVT method reduces the required routing and storage since the values only contain a real component. The calculations above are only an estimate, and they do not take the overhead added by the real-value transformation into account, but the required calculations should still have significant improvements over the original method using complex values.

3.3.1 Forward Backward Averaging

Recall from Section 2.4 that a prerequisite for the unitary transformation is that the covariance matrix is both hermitian and persymmetric. The matrix generated by the Covariance Matrix Calculation (CMC) module is only hermitian, and it is therefore required to apply forward-backward averaging for the matrix to become persymmetric. This can be done by applying Equation 3.26 as mentioned in Section 2.4.

$$R_{FB} = 0.5(R + JR^T J) \tag{3.26}$$

A potential issue with Equation 3.26 is related to when the data from the R matrix is available to the module. The data coming from the previous CMC module is sent row-by-row, which results in a problem when adding R and R^T since the transposed cannot be calculated without having access to the entire matrix. To resolve this data availability issue two methods were investigated. The first method involves calculating $JR^T J$ and storing it in a buffer. The buffer's content can then be transposed and added to R, yielding R_{FB}, but this would require additional hardware resources. The second method involves simplifying Equation 3.26 into 3.27.

$$R_{FB} = JR^T J \tag{3.27}$$

This simplification is possible because the covariance matrix R is Hermitian and therefore, the transpose of R is equal to the complex conjugate $\bar{R}$. Using these identities, we can outline the following:

$$\begin{aligned} R_{FB} &= 0.5(R + JR^T J) \\ &= 0.5(R + J\bar{R}J) \\ &= 0.5R + 0.5J\bar{R}J) \\ &= 0.5R + \frac{J\bar{R}J}{2} \end{aligned} \tag{3.28}$$

Now, we know that $J\bar{R}J$ is equal to its conjugate transpose since it involves only matrices and their complex conjugates. We can rewrite the equation above into:

$$\begin{aligned} R_{FB} &= 0.5R + 0.5\overline{(JR^T J)} \\ &= 0.5R + \frac{JR^T J}{2} \end{aligned} \tag{3.29}$$

From Equation 3.29, we can see that the expression is equal to $JR^T J/2 + 0.5R$, which is the same as $JR^T J$ since $0.5R$ is a Hermitian matrix and thus commutes with $JR^T J$. From this, we can conclude that Equation 3.26 is equal to 3.27 as long as R is hermitian.

3.3.2 Unitary Transform

The unitary transform is the second step of the real-valued transformation. The unitary transform is calculated using Equation 3.30 from Section 2.4.

$$R = URU^H \tag{3.30}$$

Note that U is highly dependent on the physical layout of the locator board. The general form shown in Section 2.4 will not directly work on the specific antenna structure in this thesis, and it, therefore, must be modified. The general formula given by Wei Zhang et al. [19] assumes that the physical locator board has a 4×4 antenna structure which results in 16 total antenna elements, but the antenna in this thesis is missing the four antennas in the middle. Recall that the Unitary Transform requires two Kronecker products. For $M = 4$, we have the following:

$$T_x = T_y = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & i & -i & 0 \\ i & 1 & 1 & -i \end{bmatrix} \tag{3.31}$$

If we take $T_x \otimes T_y$ we get:

$$U = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & j & -i & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & j & -i & 0 \\ j & 0 & 0 & -i & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & j & 0 & 0 & -i \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & j & -i & 0 & 0 & j & -i & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & j & 0 & 0 & -i & j & 0 & 0 & -i & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & j & 0 & 0 & j & -i & 0 & 0 & -i & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & j & j & 0 & 0 & -i & -i & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ j & 0 & 0 & j & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -i & 0 & 0 & -i \\ 0 & j & j & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -i & -i & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 \end{bmatrix} \tag{3.32}$$

Note that Equation 3.32 is only valid when the matrix is 16×16, but the matrix used in this thesis is only 12×12. We, therefore, need to modify the matrix by removing the missing rows and columns. In our case, we must remove rows and columns 6,7,10, and 11. This would result in the following matrix:

$$U = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & j & -i & 0 & 0 & 0 & 0 & 0 & 0 & j & -i & 0 \\ j & 0 & 0 & -i & 0 & 0 & 0 & 0 & j & 0 & 0 & -i \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & j & -i & j & -i & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & j & j & -i & -i & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 1 & -1 & 0 & 0 & 0 & 0 \\ j & 0 & 0 & j & 0 & 0 & 0 & 0 & -i & 0 & 0 & -i \\ 0 & j & j & 0 & 0 & 0 & 0 & 0 & 0 & -i & -i & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -i \end{bmatrix} \tag{3.33}$$

The matrix in Equation 3.33 will allow us to convert our covariance matrix into a real-valued matrix without losing the information stored in the complex values.

If we look at Equation 3.33 there are some obvious optimizations to be made. The first observation is that many of the entries are zero, meaning we can ignore these values when pre- and post-multiplying R with U and U^H. The second observation is that all the non-zero entries are unity which implies that there is no need to multiply our matrix with the covariance matrix since these operations can be implemented using addition. The final observation is that the values in Equation 3.33 are constant, which can be exploited when designing the hardware. The design for this module could be designed by implementing a hardware module for matrix multiplications, but with the observations above, this would most likely waste resources and time.

3.3.3 Real-value Transform Module

Figure 3.14 shows the top-level entity for the Real-Valued Transformation (RVT) module. The module will read one row of complex values each clock cycle that **din_valid** is held high. This synergizes with the previous CMC module since it outputs one row at a time. At the output, one column only containing real values is sent out when **dout_valid** is high.

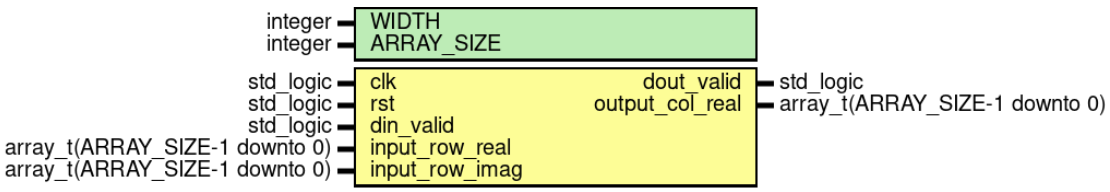


Figure 3.14: RVT top-level entity

Figure 3.15 shows how the RVT module is structured. The forward-backward averaging operation can be implemented in hardware by calculating the conjugate transpose using the same module used in the CMC, which will compute R^T. Note that it is unnecessary to pre- and post-multiply by the exchange matrix J since J has the same dimension as R^T and therefore has no effect on the result. After the conjugate transpose operation, the R^T transforms from row-by-row to column-by-column ordering. The conjugate transposed data is then moved into the unitary transformation (UT) module, which has been partitioned into a controller and datapath module. The controller contains a state machine that controls the

control signals connected to the datapath while the datapath handles the flow and operation of the data itself.

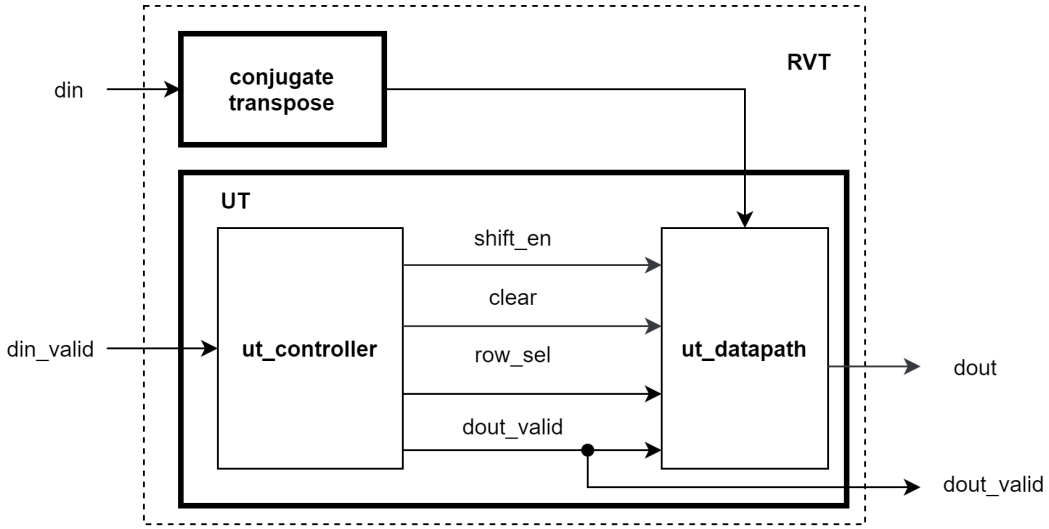


Figure 3.15: RVT architecture

3.3.4 Unitary Transform Datapath

The Unitary Transform (UT) datapath is responsible for calculating the following equation:

$$\hat{R} = \frac{1}{4} U R U^H \tag{3.34}$$

where R is the complex matrix, U is the unitary matrix and $\hat{R}$ is the real-valued matrix. Equation 3.34 can be split into two separate equations for the left- and right-hand side multiplication:

$$\tilde{R} = \frac{1}{2} U R \tag{3.35}$$

$$\hat{R} = \frac{1}{2} \tilde{R} U^H \tag{3.36}$$

The left-hand side multiplication can be implemented by multiplying each row of U with each column of R while the right-hand multiplication is calculated by multiplying each row of $\tilde{R}$ from the left-hand with each column of U^H. It is worth noting that the left-hand side operation will output one column at a time while the right-hand side requires one row at a time which makes it impossible to directly calculate the second operation without storing the intermediate results. Calculating the unitary transform using the original equation would also result in a different architecture for the left and right side multiplication since $U \neq U^H$. An alternative method can be used by rewriting 3.34 into Equation 3.37:

$$\tilde{R} = \frac{1}{2} U R \tag{3.37}$$

$$\hat{R} = \frac{1}{2} U \tilde{R}^H \tag{3.38}$$

Equation 3.37 still requires that the intermediate result is stored, but it greatly simplifies the design since the same operation of multiplying U with an arbitrary matrix R with the same shape. The only difference is that in the second multiplication, R has been transposed. Figure 3.16 shows the datapath inside the Unitary Transformation module. The module is designed to work around the Unitary Multiplication (UMUL) module, which is the module that calculates the multiplication operation shown in Equation 3.37

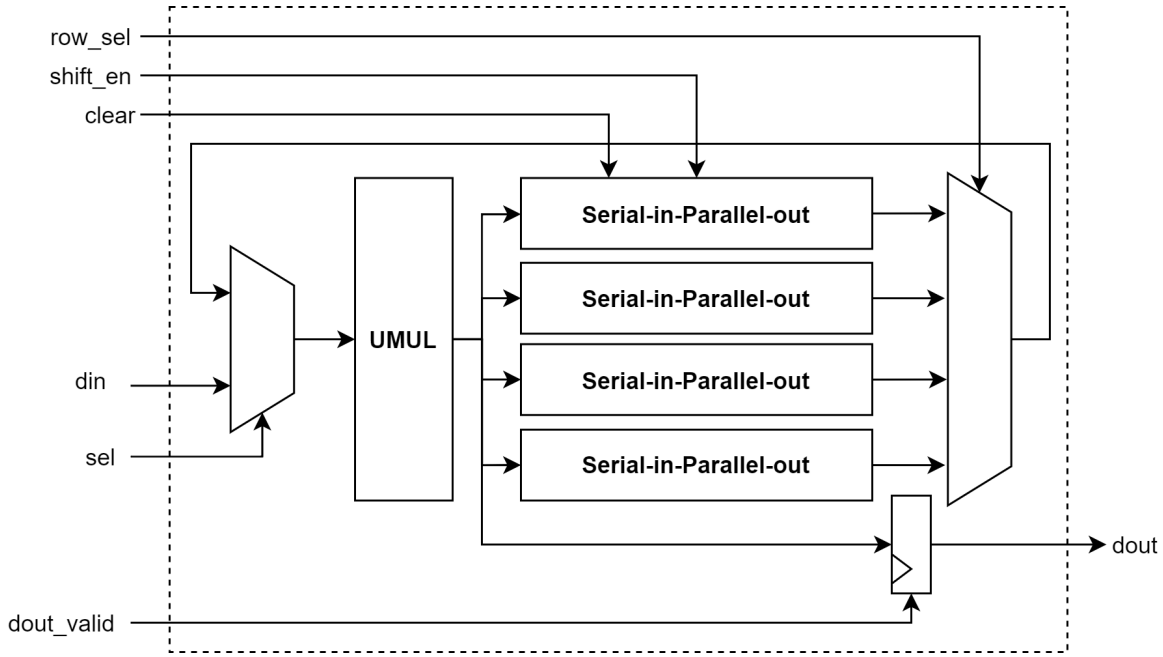


Figure 3.16: Unitary Transform architecture

The module is centered around the UMUL module, which calculates the multiplication of the unitary transform with either R or R^T. A multiplexer controls which data is sent to the UMUL module. During the first multiplication, the multiplexer selects **din** while **sel** is held low. The output of the UMUL is shifted into N number of serial-in-parallel-out shift registers. The entire matrix $U \times R$ is stored inside the shift registers. When all values are written, the output multiplexer reads out every value of one shift register at a time. The state machine controls the multiplexer. The value selected by the multiplexer is then sent back to the input multiplexer by holding **sel** high and into the UMUL module a second time. The shift registers will transpose the result of the first and multiply it a second time, and the design can utilize the UMUL for both operations. The output value is sent out one column at a time when **dout_valid** is held high.

3.3.5 Unitary Multiplier module

Unitary Multiplier (UMUL) module calculates $U \times A$ where A is an arbitrary 12×12 matrix. The module accepts one column at a time from the matrix A each cycle, and it will multiply

each column with every row of U. The module outputs one column every cycle. Figure 3.17 shows the microarchitecture of the UMUL module. It has the following sub-modules: the index lookup, which is used to control the multiplexer that select the correct value from the input column, and the index opcode, which controls the correct operation for the signset module. The signset module changes the sign according to the entries in the U matrix. The values modified by the signset module are then moved into the vec_add module, which adds the four values together and divides them by two using right shift by one. Figure 3.17 only show one instance of the logic inside the UMUL module, but the real design has N copies.

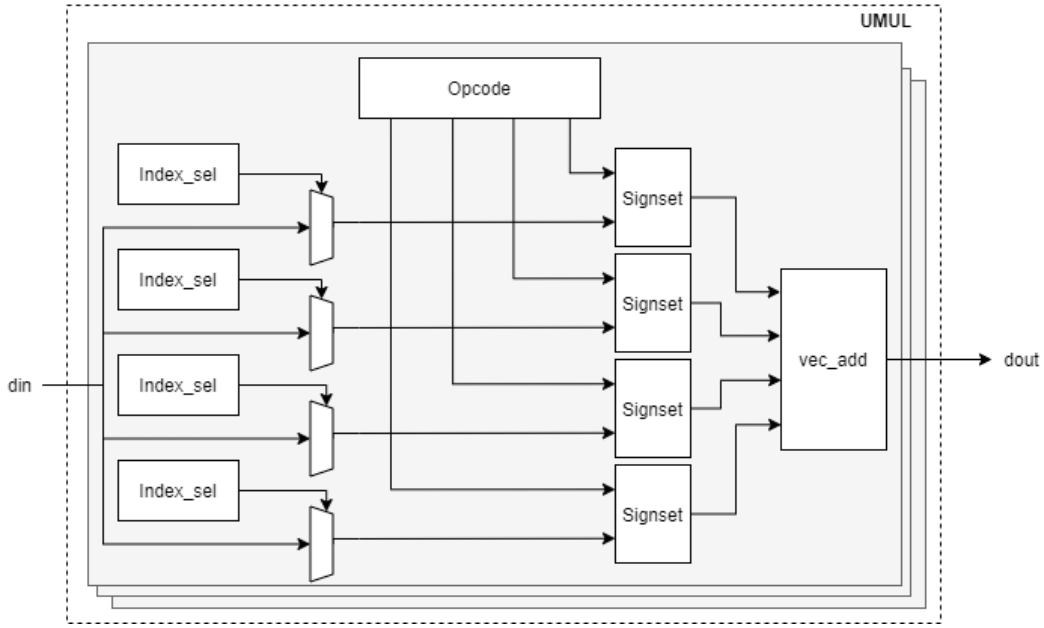


Figure 3.17: UMUL architecture

Index Lookup Module

The index lookup module indicates which values of the U matrix contain non-zero values. The zero values are ignored as multiplication by zero yields zero. The index lookup table is shown in Algorithm 1. If we take row 0 as an example, the U matrix has non-zero entries in positions 0, 3, 8, and 11, which means that the value for (0,0) in the resulting matrix is found by adding the $(0,0) + (3,0) + (8,0), (11,0)$ entries in the R matrix. Entry (1,0) and so forth can be found in a similar way by looking up the correct row index.

Algorithm 1 Index Lookup algorithm

if $row = 0$ or $row = 3$ or $row = 8$ or $row = 11$ **then**
  $I_A \leftarrow 0$, $I_B \leftarrow 3$, $I_C \leftarrow 8$, $I_D \leftarrow 11$
else if $row > 3$ and $row < 8$ **then**
  $I_A \leftarrow 4$, $I_B \leftarrow 5$, $I_C \leftarrow 6$, $I_D \leftarrow 7$
else
  $I_A \leftarrow 1$, $I_B \leftarrow 2$, $I_C \leftarrow 9$, $I_D \leftarrow 10$
end if

Index Opcode Module

The index opcode module is used to retrieve the correct opcode, which is used by the signset module to generate the correct operation. The index opcode module can be controlled by setting the appropriate row, and the module will output the corresponding opcode. For example, row = 0 has real ones in positions 0, 3, 8, and 11, and the remaining, together with the complex values, are set to zero. The module will therefore set the opcodes for the four real values to positive and the others to null. The same method is used for all the other rows.

Algorithm 2 Index Opcode algorithm

if $row = 2$ or $row = 3$ or $row = 5$ **then**
  $OP_A(Re) \leftarrow$ NULL, $OP_A(Im) \leftarrow$ POS
  $OP_B(Re) \leftarrow$ NULL, $OP_B(Im) \leftarrow$ NEG
  $OP_C(Re) \leftarrow$ NULL, $OP_C(Im) \leftarrow$ POS
  $OP_D(Re) \leftarrow$ NULL, $OP_D(Im) \leftarrow$ NEG
else if $row = 6$ or $row = 8$ or $row = 9$ **then**
  $OP_A(Re) \leftarrow$ NULL, $OP_A(Im) \leftarrow$ POS
  $OP_B(Re) \leftarrow$ NULL, $OP_B(Im) \leftarrow$ POS
  $OP_C(Re) \leftarrow$ NULL, $OP_C(Im) \leftarrow$ NEG
  $OP_D(Re) \leftarrow$ NULL, $OP_D(Im) \leftarrow$ NEG
else if $row = 7$ or $row = 10$ or $row = 11$ **then**
  $OP_A(Re) \leftarrow$ NEG, $OP_A(Im) \leftarrow$ NULL
  $OP_B(Re) \leftarrow$ POS, $OP_B(Im) \leftarrow$ NULL
  $OP_C(Re) \leftarrow$ POS, $OP_C(Im) \leftarrow$ NULL
  $OP_D(Re) \leftarrow$ NEG, $OP_D(Im) \leftarrow$ NULL
else
  $OP_A(Re) \leftarrow$ POS, $OP_A(Im) \leftarrow$ NULL
  $OP_B(Re) \leftarrow$ POS, $OP_B(Im) \leftarrow$ NULL
  $OP_C(Re) \leftarrow$ POS, $OP_C(Im) \leftarrow$ NULL
  $OP_D(Re) \leftarrow$ POS, $OP_D(Im) \leftarrow$ NULL
end if

Signset Module

The signet module modifies the sign when adding two complex values, as shown in Equation 3.39. The correct opcode is fetched from the index opcode module. According to Algorithm 2, if either the real or complex opcode is not NULL, the other one will be NULL and the signet, therefore, only has to look at if either the real or complex value has an opcode that is non-zero.

$$Out(Re), Out(Im) \begin{cases} +Re, +Im & \text{if OP(Re)} = \text{pos} \\ -Re, -Im & \text{if OP(Re)} = \text{neg} \\ -Im, +Re & \text{if OP(Im)} = \text{pos} \\ +Im, +Re & \text{otherwise.} \end{cases} \tag{3.39}$$

3.3.6 Unitary Transform Controller

Figure 3.18 shows the state diagram of the UT controller that controls the UT datapath. The UT module is enabled by setting **din_valid** high for one or more clock cycles. After **din_valid** is high the module will stay in the **st_fill** state where it will hold the **shift_enable** signal high for N clock cycles. After N clock cycles the controller changes to the **st_output** state where it will set **shift_enable** low for N number of clock cycles before changing back to **st_idle**. The shift register will be cleared whenever the controller stays in the **st_idle** state.

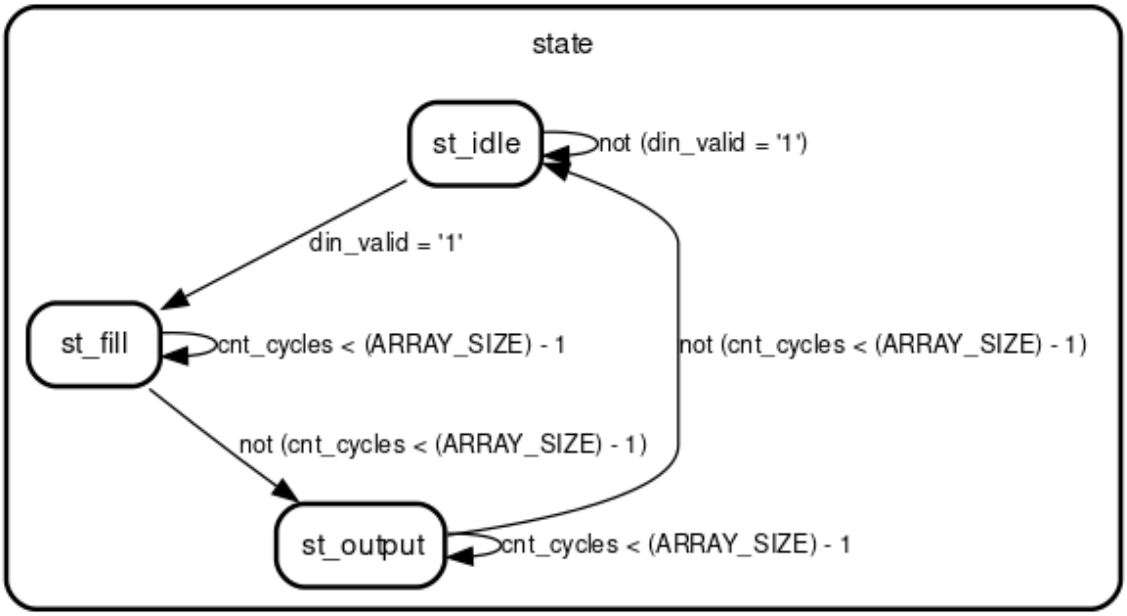


Figure 3.18: Unitary transform state diagram

3.3.7 RVT Synthesis Report

Figure 3.8 shows the synthesis report for the RVT module. The design uses 7% of the available LUTs, 1% of the registers, and 1% of the F7 multiplexers. The module can meet all timing constraints up to 100 MHz.

Table 3.8: Synthesis report for the RVT module

Parameter	Usage	Utilization (%)
Look-up tables	3780	7.10
Registers	967	0.91
Block ram	0	0
DSP slices	0	0
F7 Mux	336	1.26
Max clock frequency [MHz]	100	

Note: $W_L = 16$ and $N = 12$

3.4 Eigenvalue Decomposition

This section details the design of the Eigenvalue Decomposition (EVD) module. The EVD calculates the eigenvalue and eigenvector of a real symmetrical matrix. Two potential architectures were considered for solving the eigenvalue problem: Cyclic-by-row- and the parallel-Jacobi method. Computational time and hardware usage are important metrics for selecting a suitable design. Table 3.9 compares the two potential solutions.

Table 3.9: Time complexity and resource comparison between two EVD implementations

	Cyclic Jacobi (A)	Parallel Jacobi (B)
Time complexity	$O\left(\frac{N(N-1)}{2} \times 3W_L \times J_S\right)$	$O\left(3W_L \times J_S\right)$
Resource usage	N	$\frac{N(N-1)}{2}$

Note: $N \times N$: matrix size, W_L: word length, and J_S: number of Jacobi sweeps

Table 3.9 indicates that the computational time for the Cyclic-by-row method can be approximated to the square of the matrix size, while the parallel solution only depends on the word length and the number of sweeps. If we assume that $N = 12$, $W_L = 16$ and $J_S = 6$ then the delay for each method are the following:

$$\text{Delay(A)} : \frac{12(12-1)}{2} \times 3 \times 16 \times 6 = 19008 \tag{3.40}$$

$$\text{Delay(B)} : 3 \times 16 \times 3 = 288 \tag{3.41}$$

If we assume that the design can operate at 100 MHz, then the calculation time for the eigenvalue decomposition would be:

$$\text{Time(A)} = 19008 \times 0.01\mu s = 190.08\mu s \tag{3.42}$$

$$\text{Time(B)} = 144 \times 0.01\mu s = 2.88\mu s \tag{3.43}$$

Recall from 2.1 that each CTE has a maximum length of 160 µs which implies we ideally want to be able to calculate a new direction every 160 µs. The cyclic Jacobi uses 30 µs longer than the CTE window which means that it is impossible to fully utilize each CTE window. It is obvious that parallel implementation is much faster, even with a fairly small matrix, but this comes at the cost of increased hardware usage. From Table 3.9, we can see that the hardware is approximately constant for the Cyclic implementation, but the square of the matrix size for the parallel. This implies the designer must select a trade-off between computational time and area. For implementation on an FPGA and the desire for real-time performance, the parallel method would be the ideal choice assuming that there are enough resources and that the matrix size is rather small. For this thesis, the matrix size is fixed to 12×12, which is considered small enough for the parallel implementation to be feasible [36].

3.4.1 Eigenvalue Decomposition Module

Figure 3.19 shows the architecture of the EVD module. The **din_valid** and **dout_valid** signals indicate when the input and output data are valid. The data from the real-valued covariance matrix from the previous module is fed into the eigenvalue module. The eigenvector module does not need to be fed any data, but instead, it is initialized with an identity matrix. The eigenvector module is fed the rotation parameters one sign at a time by the eigenvalue module. The eigenvalue and eigenvector modules are controlled by a finite state machine inside the **evd_cntrl** module.

Figure 3.19: EVD architecture

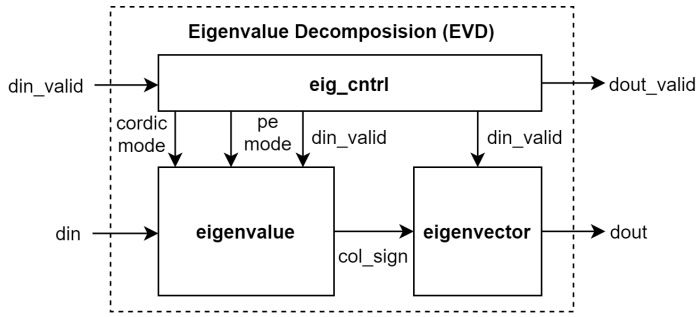
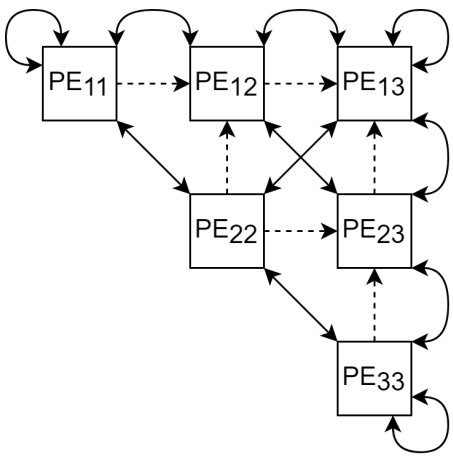


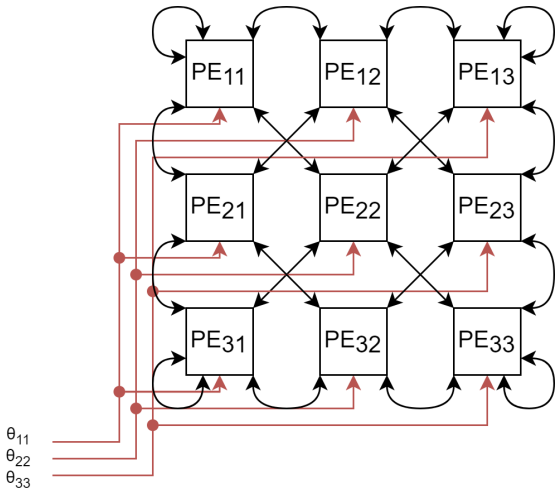
Figure 3.20 shows an overview of the systolic array used to calculate the eigenvalue. The design in the figure shows how it would look for a 6×6 matrix where each PE contains a 2×2 sub-matrix, but the design can be extended for any size as long as the matrix is symmetrical and even-numbered. The design works similarly to the one detailed in 2.6.3 but with some added optimizations. Instead of broadcasting the angle between PEs, it is possible to broadcast the rotation sign used by the CORDIC modules directly. This method removes 15 of 16 bits if 16-bit values are used. The sign is also directly connected to every PE on the same row and column, which allows every PE to work on the same clock cycle and therefore simplifies the control logic. By directly broadcasting the sign, the off-diagonal units can work in an on-the-fly mode where the diagonal units control the sign controlling the rotation direction. The CORDIC modules inside the off-diagonal PEs do not need an angle accumulator since the sign is sent from the diagonal PEs resulting in reduced size.

Figure 3.20: An upper triangle 3×3 Brent-Luk-EVD array, where $n = 6$ for an 6×6 symmetric matrix. Used for solving the eigenvalue problem.



The eigenvectors can also be found using a similar structure, except it needs to be a full 6×6 matrix. The eigenvector array does not need to calculate the optimal angle θ. Instead, it uses the angle calculated in the eigenvalue array. The angle is broadcast one sign at a time in the same way as the eigenvalue array, therefore, simplifying the routing. Each column in the eigenvector array uses the same angle as the one in the same column in the eigenvalue array, as shown in Figure 3.21. Since the PEs in the eigenvector array do not have to calculate the angle themself, they can also utilize a simplified CORDIC without an angle accumulator. The values inside the eigenvector array are initialized with an identity matrix with ones along the diagonal and zeroes elsewhere. The eigenvectors can be found along each column in the array after sufficient Jacobi sweeps have been completed. The eigenvalues will already be sorted in ascending order along the diagonal towards the lower corner. The corresponding eigenvector will also follow this order so the that the eigenvector corresponding to the largest eigenvalue is found at the rightmost column. This column is then sent to the spectral peak search module.

Figure 3.21: A 3×3 Brent-Luk-EVD array, where $n = 6$ for an 6×6 symmetric matrix.



Both the eigenvalue and eigenvector modules are controlled by the **evd_cntrl** module. The controller is responsible for selecting the correct mode of each processing element and for exchanging data at the correct time. Table 3.10 shows the different modes in one sweep of the parallel Jacobi method. For multiple sweeps, the controller will loop through these states multiple times. In the first state, only the diagonal PEs are active calculating the optimal rotation angle and all other PEs will be idle. In the second state, the diagonal and off-diagonal PEs will calculate the left-hand side rotation while vector PEs are idle. In the third state, all the PEs will calculate the right-hand side rotation. In the last state, all PEs will exchange their values with adjacent PEs.

Table 3.10: EVD module states where W_L is equal to the word length

State	Time	DPE	OPE	VPE
Calculate optimal angle	W_L	Vectoring	Idle	Idle
LHS rotation	W_L	Rotation	Rotation	Idle
RHS rotation	W_L	Rotation	Rotation	Rotation
Exchange	1	Exchange	Exchange	Exchange

3.4.2 Diagonal Processing Element

The Diagonal Processing Element (DPE) calculates the optimal angle θ and the double-sided rotation. The optimal angle is calculated using Equation 3.44.

$$\theta = \frac{1}{2} arctan2 \left[\frac{2 \times b}{a - d} \right] \tag{3.44}$$

where a, b, c, d are the entries of a two-by-two matrix. The left-hand rotation is calculated as follows:

$$a_1, c_1 = rotate(a_0, c_0, \theta) \tag{3.45}$$
$$b_1, d_1 = rotate(b_0, d_0, \theta) \tag{3.46}$$

where θ is the optimal angle calculated in Equation 3.44. The corresponding right-hand side rotation is calculated by:

$$a_2, b_2 = rotate(a_1, b_1, \theta) \tag{3.47}$$
$$c_2, d_2 = rotate(c_1, d_1, \theta) \tag{3.48}$$

Figure 3.22 shows the architecture of the DPE module. The module has four data inputs corresponding to the four values in a 2×2 matrix and the same number of outputs. The DPE uses two types of CORDIC modules called CORDICA and CORDICB. The CORDICA includes a full CORDIC module that works in vectoring and rotation modes. The CORDICA module is used for both finding the optimal rotation angle and rotating a vector according to the rotation angle. The sign is sent sign-by-sign to the CORDICB while working in rotation mode. Both CORDIC modules include scaling at the output of the X and Y registers to correct the output error created by using CORDIC. The angle correction module guarantees that the input values when working in vectoring mode are within the right-hand side of the unit circle. The input to the Z register is set to zero when the CORDICA module is enabled while in vectoring mode and the output is kept inside an angle storage register. The angle storage retrieves the optimal angle while in rotating mode, as the values stored in the internal Z register are overwritten.

The output from the CORDIC modules needs to be scaled by a factor of $1/K$ to account for the error generated when working in rotation mode. The scaling factor can be implemented in hardware using a combination of shift, addition, and subtracts, as shown in Equation 3.49 and Figure 3.23.

$$\frac{1}{K} = 0.6073 \approx 2^{-1} + 2^{-3} - 2^{-6} - 2^{-9} - 2^{-13} \tag{3.49}$$

Figure 3.11 shows the synthesis report for the OPE. Each OPE uses 491 look-up tables and 106 registers and can meet all timing constraints at a clock frequency of up to 150 MHz.

Figure 3.22: Diagonal Processing Element (DPE) architecture

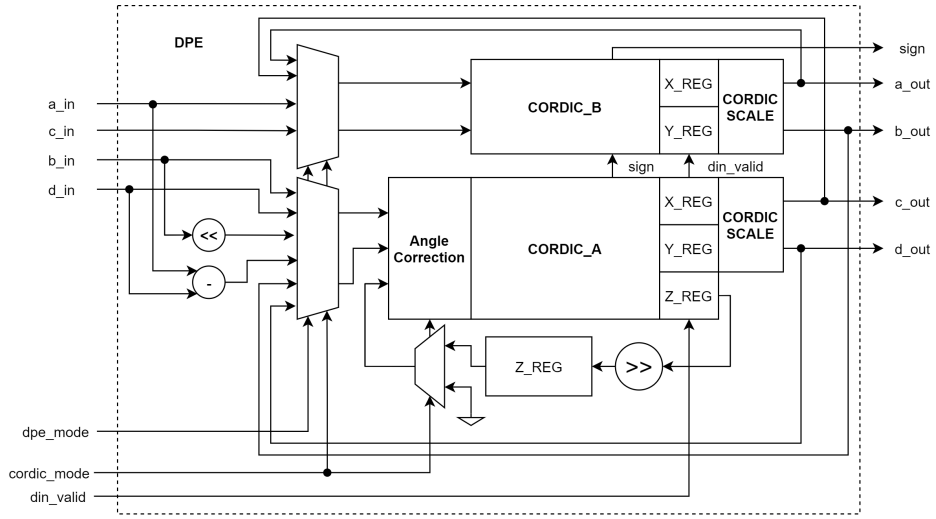


Figure 3.23: CORDIC Scale architecture

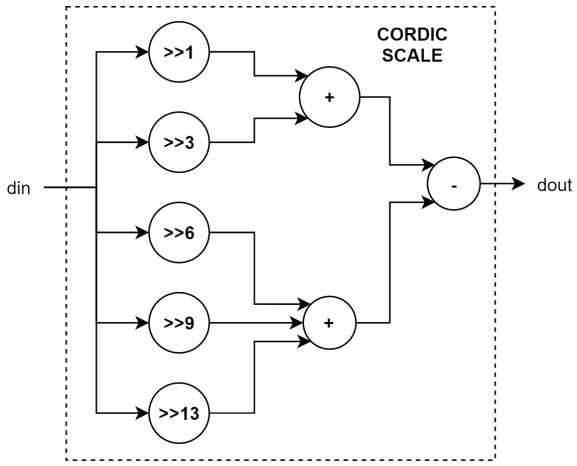


Table 3.11: Synthesis report for the DPE

Parameter	Value
Look-up tables	491
Registers	106
Maximum clock frequency [MHz]	150

3.4.3 Off-diagonal Processing Element

The Off-diagonal Processing Element (OPE) is a part of the eigenvalue module. It is stripped-down version of the DPE. It is used to calculate a two-sided rotation of a 2×2 sub-matrix. The left-hand side rotation is calculated as the following:

$$a_1, c_1 = rotate(a_0, c_0, \theta_{row}) \tag{3.50}$$

$$b_1, d_1 = rotate(b_0, d_0, \theta_{row}) \tag{3.51}$$

and the corresponding right-hand side rotation:

$$a_2, b_2 = rotate(a_1, b_1, \theta_{col}) \tag{3.52}$$
$$c_2, d_2 = rotate(c_1, d_1, \theta_{col}) \tag{3.53}$$

where a, b, c, d are the entries of a 2×2 sub-matrix. This calculation can be implemented in hardware using two CORDIC rotational mode modules as shown in Figure 3.24. The **ope_mode** signal is used to control the input data MUX. The MUX control whenever the CORDIC modules will calculate a left- or right-hand side rotation. The **ope_mode** signal also control the sign MUX which will either pick the sign from the DPE on the same row or column. Both the X and Y output values are corrected at the end to account for the error caused by CORDIC.

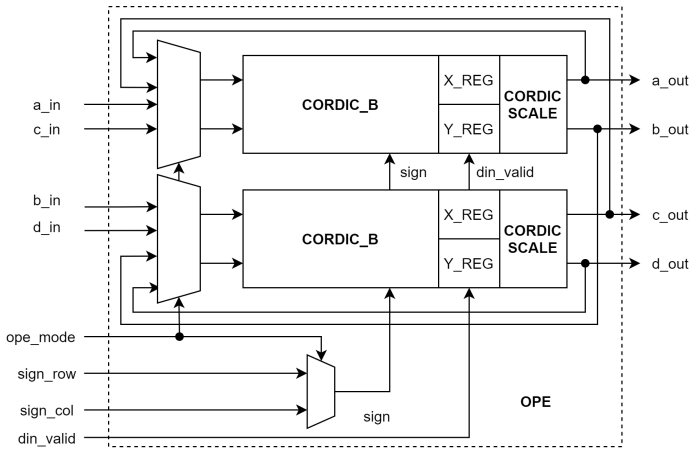


Figure 3.24: Off-Diagonal Processing Element (OPE) architecture

Figure 3.12 shows the synthesis report for the OPE. Each OPE uses 360 look-up tables and 74 registers and is able to meet all timing constraints at a clock frequency up to 150 MHz.

Table 3.12: Synthesis report for the OPE generated using Vivado 2022.2

Parameter	Value
Look-up tables	360
Registers	74
Maximum clock frequency [MHz]	150

3.4.4 Vector Processing Element

The Vector Processing Element (VPE) is part of the eigenvector module. The VPE only needs to calculate a single rotation of a 2×2 sub-matrix. It consists of two CORDICB and associated CORDIC scaling modules. It calculates the following:

$$a_1, b_1 = rotate(a_0, b_0, \theta_{col}) \tag{3.54}$$
$$c_1, d_1 = rotate(c_0, d_0, \theta_{col}) \tag{3.55}$$

where θ is the optimal angle sent sign-by-sign from the DPE on the same column, the **din_valid** signal is held high for one clock cycle in order to exchange values with adjacent VPE modules.

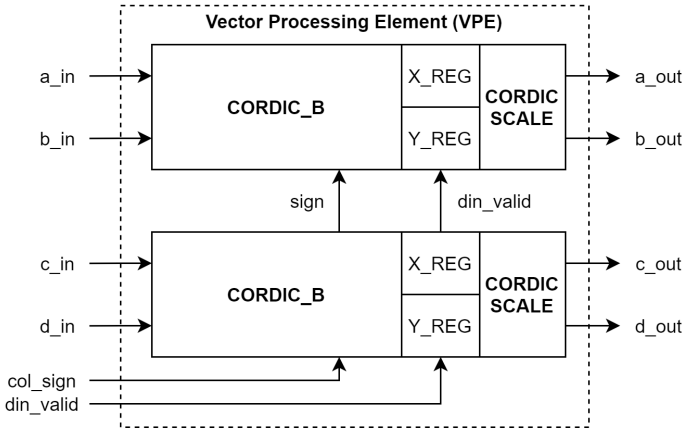


Figure 3.25: Vector Processing Element (VPE) architecture

Table 3.13 shows the synthesis report generated by Vivavdo. Each VPE module uses 302 LUTs and 74 registers while meeting timing requirements up to 200 MHz.

Table 3.13: Synthesis report for the VPE generated using Vivado 2022.2

Parameter	Value
Look-up tables	302
Registers	74
Maximum clock frequency [MHz]	150

3.4.5 EVD synthesis report

Figure 3.14 shows the synthesis report for the EVD module. The module uses 36% and 4% of the available look-up tables and registers receptively. No DSP slices or block ram is used. The module can meet timing constraints of up to 100 MHz.

Table 3.14: Synthesis report for the EVD module

Parameter	Usage	Utilization (%)
Look-up tables	19 382	36.43
Registers	4 306	4.05
Block ram	0	0
DSP slices	0	0
Max clock frequency [MHz]	100	

Note: $C_I = W_L = 16$, $N = 12$, and $J_S = 6$

3.5 Spectral Peak Search

This section will outline and briefly explain the Spectral Peak Search (SPS) module designed by Jacob August Rangnes [5]. The spectral peak search involves finding the largest peak using a 2D search. The goal of the search is to find the combination of θ and φ, which yields the highest value of $||a^H(\theta, \varphi)E_S||$.

A common trade-off in such a search function is choosing the best step size. The step size is directly related to the resolution of the final result, but reducing the step size significantly increases search time. Jacob opted to use a two-step search method to reduce the number of searches while still having good resolution [5]. The design involves utilizing a course and fine search where the course search has a high step size to narrow down the search area before the fine search begins.

Figure 3.26 shows the hardware for the SPS core. The **dataIn_valid** signal indicates that the EVD module is finished and that the eigenvector E_S is available at the input. The search core can start when the **dataIn_valid** is set high. Both $a(\theta, \varphi)$ and E_S are of size (1,12) and (12,1), respectively, and they are passed into the search core, where they are multiplied and compared to previously multiplied combinations. The final direction can be found at the output when **dataOut_valid** is set high.

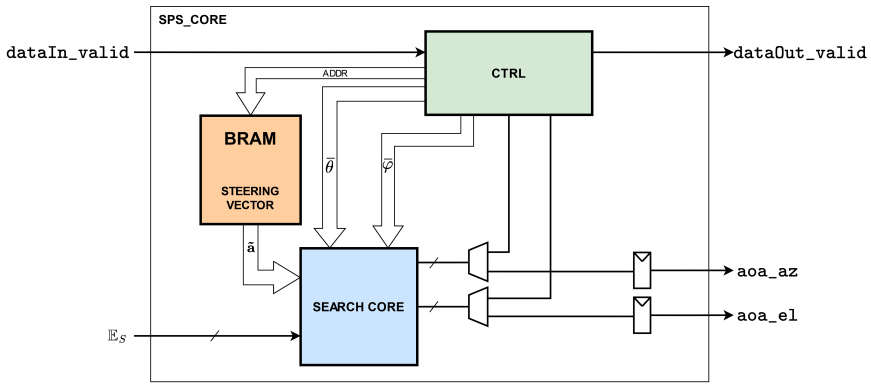


Figure 3.26: SPS Core architecture [5]

Table 3.15 shows the synthesis report for the SPS module designed by Jacob. The module uses 11%, 6%, 86%, and 73% of the available look-up tables, registers, block ram, and DSP slices, respectively.

Table 3.15: Synthesis report for the SPS module

Parameter	Usage	Utilization (%)
Look-up tables	6 076	11.42
Registers	6 364	5.98
Block ram	120	85.71
DSP slices	161	73.18
Max clock frequency [MHz]	143	

Note: $W_L = 16$, $N = 12$, $V_M = 10$: number of parallel multiplicators

Chapter 4

Results

This chapter will present the results obtained while testing the software and hardware. The chapter starts by demonstrating how the various parameters of the MUSIC algorithm will impact performance using a High Level Model (HLM). Then the key metrics from the synthesis and implementation reports from Vivado are highlighted. The synthesis and implementation reports will show how the parameters, such as clock frequency and bit-width, affect Performance, Power and Area (PPA). The chapter will also compare the accuracy and run time between the hardware and software. Finally, the effect of the error generated by the EVD module is demonstrated. The stimuli data for both the HLM and the hardware is generated as shown in Appendix A. The results are discussed in Chapter 5.

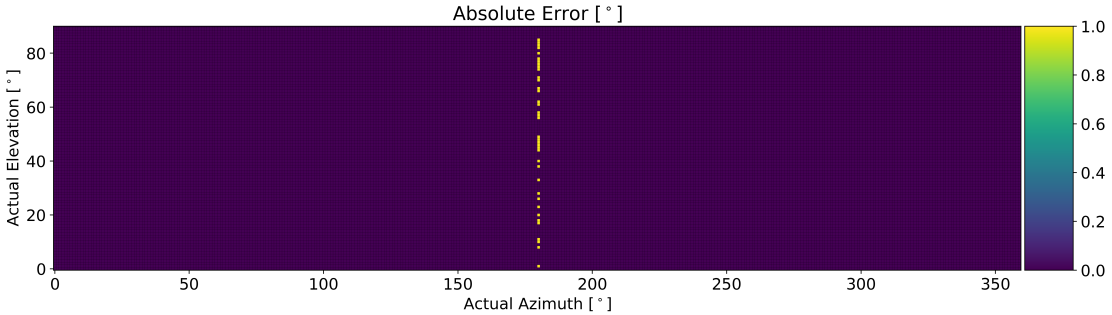
4.1 High Level Testing

High-level testing is done to verify that the algorithm can achieve the desired accuracy or, alternatively, how large the deviation is. The high-level testing will investigate the effects of the real-valued transformation and how robust the algorithm is against noise, round-of errors, and the number precision used. The testing will also see if there is any loss of accuracy by using a two-step search instead of one. The high-level testing is accomplished by using a model written in MATLAB and Python. The Covariance Matrix Calculation (CMC), Real-valued Transformation (RVT), and Eigenvalue Decomposition (EVD) is implemented in MATLAB, while the final Spectral Peak Search (SPS) is implemented in Python. For the following plots, the algorithm's error is for simplicity given in absolute error. The absolute error gives us the error of both the azimuth and elevation as shown in Equation 4.1.

$$Abs_{err} = \sqrt{az_{err}^2 + el_{err}^2} \tag{4.1}$$

Figure 4.1 shows the accuracy of the original MUSIC algorithm written in Python without using a real-valued transformation. The plot uses a heat map to indicate where the largest errors are found. The scale on the right of the plot indicates which color corresponds to the highest error. The following tests will calculate the absolute error where the elevation is limited to $\theta_k(0 < \theta < \pi/2)$ and azimuth is between $\varphi_k(0 < \varphi < 2\pi)$ with a step size of 1°. The following tests use 32-bit and without any noise added. The complex-valued MUSIC algorithm has 1° error around $\varphi = 180$ and zero elsewhere.

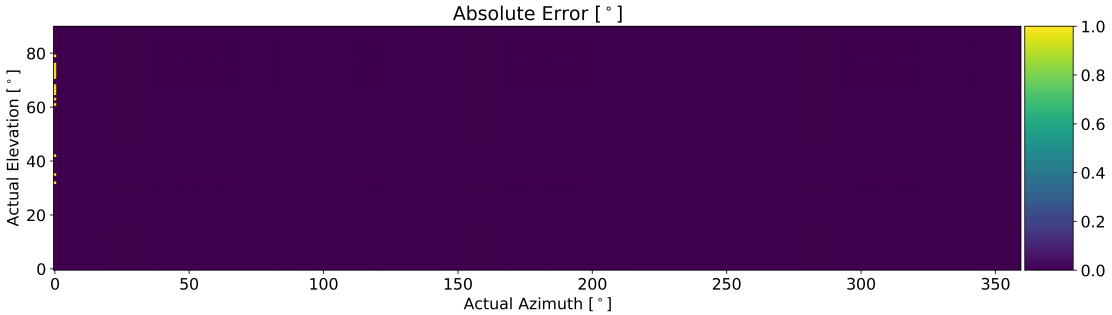
Figure 4.1: Complex-valued MUSIC algorithm with all possible angles without noise



Note: Complex-valued, two-step search, $W_L = 32$

Figure 4.2 shows the accuracy of the real-valued MUSIC algorithm with the same setup as in Figure 4.1, but with a real-valued transformation added. The real-valued MUSIC algorithm has 1° of error at $\varphi = 0$ and zero elsewhere. This indicates that the real-value transformation has little impact on the actual accuracy.

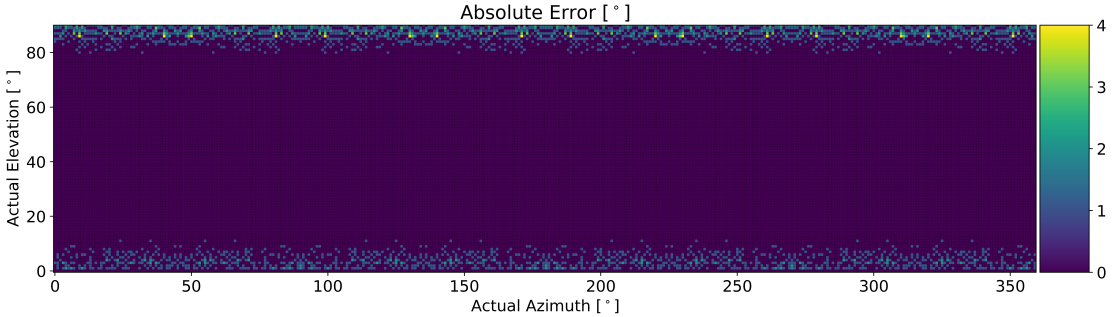
Figure 4.2: Real-valued MUSIC algorithm with all possible angles without noise



Note: Real-valued, two-step search, $W_L = 32$

Figure 4.3 has the same setup as Figure 4.2 but with 16-bit precision instead of 32. We can see that the reduction in precision greatly affects the accuracy of the calculated AoA. The 16-bit version has a maximum error of 4° of error along $\theta = 0$ and $\theta = 90$. The 16-bit implementation also has a more evenly distributed error along high and low elevations.

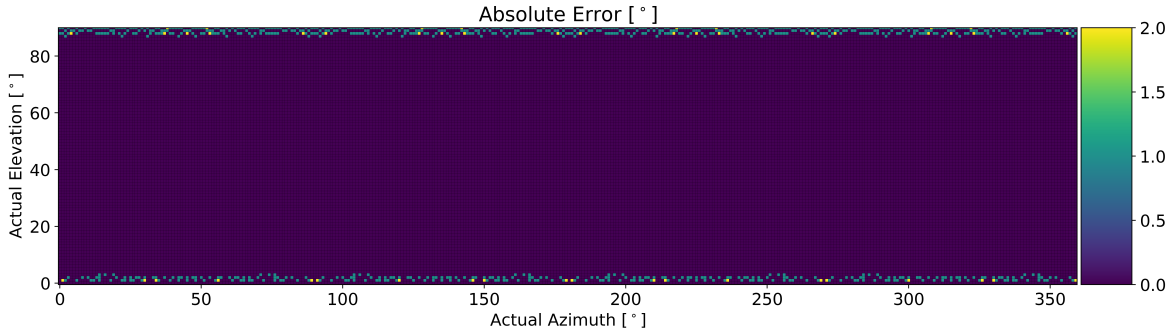
Figure 4.3: Real-valued MUSIC algorithm with all possible angles without noise



Note: Real-valued, two-step search, $W_L = 16$

Figure 4.4 shows that the error in the AoA is significantly reduced when using 18-bit accuracy compared to 16. The maximum error is reduced from 4° to 2°, and less error overall.

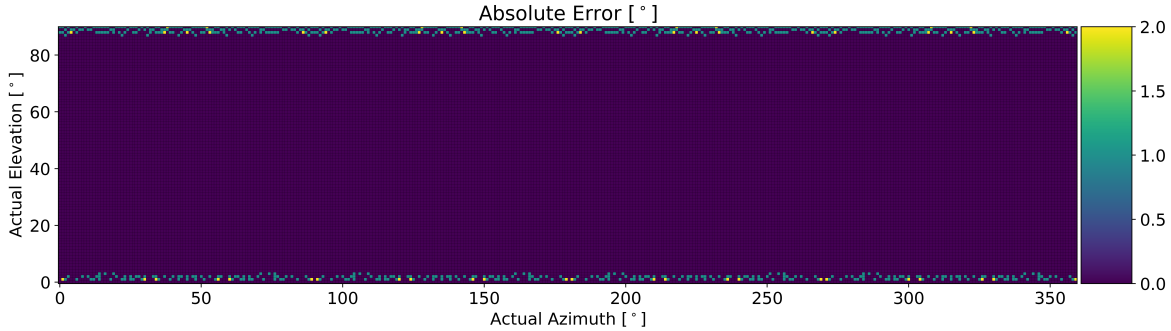
Figure 4.4: Real-valued MUSIC algorithm with all possible angles without noise



Note: Real-valued, two-step search, $W_L = 18$

Figure 4.5 shows that there is no visible difference in accuracy between one and two searches.

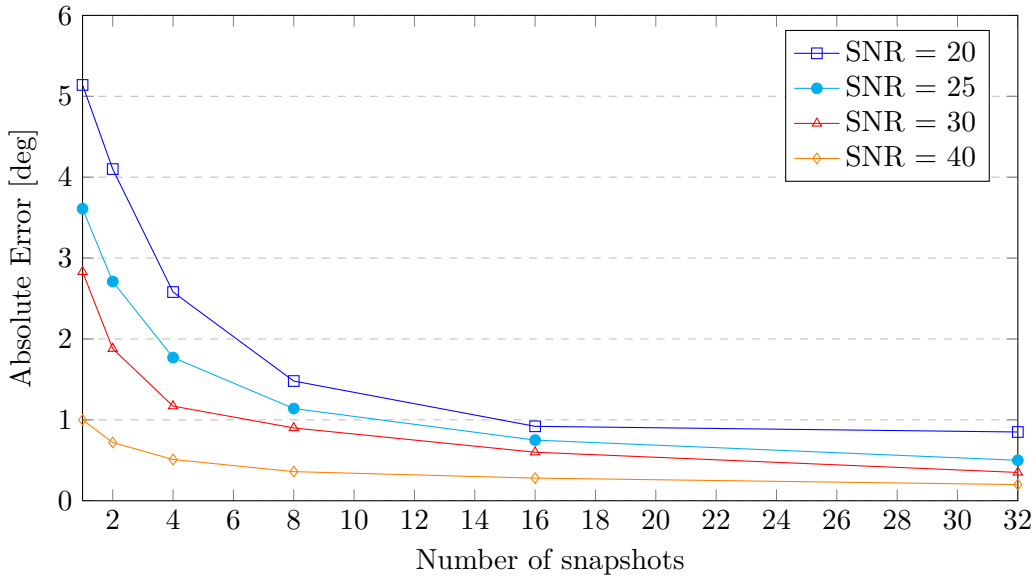
Figure 4.5: Real-valued MUSIC algorithm with all possible angles without noise



Note: Real-valued, one step search, $W_L = 18$

Figure 4.6 shows the absolute error of the predicted AoA using different numbers of snapshots and signal-to-noise ratios (SNR). All measurements use 32-bit precision. Figure 4.6 shows the error is largely dependent on both the noise floor and the number of snapshots taken. The largest difference in accuracy is found when the number of snapshots is between one to eight. In order to achieve the desired accuracy of one degree, the algorithm needs to use between eight to sixteen snapshots

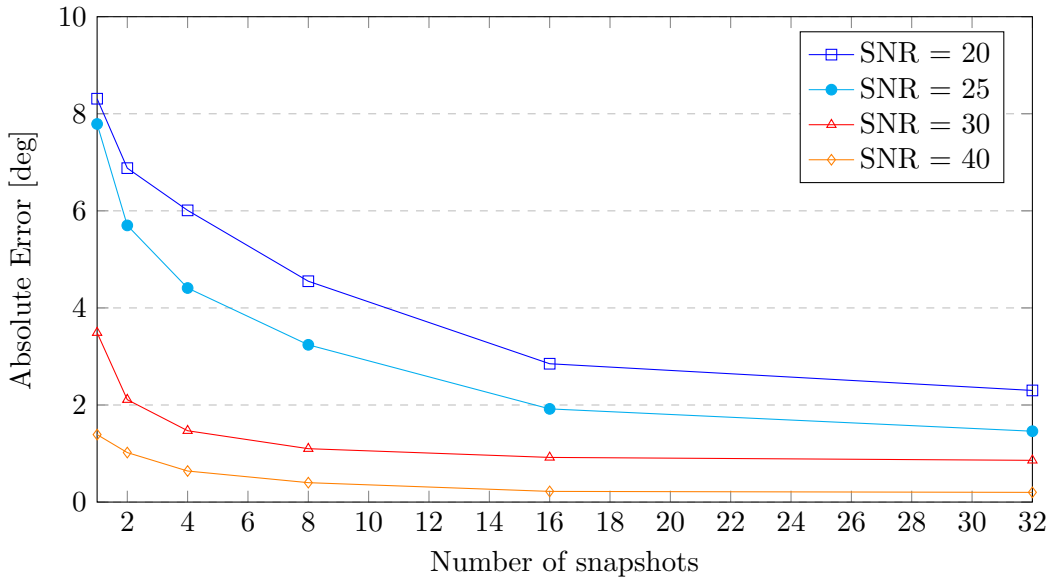
Figure 4.6: Accuracy depending on the number of snapshots using $W_L = 32$



Note: $n = 10$, $\Delta\theta_c = 3$, $\Delta\varphi_c = 1$, $\Delta\theta_f = \Delta\varphi_f = 0.1$, $C_I = W_L = 32$ and $J_S = 6$.

Figure 4.7 shows the effect the number of snapshots has on the accuracy of the MUSIC algorithm with different signal-to-noise ratio levels. Figure 4.7 uses a bit-width of 16, and we can see that the error is higher compared to the full 32-bit.

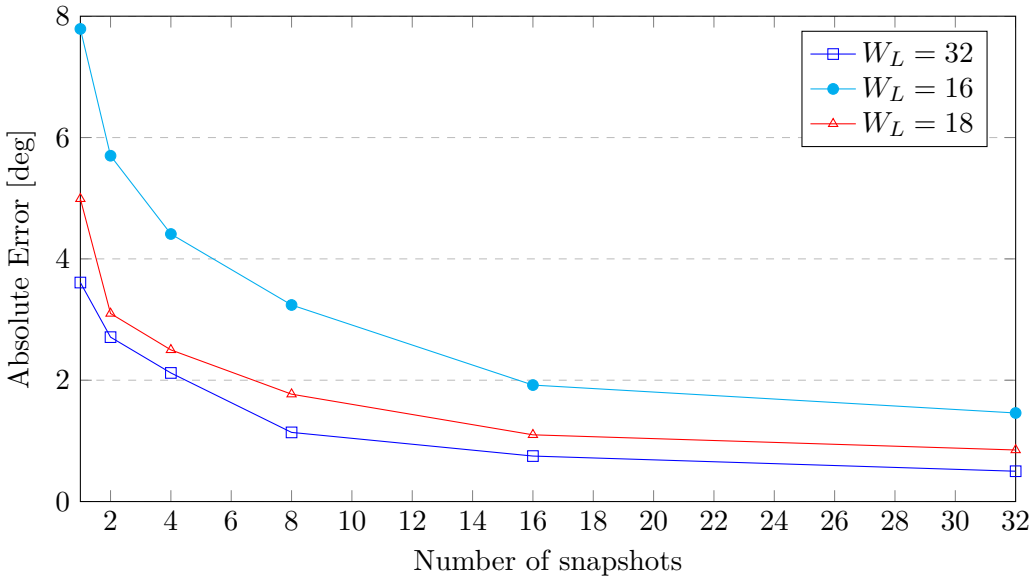
Figure 4.7: Accuracy depending on the number of snapshots using $W_L = 16$



Note: $n = 10$, $\Delta\theta_c = 3$, $\Delta\varphi_c = 1$, $\Delta\theta_f = \Delta\varphi_f = 0.1$, $C_I = W_L = 16$ and $J_S = 6$.

Figure 4.8 shows the effect the number of bits used in the MUSIC algorithm has on the accuracy of the AoA. All calculations have a signal-to-noise ratio of 25. From the figure, we can see that the error is large when the number of snapshots is less than four but that all bit widths are trending downward with sufficient snapshots. With four snapshots taken, the 16-bit algorithm has an error of around four, while the others have an error close to two. At eight snapshots, the full precision version is able to achieve an error of almost one, while the others still have a high degree of error. At 16 snapshots, the 18-bit and 32-bit versions can achieve an error of less than one degree, while the 16-bit is around two degrees.

Figure 4.8: Accuracy depending on the number of snapshots and bit-width



Note: $n = 10$, $\Delta\theta_c = 3$, $\Delta\varphi_c = 1$, $\Delta\theta_f = \Delta\varphi_f = 0.1$, $C_I = 16$, $J_S = 6$ and SNR $= 25$

4.2 Utilization and Timing

This section will highlight key results from the synthesis and the implementation of each module and the MUSIC core. The synthesis and implementation reports include resource usage, timing information, and power draw. The results are generated by synthesizing the design using Vivavdo version 2022.2. Table 4.1 shows an overview of the maximum clock frequencies of all four components of the MUSIC algorithm and the MUSIC core. The MUSIC core can meet timing constraints at a clock frequency of 100 MHZ. The core uses 622 clock cycles and a calculation time of 6.22 µs for each AoA calculation at a frequency of 100 MHz.

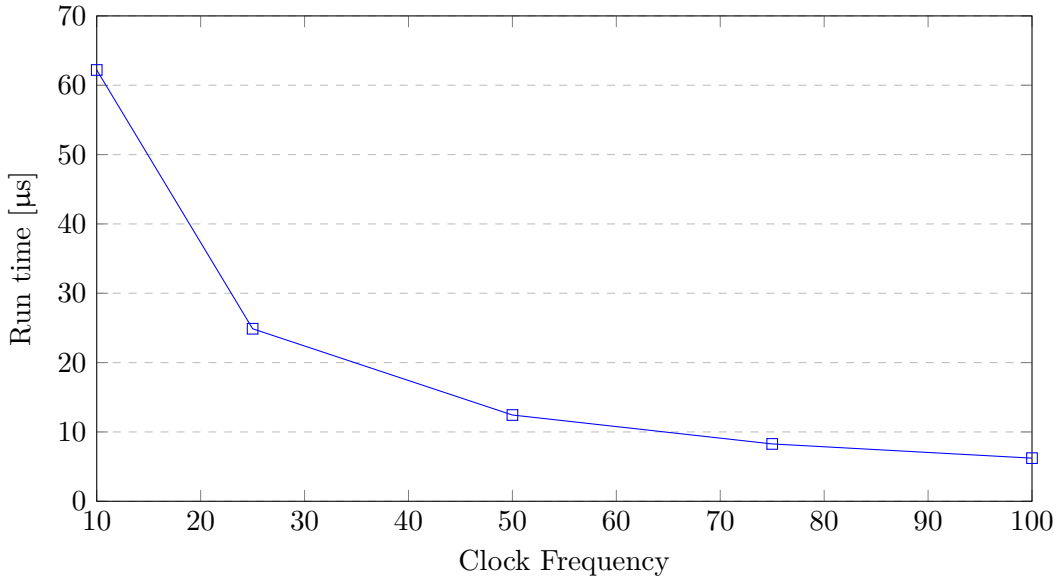
Table 4.1: Timing report

	f_{max}	Clock cycles	Time used [µs]
Correlation Matrix	160	54	0.54
Real-value Transformation	100	24	0.24
Eigenvalue Decomposition	100	288	2.88
Spectral Peak Search	140	242-247	2.40 - 2.47
MUSIC core	100	622	6.22

Note: $C_I = W_L = 16$, $N = 12$, $M = 4$, $J_S = 6$ $\Delta\theta_c = \Delta\varphi_c = 4$, $\Delta\theta_f = \Delta\varphi_f = 1$ and $f_{clk} = 100$ MHz

Figure 4.9 shows the run time of the MUSIC core at different clock frequencies. The largest speedup is found between 10 to 50 MHz. Reducing the clock frequency after 50 MHz has a limited reduction in run time.

Figure 4.9: Run time of the MUSIC core depending on clock frequency



Note: $C_I = W_L = 16$, $N = 12$, $M = 4$, $J_S = 6$ $\Delta\theta_c = \Delta\varphi_c = 4$, and $\Delta\theta_f = \Delta\varphi_f = 1$

Table 4.2 shows an overview of the total resource usage of the MUSIC core and each module on the Zynq Z2 FPGA. The design uses 59%, 20%, 89%, 85%, and 3% of the available look-up tables, registers, DSP slices, block ram, and F7 multiplexers, respectively.

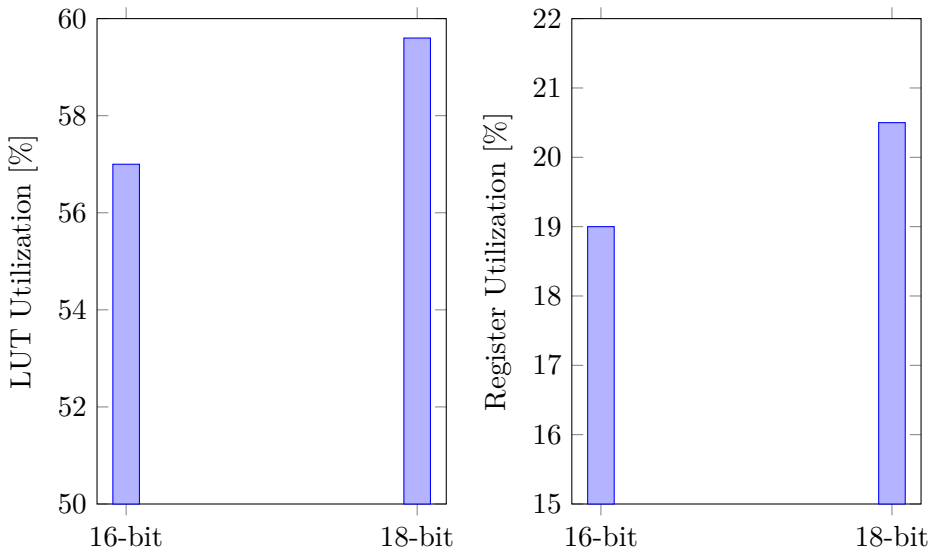
Table 4.2: Utilization report

	LUTs	Registers	DSP slices	Block ram	F7 Muxes
Correlation Matrix	3 424	2 816	36	0	0
Real-value Transformation	3 780	967	0	6	360
Eigenvalue Decomposition	19 218	5 964	0	0	0
Spectral Peak Search	6 076	6 364	161	120	495
MUSIC Core	31 711	21 425	197	120	855
Device utilization (%)	59.60	20.14	89.55	85.71	3.21

Note: $C_I = W_L = 16$, $N = 12$, $M = 4$, $J_S = 6$ $\Delta\theta_c = \Delta\varphi_c = 4$, and $\Delta\theta_f = \Delta\varphi_f = 1$

Figure 4.10 shows the LUT and register usage depending on the number of bits used in the MUSIC core. The number of DSP slices, block ram, and multiplexers are not dependent on the number of bits and, therefore, not added to the comparison. The 16-bit implementation uses 59.6% and 20.14% of the available look-up tables and registers, respectively. The 18-bit implementation has about 2% more look-up tables and 1.5% registers compared to the 16-bit.

Figure 4.10: LUT and register utilization depending on bit-width



Note: $\Delta\theta_c = 3$, $\Delta\varphi_c = 1$, $\Delta\theta_f = \Delta\varphi_f = 1$, $C_I = 16$ and $J_S = 6$.

Table 4.3 shows the power report for each module and the total power usage for the MUSIC core. The design uses 0.14 W of static power and 1.3 W of dynamic power at an ambient temperature of 25 °C. This results in a total power draw of 1.4 W at 100 MHz.

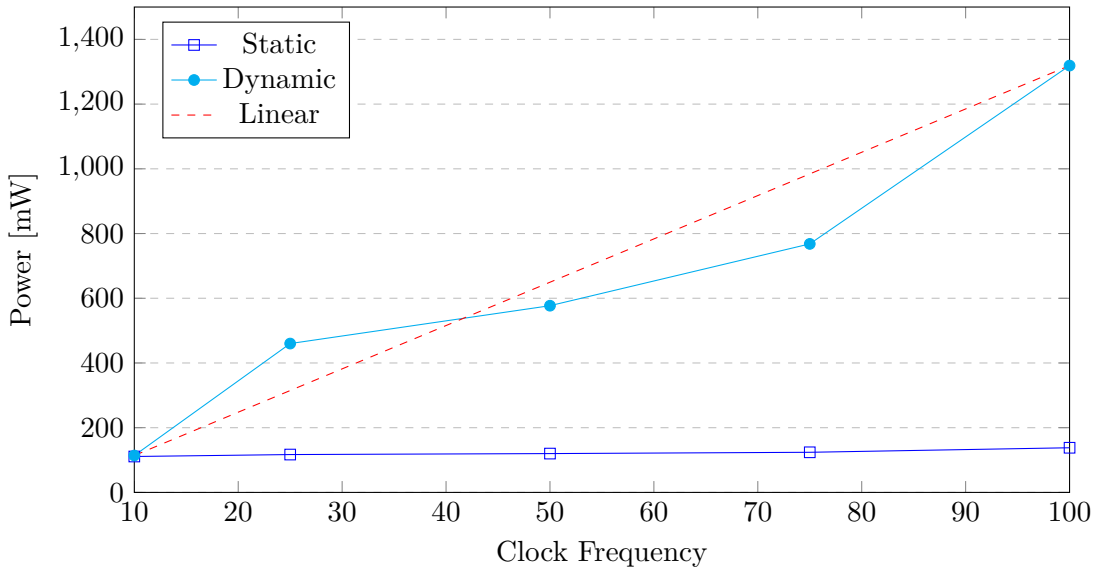
Table 4.3: Power report at ambient temperature

	Static power [mW]	Dynamic power [mW]
Correlation Matrix	110	225
Real-value Transformation	106	78
Eigenvalue Decomposition	119	836
Spectral Peak Search	122	598
MUSIC Core	138	1319

Note: $C_I = W_L = 16$, $N = 12$, $M = 4$, $J_S = 6$ $\Delta\theta_c = \Delta\varphi_c = 4$, $\Delta\theta_f = \Delta\varphi_f = 1$, $f_{clk} = 100$ MHz, and $T_A = 25°\text{C}$

Figure 4.11 shows the static and dynamic power of the MUSIC core at different clock frequencies. The static power indicated in blue is almost independent of the clock frequency, with a minimal difference between 10 and 100 MHz. The core draws 111 mW of static power at 10 MHz and 138 mW at 100 MHz. The dynamic power highly depends on the frequency, as shown in cyan. The dynamic power is almost linearly dependent on the clock frequency, as shown by the dotted red line. At 100 MHz, the core uses 1319 mW of dynamic power compared to 577 mW at 50 MHz, which is close to half the power. At 10 MHz, the core only uses 114 mW of dynamic power.

Figure 4.11: Power draw depending on clock frequency



Note: $C_I = W_L = 16$, $N = 12$, $M = 4$, $J_S = 6$ $\Delta\theta_c = \Delta\varphi_c = 4$, $\Delta\theta_f = \Delta\varphi_f = 1$, and $T_A = 25°\text{C}$

4.3 Software and Hardware comparison

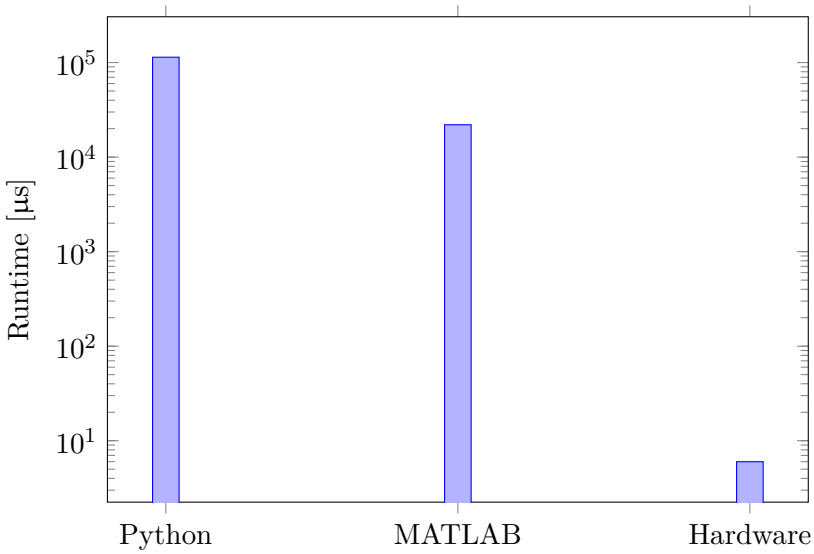
This section will compare the run time and accuracy of the hardware and software implementation of the MUSIC algorithm. Table 4.4 shows the system used for testing the software implementation. The computer system used for comparison draws roughly 60 W during the execution of the MUSIC algorithm.

Table 4.4: Software testing system

	Component name
CPU	Intel Core i7-12700K 5.0GHz
RAM	Kingston FURY Beast DDR5 5600MHz 16GB
Storage	Samsung 970 EVO Plus M.2 NVMe SSD 1TB

The run time comparison between hardware and software implementation considers two different implementations of the MUSIC algorithm in software. The first implementation uses the built-in MUSIC algorithm function in MATLAB, while the second uses the High Level Model written in Python. The run time of the MUSIC core is taken from the simulation of the design. Figure 4.12 compares the different implementations. The run time of the software implementations is run multiple times, and the average is taken to account for background tasks in the operating system. The MATLAB function can compute the AoA in 22 ms, while the Python function uses 114 ms. Compared to the software, the hardware uses 6.22 µs to compute the AoA, which is four orders of magnitude faster than the HLM in Python.

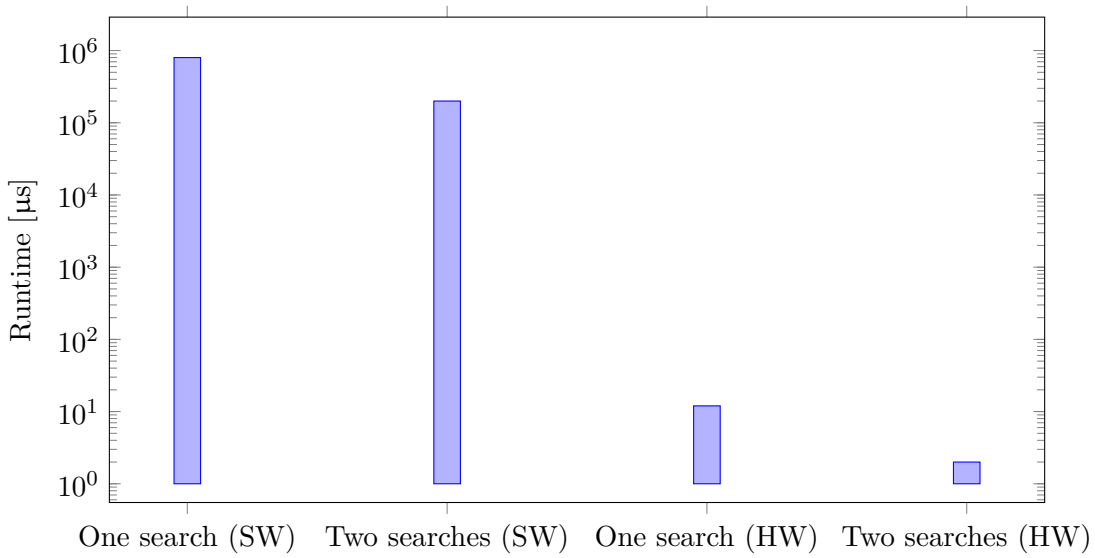
Figure 4.12: Time comparison between software and hardware



Note: $\Delta\theta_c = 3$, $\Delta\varphi_c = 1$, $\Delta\theta_f = \Delta\varphi_f = 1$, $C_I = W_L = 16$ and $J_S = 6$.

Figure 4.13 shows the time it takes for the search function to find the AoA with one and two searches for both software and hardware. In software, the two search method reduces the search time from 800 ms to 200 ms. In hardware, the search time is reduced from 12 µs to 2 µs.

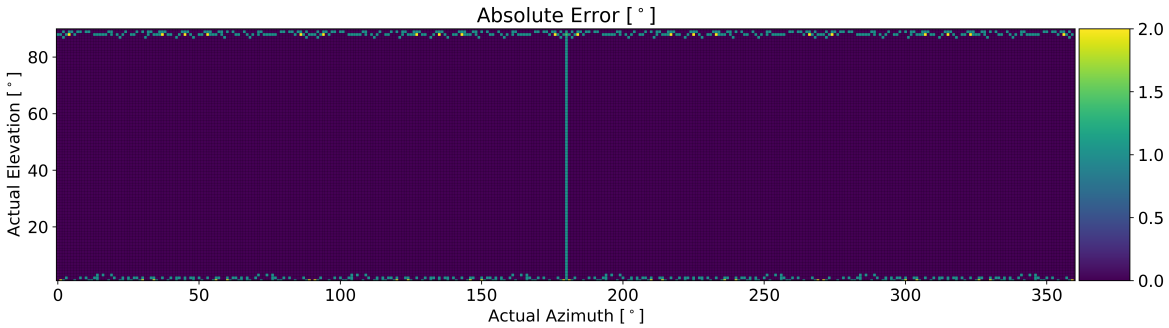
Figure 4.13: Search time comparison



Note: $\Delta\theta_c = 3$, $\Delta\varphi_c = 1$, $\Delta\theta_f = \Delta\varphi_f = 1$, $C_I = W_L = 16$ and $J_S = 6$.

Figure 4.14 shows the accuracy of the MUSIC algorithm simulated in hardware with one search and 18-bit precision. The following test is without any noise added. The one-search implementation has a maximum error of 2° along extreme elevation angles. The one-search implementation also has 1° of error at $\varphi = 180$.

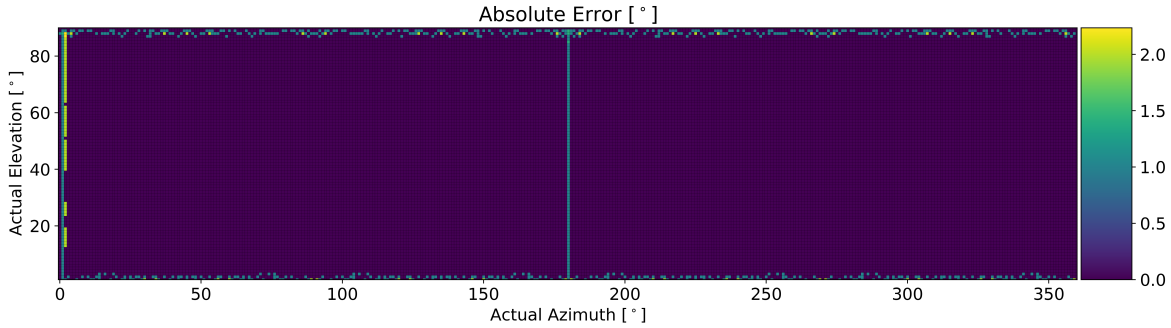
Figure 4.14: Hardware Real-valued MUSIC algorithm with one search and $W_L = 18$



Note: Real-valued, one step search, $W_L = 18$

Figure 4.15 shows the hardware simulation using two searches. The two search implementation has 2° of error at $\theta = 0$ and $\theta = 90$, similar to Figure 4.14. The two-step implementation also has additional errors at $\varphi = 0$.

Figure 4.15: Hardware Real-valued MUSIC algorithm with two searches and $W_L = 18$



Note: Real-valued, Two-step search, $W_L = 18$

4.4 Eigenvalue Decomposition Accuracy

This section will compare the accuracy between the High-Level Model (HLM) and the EVD hardware module and compare these values against the built-in function in MATLAB. Matrix A in Equation 4.2 is used for the comparison. The EVD module and HLM are configured with $C_I = W_L = 16$, $J_S = 4$, and $N = 4$.

$$A = \begin{bmatrix} 1000 & 2000 & 1000 & 1000 \\ 2000 & 4000 & 2000 & 2000 \\ 1000 & 2000 & 1000 & 1000 \\ 1000 & 2000 & 1000 & 1000 \end{bmatrix} \tag{4.2}$$

The non-zero eigenvalue of A:

$$\lambda_{\text{MATLAB}} = 7000,\ \lambda_{\text{HLM}} = 7000,\ \lambda_{\text{EVD}} = 7020 \tag{4.3}$$

and its respective eigenvector:

$$v_{\text{MATLAB}} = \begin{bmatrix} 24782 \\ 49565 \\ 24782 \\ 24782 \end{bmatrix}, v_{\text{HLM}} = \begin{bmatrix} 24782 \\ 49565 \\ 24782 \\ 24782 \end{bmatrix}, v_{\text{EVD}} = \begin{bmatrix} 24768 \\ 49563 \\ 24782 \\ 24787 \end{bmatrix} \tag{4.4}$$

The eigenvalues and eigenvectors generated by MATLAB have been converted to an integer for easier comparison. The error caused by the hardware module is calculated by taking the average of the percentage difference using Equation 4.5.

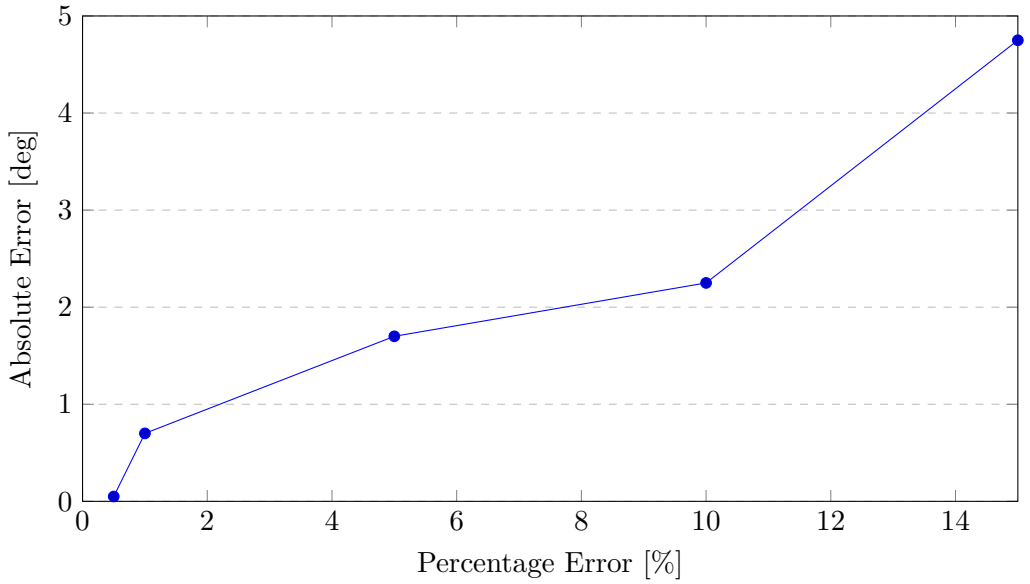
$$Error = \frac{1}{n} \sum \frac{|a - b|}{\frac{1}{2}(a + b)} * 100\% \tag{4.5}$$

Equation 4.6 shows the average error generated by the EVD hardware module is 0.285% for the eigenvalue and 0.569% for the eigenvector.

$$\lambda_{Error} = 0.285\%,\ v_{Error} = 0.569\% \tag{4.6}$$

Figure 4.16 shows how the error in finding the eigenvalue affects the rest of the algorithm. At an error of 1% or less, the AoA has an error of less than 1%. The AoA can achieve an accuracy of 2 degrees until the error is larger than 8%. The figure suggests that the algorithm is robust against error as long as the error in the EVD is not too great.

Figure 4.16: Eigenvalue calculation error effect on the accuracy



Note: $n = 10$, $\Delta\theta_c = 3$, $\Delta\varphi_c = 1$, $\Delta\theta_f = \Delta\varphi_f = 0.1$, $C_I = W_L = 16$ and $J_S = 6$.

Chapter 5

Discussion

5.1 Accuracy

The testing has demonstrated that the MUSIC algorithm can calculate the correct Angle of Arrival with an accuracy of 1° under certain conditions. Some factors that make the algorithm deviate from the desired accuracy are extreme angles, low signal-to-noise ratio, and the number precision utilized. The software implementation of the original complex-valued MUSIC algorithm struggled when the azimuth angle was around 180°, while the real-valued algorithm had issues at an azimuth of 0°. The hardware implementation had two degrees of error at an azimuth of 180° and when the elevation was close to 0° or 90°. The hardware implementation also had other minor errors scattered around extreme elevation angles. Due to time constraints, the Spectral Peak Search (SPS) module designed by Jacob Ranges could not support a search step size smaller than 1 degree. This resulted in the hardware being susceptible to noise and round-off errors. This, in turn, means that the hardware implementation quickly had at least one degree of error as soon as non-ideal conditions were introduced.

Noise and the number of snapshots also had a significant impact on the accuracy of the algorithm. This is as expected since the noise in the samples is amplified by calculating the covariance matrix. The effect of noise can be drastically reduced by increasing the number of snapshots. The best results were found when the number of snapshots where around eight to sixteen. Recall from Section 2.1 that the maximum number of snapshots in one CTE window is three to four, depending on the sampling order used. This is lower than the ideal number of snapshots, meaning the sampling method must be modified to achieve the desired accuracy. One potential solution is to apply a sliding window method that utilizes samples from two separate CTE windows.

The number of bits used in the algorithm had diminishing returns after 18 bits, indicating that this is a good compromise between performance and area. The 16-bit implementation might have a use case if the FPGA is small, but the performance is significantly reduced. In addition, increasing the bit to 18 only increased the utilization by a couple of percentages. The 32-bit implementation had the best performance, but the area requirement would most likely not be worth the increase in area usage.

5.2 Run time and Power Usage

The hardware implementation of the MUSIC algorithm can achieve a significant speedup compared to running on a personal computer. The two-step search method had a limited impact on the calculation time when running in software, but it gave a five times improvement for the hardware. This speedup would, however, significantly increase if the search core had a smaller step size, such as $0.5°$ or $0.1°$. Due to time constraints, the design could not be connected to the software running on the processing system on the Zynq Z2 FPGA. This means that the true run time of the MUSIC algorithm would be higher in the real world. There is, however, a reason to believe that the overhead would be limited as the amount of data transfers is relatively low.

The design can be configured to fit a desired power and speed profile depending on the specific needs of an application. In chapter 4, we saw that a significant amount of dynamic power can be saved by lowering the clock frequency while still meeting the run time deadline of 160 µs. The best trade-off between speedup and power usage seems to be between 25 to 75 MHz. Lowering the clock further significantly impacts the calculation time. Increasing the clock frequency also had a limited impact on the run time. At 50 MHz, the MUSIC core used about 0.6 W compared to 1.3 W at 100 MHz. The personal computer used an average of 60 W of power in comparison.

5.3 Eigenvalue decomposition

The Eigenvalue Decomposition could not be finished within writing the thesis due to the high complexity of the design. Currently, the EVD module can calculate the correct eigenvalue and eigenvectors for 4×4 and 6×6 matrices but not for larger matrices. According to the high-level module, the design should also work for any even symmetrical matrix. There is, therefore, a reason to assume that there is no theoretical reason for the design to be unable to handle larger matrices but rather an issue with the current implementation. The current hypothesis is that there is an issue with the interconnect between PEs. The module can calculate the eigenvector with an average error of 0.5%. This error can be reduced by increasing the number of bits or increasing the number of CORDIC iterations.

From Chapter 4, we saw that a small degree of error in the EVD had little to no effect on the calculated AoA. The signal-to-noise ratio had a much more significant impact on the accuracy than the EVD accuracy. In the architectural exploration phase, the goal was originally to implement a spectral search module with $0.1 - 0.5°$, but this had to be removed due to time constraints. Since the intended step size is much smaller than the step size used in the actual step size, the assumption was that the search would use most of the available calculation time. This is why the parallel Jacobi method was selected, but in retrospect, the serial method would likely achieve the desired performance. The serial method also uses much less area than the parallel implementation, which might benefit some applications.

Chapter 6

Conclusion

This thesis has researched the feasibility of achieving real-time processing of the MUSIC algorithm by accelerating it on an FPGA. The results have demonstrated that the MUSIC core can calculate the AoA in approximately 6 µs at 100 MHz, which is four orders of magnitude faster than the software implementation. The equivalent Python and MATLAB programs had an average run time of 112 ms and 22 ms, respectively. Creating an external interface connecting the core to the locator board would be ideal in the future. This would determine the time it takes from sampling the data to the calculated AoA. Combining the core with the locator board would answer the question of how much overhead is needed for data transfer and how close the design is to reaching real-time performance.

The MUSIC core can achieve an accuracy of 1 ° under certain conditions, which is not as accurate as the software implementation, which estimated the correct AoA down to an error of 0.1 ° without noise. One reason for the lower accuracy in the hardware implementation is that the search core was not designed to handle search steps lower than 1 °. The accuracy difference between the two implementations did decrease as more noise was added, but still limited to 1 °due to the SPS module. This suggests that the difference is insignificant in the real world, where there will always be noise in data sampling. Both implementations were tested with simulated data and not the samples generated from the locator board, but in the future, it would be desirable to test them both with real-world data.

The EVD module could not be finished due to lack of time. The EVD module could, therefore, not calculate the correct eigenvectors for matrices larger than 6×6, which again meant that the SPS could not calculate the correct AoA. Consequently, this introduces some potential causes of uncertainty regarding the algorithm's accuracy as the SPS module used eigenvectors generated with the High-level model. Still, we can conclude that there is reason to believe that the issue with the EVD module is a fault with the implementation rather than a theoretical limitation. Therefore, future work would include modifying the EVD module to support larger matrices. There is, however, a reason to believe that the synthesis data should be relatively accurate. The design utilizes 61%, 15%, 89%, 90%, and 1.35% of the available look-up tables, registers, DSP slices, block ram, and F7 Muxes on the Zynq Z2, respectively. The total power draw of the hardware design is 1.4 W at 100 MHz and 0.6 W at 50 MHz. This is significantly less than a general-purpose computer that drew roughly 60 W.

Appendix A

AoA Signal Simulation

The Angle of Arrival is given by the elevation θ and azimuth φ of the incoming signal. The spherical coordinates in radians are transformed to Cartesian coordinates using Equation A.1.

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} sin(\theta)cos(\varphi) \\ sin(\theta)sin(\varphi) \end{bmatrix} \tag{A.1}$$

The simulated phase signal is then calculated using the following:

$$\Psi = \pi \begin{bmatrix} x \\ y \end{bmatrix} \tag{A.2}$$

The antenna geometry ρ is the relative location between each antenna element given in $[x, y]$ coordinates. Equation A.3 convert the coordinates into the physical positions η on the real locator board:

$$\eta = \rho \frac{d}{\lambda/2} \tag{A.3}$$

where d is the physical distance between antenna elements and λ is the wavelength of the incoming signal. The samples are then calculated using the following:

$$S = \exp(j\Psi\eta) \tag{A.4}$$

S is the simulated signal given in matrix form. For 12 antenna elements and 1 snapshot S has the size (12,1).

Appendix B

Source Files

The source files related to the thesis can be found on GitHub or in the zip file submitted alongside the thesis. Note that the GitHub repository will be made private three months after the submission. The authors' names can be found at the top of each file. The project is structured as the following:

```
EmLogic-Students
├── High-Level Model
│   ├── Matlab
│   └── Python
└── HDL
    ├── Constraints
    ├── Sources
    │   ├── COMMON
    │   ├── CMC
    │   ├── RVT
    │   ├── EVD
    │   │   └── CORDIC
    │   └── SPS
    └── Testbenches
```

Bibliography

[1] N. Shiode, C. Li, M. Batty, P. Longley, and D. Maguire, "The Impact and Penetration of Location-Based Services." University College London, May 2002.

[2] N. El-Sheimy and Y. Li, "Indoor navigation: state of the art and future trends," *Satellite Navigation*, vol. 2, no. 1, p. 7, May 2021.

[3] J. Urena, "Reduction of Ultrasonic Indoor Localization Infrastructure based on the use of Graph Information." IEEE, Aug. 2016, publication Title: International Conference on Indoor Positioning and Indoor Navigation, IPIN 2016, Alcala de Henares, Spain, October 4-7, 2016.

[4] B. Molina, E. Olivares, C. E. Palau, and M. Esteve, "A Multimodal Fingerprint-Based Indoor Positioning System for Airports," *IEEE Access*, vol. 6, pp. 10 092–10 106, 2018.

[5] J. A. Rangnes, "Hardware Acceleration for Real-Time Angle of Arrival Positioning," NTNU, Trondheim, Master thesis, Jun. 2023.

[6] "Indoor Location/ Positioning Market Report- Industry Insights 2024." [Online]. Available: https://www.goldsteinresearch.com/report/global-indoor-positioning-and-indoor-navigation-ipin-market-outlook-2024-global-opportunity-and-demand-analysis-market-forecast-2016-2024

[7] H. Li, H. Lu, L. Shou, G. Chen, and K. Chen, "In Search of Indoor Dense Regions: An Approach Using Indoor Positioning Data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 8, pp. 1481–1495, Aug. 2018, conference Name: IEEE Transactions on Knowledge and Data Engineering.

[8] "Bluetooth Direction Finding: A Technical Overview," Mar. 2019. [Online]. Available: https://www.bluetooth.com/bluetooth-resources/bluetooth-direction-finding/

[9] T. A. Opstad, "Development of a Indoor Bluetooth Tracking Tag," Project Thesis, NTNU, Trondheim, Dec. 2022.

[10] J. A. Rangnes, "Implementation of a real time location system using Bluetooth Low Energy," Project Thesis, NTNU, Trondheim, Dec. 2022.

[11] R. Schmidt, "Multiple emitter location and signal parameter estimation," *IEEE Transactions on Antennas and Propagation*, vol. 34, no. 3, pp. 276–280, Mar. 1986, conference Name: IEEE Transactions on Antennas and Propagation.

[12] A. Paulraj, R. Roy, and T. Kailath, "Estimation Of Signal Parameters Via Rotational Invariance Techniques- Esprit," in *Nineteeth Asilomar Conference on Circuits, Systems and Computers, 1985.*, Nov. 1985, pp. 83–89, iSSN: 1058-6393.

[13] "ISP1907-AOA-DK." [Online]. Available: https://www.insightsip.com/fichiers_insightsip/pdf/ble/ISP1907/isp_aoa_AN210401.pdf

[14] H. Krim and M. Viberg, "Two decades of array signal processing research: the parametric approach," *IEEE Signal Processing Magazine*, vol. 13, no. 4, pp. 67–94, Jul. 1996, conference Name: IEEE Signal Processing Magazine.

[15] G. C. Bagley, "Introduction to Adaptive Arrays. R. A. Monzingo and T. W. Miller. John Wiley, Chichester. 1980. 541 pp. Illustrated. Â£19.15." *The Aeronautical Journal*, vol. 85, no. 847, pp. 349–349, Sep. 1981, publisher: Cambridge University Press.

[16] M. Kim, K. Ichige, and H. Arai, "Implementation of FPGA based fast DOA estimator using unitary MUSIC algorithm [cellular wireless base station applications]," in *2003 IEEE 58th Vehicular Technology Conference. VTC 2003-Fall (IEEE Cat. No.03CH37484)*, vol. 1, Oct. 2003, pp. 213–217 Vol.1, iSSN: 1090-3038.

[17] K.-C. Huarng and C.-C. Yeh, "A unitary transformation method for angle-of-arrival estimation," *IEEE Transactions on Signal Processing*, vol. 39, no. 4, pp. 975–977, Apr. 1991, conference Name: IEEE Transactions on Signal Processing.

[18] A. Lopez-Parrado and J. Velasco-Medina, "Efficient systolic architecture for Hermitian eigenvalue problem," in *2012 IEEE 4th Colombian Workshop on Circuits and Systems (CWCAS)*, Nov. 2012, pp. 1–6.

[19] W. Zhang, W. Liu, J. Wang, and S. Wu, "Computationally efficient 2-D DOA estimation for uniform rectangular arrays," *Multidimensional Systems and Signal Processing*, vol. 25, no. 4, pp. 847–857, Oct. 2014.

[20] "Understanding Bluetooth Range." [Online]. Available: https://www.bluetooth.com/learn-about-bluetooth/key-attributes/range/

[21] "ANT-B10 antenna board," Mar. 2022. [Online]. Available: https://www.u-blox.com/en/product/ant-b10-antenna-board

[22] "BG22 Bluetooth Dual Polarized Antenna Array Radio Board - Silicon Labs." [Online]. Available: https://www.silabs.com/development-tools/wireless/bluetooth/bg22-rb4191a-bg22-bluetooth-dual-polarized-antenna-array-radio-board

[23] D. Lay, S. Lay, and J. McDonald, *Linear Algebra and Its Applications, Global Edition*, 5th ed. Pearson, 2016. [Online]. Available: https://www.pearson.com/en-gb/subject-catalog/p/linear-algebra-and-its-applications-global-edition/P200000004712/9781292419046

[24] L. L. Whitcomb, "Notes on Kronecker Products," *Johns Hopkins University*, Mar. 2020. [Online]. Available: https://dscl.lcsr.jhu.edu/wp-content/uploads/2020/03/Notes_on_Kronecker_Products.pdf

[25] "Unitary Matrix - Definition, Formula, Properties, Examples." [Online]. Available: https://www.cuemath.com/algebra/unitary-matrix/

[26] J. G. Proakis and D. K. Manolakis, *Digital Signal Processing (4th Edition).* USA: Prentice-Hall, Inc., Feb. 2006.

[27] J. E. Volder, "The CORDIC Trigonometric Computing Technique," *IRE Transactions on Electronic Computers*, vol. EC-8, no. 3, pp. 330–334, Sep. 1959, conference Name: IRE Transactions on Electronic Computers.

[28] J. S. Walther, "A unified algorithm for elementary functions," in *Proceedings of the May 18-20, 1971, spring joint computer conference*, ser. AFIPS '71 (Spring). New York, NY, USA: Association for Computing Machinery, 1971, pp. 379–385.

[29] Y. Luo, Y. Wang, Y. Ha, Z. Wang, S. Chen, and H. Pan, "Generalized Hyperbolic CORDIC and Its Logarithmic and Exponential Computation With Arbitrary Fixed Base," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. PP, pp. 1–14, Jun. 2019.

[30] J. Valls, M. Kuhlmann, and K. K. Parhi, "Evaluation of CORDIC Algorithms for FPGA Design."

[31] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers," in *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays - FPGA '98.* Monterey, California, United States: ACM Press, 1998, pp. 191–200.

[32] C.-C. Sun, J. Gotze, and G. Jan, "Parallel Jacobi EVD Methods on Integrated Circuits," *VLSI Design*, vol. 2014, Jul. 2014.

[33] H. Jeffreys and B. Jeffreys, *Methods of Mathematical Physics*, 3rd ed. Cambridge Mathematical Library, Jan. 2000.

[34] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing.* USA: Cambridge University Press, 1992.

[35] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in Fortran 77: The Art of Scientific Computing*, 2nd ed. Cambridge England ; New York: Cambridge University Press, Sep. 1992.

[36] A. Ahmedsaid, A. Amira, and A. Bouridane, "Improved SVD systolic array and implementation on FPGA," in *Proceedings. 2003 IEEE International Conference on Field-Programmable Technology (FPT) (IEEE Cat. No.03EX798)*, Dec. 2003, pp. 35–42.

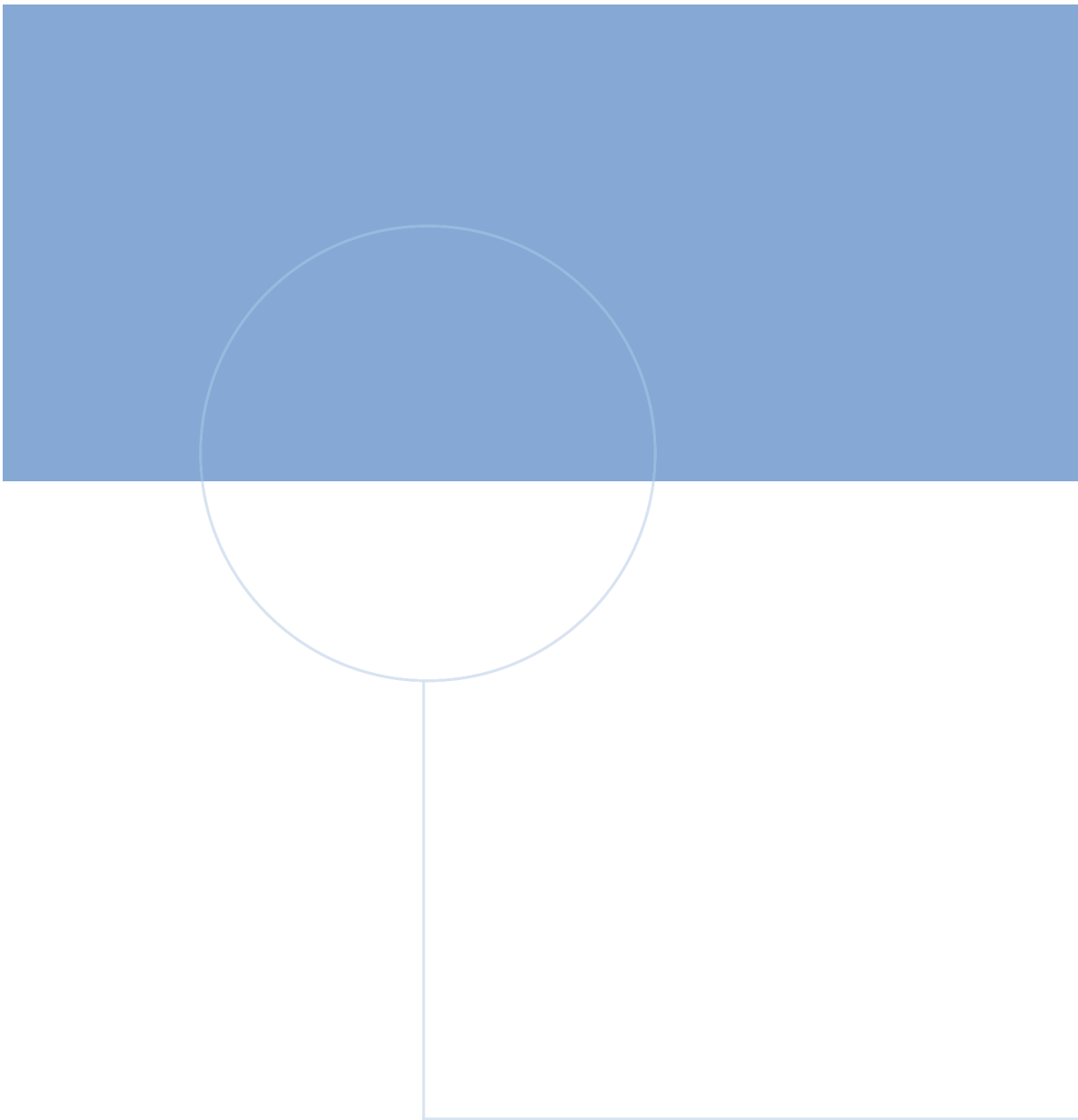
[37] "Vivado ML Overview." [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html

[38] "XUP PYNQ-Z2." [Online]. Available: https://www.xilinx.com/support/university/xup-boards/XUPPYNQ-Z2.html

[39] "7 Series DSP48E1 Slice," *Xilinx*, Feb. 2018. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug479_7Series_DSP48E1

[40] L. Liu, Y. Cao, and M. Guo, "FPGA Implementation of DOA Estimation Method Based on Polarization Sensitive Array," In Review, preprint, Jun. 2022.

[41] A. Karatsuba and Y. Ofman, "Multiplication of Multidigit Numbers on Automata," *Soviet physics. Doklady*, 1963. [Online]. Available: https://www.semanticscholar.org/paper/Multiplication-of-Multidigit-Numbers-on-Automata-Karatsuba-Ofman/bd6bd9ad3c6887bb7da4d11aa49222fc179a8231



NTNU

Norwegian University of
Science and Technology