Faculty of Electrical Engineering,
Mathematics & Computer Science

Hardware/software co-design of an RFID signal processing system

Mika Uytdewilligen
M.Sc. Thesis
July 2021

Supervisors:
Dr.ir. A.B.J. Kokkeler
Dr. S. Safapourhajari
Dr.ir. N. Alachiotis
Dr.ir. M.J.J.van Megen
Dr. P.J. Compaijen

Radio Systems Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

This thesis will investigate the realisation of an RFID signal processing system to determine an RFID tag's location using the MUSIC algorithm. The RFID signal processing system can be a replacement for Electronic Article Surveillance for retail at the entrance of stores. The optimal implementation of the MUSIC algorithm on an embedded system using hardware and software co-design is the main goal for this thesis. A System-On-Chip (SoC) is used to implement the processing blocks of the MUSIC algorithm: covariance matrix calculation, eigendecomposition and localization. These processing blocks are implemented on the processing platform that is most well suited for the timing and resource requirements. The platforms chosen are an FPGA for the covariance matrix calculation to adhere to the hard deadline of the input processing, and an ARM CPU for eigendecomposition and localization for the usage of the integrated Floating Point Unit(FPU) and the appropriateness to use a high level programming language to reduce the development time. The implementation satisfies the requirements of a total processing chain latency of 0.5 seconds, has an position estimation accuracy of 8 cm and is implemented within the available resources.

Contents

1 Introduction **7**
1.1 Introduction . . . 7
1.1.1 Constraints of the design . . . 8
1.1.2 Hardware platform . . . 10
1.1.3 Things to take into account . . . 11
1.2 Related work . . . 12
1.3 Research questions . . . 12
1.4 Structure of thesis . . . 12

2 Theory **14**
2.1 Covariance . . . 20
2.2 Eigendecomposition . . . 22
2.3 Localization . . . 23

3 Platform **26**
3.1 FPGA . . . 26
3.2 Processors . . . 27

4 Design **29**
4.1 Covariance . . . 29
4.2 Eigendecomposition . . . 30
4.3 Localization . . . 31

5 Implementation **33**
5.1 Covariance . . . 33
5.1.1 Structural process . . . 34
5.1.2 Partial sums . . . 35
5.1.3 Clock frequency . . . 35
5.2 Eigendecomposition . . . 42
5.2.1 CPU . . . 42
5.3 Localization . . . 43
5.3.1 CPU . . . 43

6 Analysis of the solutions **44**
6.1 Covariance . . . 44
6.2 Eigendecomposition . . . 45
6.3 Localization . . . 46
6.4 Complete signal processing chain . . . 47

7 Results and discussion **50**
7.1 Covariance . . . 50
7.1.1 Results . . . 50
7.1.2 Discussion . . . 51
7.2 Eigendecomposition . . . 51
7.2.1 Results . . . 51
7.2.2 Discussion . . . 52
7.3 Localization . . . 53
7.3.1 Results . . . 53
7.3.2 Discussion . . . 55

8 Conclusion **56**
8.1 Recommendations . . . 58
8.1.1 ADC number of bits . . . 58
8.1.2 Throughput . . . 58
8.1.3 Amount of antennas . . . 58

A Behavioral process **62**

B Eigendecomposition **63**

List of Figures

1.1 The detection area with latency constraint 9
1.2 The existing hardware platform 11

2.1 The receiving antenna array with the angle of arrival of the signal 15
2.2 IQ demodulation of the antenna signal 15
2.3 Multiple antenna array setup with IQ demodulation 16
2.4 IQ scatter plot of 4 channel ADC data [20] 17
2.5 The receiving antenna array for the transmitting source to be located 17
2.6 Simple covariance plots for different covariance values of X and Y variable 18
2.7 Simple visualisation of the result of the eigendecomposition of the covariance matrix of figure 2.6 19
2.8 The processing blocks of the MUSIC algorithm 19
2.9 The relation between the receiving antenna array and the covariance matrix 20
2.10 Fundamental calculation blocks of equation 2.5 21
2.11 Covariance matrix format 22
2.12 Visualisation of the steering vector 24
2.13 MUSIC spectrum example 25

4.1 The design to be implemented 29
4.2 The covariance calculation blocks 30

5.1 The processing blocks to be implemented 33
5.2 The covariance processing diagram 34
5.3 The FPGA structural summation with shift register to store the partial sums 34
5.4 The FPGA structural summation of the product with shift register 35
5.5 The implementation with colored dotted boxed around the parts that share the same timing constraints. Teal is per ADC sample, Blue is 78 times per 1024 ADC samples and Red is once per 1024 ADC samples 36
5.6 Illustration of the largest combinatorial path 37
5.7 Covariance calculation block diagram 38

5.8 Bit depth for summation 39
5.9 Bit depth for the product calculation 39
5.10 Bit depth for the covariance calculation 40
5.11 Filling the covariance matrix using the FPGA data by applying the Hermitian conjugate to the upper triangle 42

6.1 Histogram of the eigendecomposition computation times zoomed in 46
6.2 Histogram of the localization computation times 47
6.3 The complete processing chain 48
6.4 Histogram of the completer ARM computation times 49

7.1 MUSIC spectra generated by the different implementations . . . 54

8.1 The complete processing chain 57

A.1 The FPGA behavioral covariance process 62

B.1 Histogram of the eigendecomposition computation times 63

Chapter 1

Introduction

This chapter will give the context for the thesis and introduce the problem that is solved by this thesis.

1.1 Introduction

The problem to be solved in this thesis is hardware and software co-design in an RFID tag signal processing system for an anti-theft system. The system to be designed will locate an RFID tag by processing the response signal of the RFID tag received by an antenna array. To do the localization of the RFID tag the MUSIC(MUltiple SIgnal Classification) algorithm is used [19]. This algorithm uses the correlation between the received signals on the antenna array to determine the angle of arrival of one or more signals. The angle of arrival can be transformed to the location of the RFID tag.

The localization of RFID tags is a method with multiple purposes. It can be applied for store inventory and it could be used for example as an anti-theft measure. For instance, for theft alarms, localization could result in a smarter alarm that does not alarm only if RFID tags are too close to the gate but also takes into account a larger area where the movement could be followed. In case there is a tag with a direction outbound of the store with a certain speed, the alarm could sound before the location where the gates would normally be placed. Angle of arrival estimation is considered as the solution for this localization problem, because it is able to estimate the location of the signal source in a single observation. The angle of arrival method takes snapshots of the antennas at the same time; thus, it has the benefit of having all the antenna data at the exact same point in time.

Previous work on the implementation of the MUSIC algorithm uses one of the possible architectures, hardware or software, for example pure FPGA implementation instead of using a combination of CPU and FPGA [7]. This thesis focusses on the implementation using hardware and software co-design on an embedded system. This allows for selecting the optimal processing architec-

ture for the separate parts of the algorithm according to the characteristics of software and hardware.

1.1.1 Constraints of the design

This hardware and software co-design is subject to four main constraints on the design, in order of importance:

1. Performance
   - Latency
   - Throughput
   - Sample size
2. Agility
   - Fast development iterations
3. Resource utilization
4. Power consumption

These constraints will be handled in more detail in the paragraphs below.

Performance For the system, the latency is the time from receiving the signal to the location determination, throughput how many tags can be processed and the sample size the amount of samples used to calculate the covariance.

Latency: The latency of the system is the time frame within which the tag should be processed and the location should be known. When a thief tries to leave the store there is a 2 meter long detection area, therefore a latency constraint should be set where the (s)he is detected before the thief is outside the store. The constraint is taken to be 0.5 seconds as that is the time when a walking person does not exit the alarm area before the tag is processed. This requires the tag to be located in the alarming area within the time limit of 0.5 seconds. This gives a relevant timing for the alarm to activate and the shop employees to act accordingly as can be seen in figure 1.1.

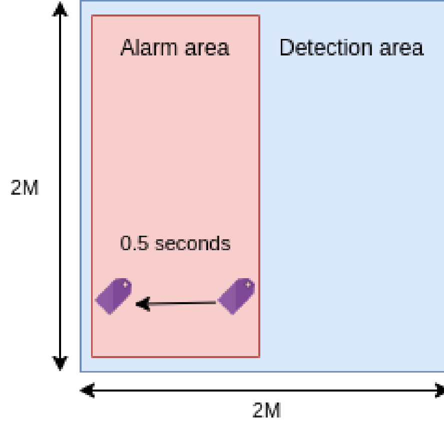


Figure 1.1: The detection area with latency constraint

Throughput: The throughput, which is the amount of tag-reads per second, depends on the RFID reader used in the system. The RFID reader used in this thesis has a throughput of 180 tag reads per second. This dependency is due to the fact that the localization algorithm does not know the Electronic Product Code(EPC) of the tag it is locating. The location returned by the algorithm should be connected to the EPC read by the RFID reader. This results in the throughput constraint to be the throughput of the RFID reader.

Sample size: Sample size of the signal samples is constrained by the time during which the tag is transmitting and the sampling speed of the ADC. The total amount of samples that is possible to measure during the communication between the tag and the reader is constraint by equation 1.1 where the transmission time is determined by equation 1.2 [9]. The transmission time is dependent on the data size to transmit, the data transfer rate, back link frequency of the tag and the Miller number used in the communication protocol between the tag and reader. Where the Miller number is the amount of cycles that are required for every bit transmitted using the Miller-Modulated Subcarrier encoding [9].

$$\text{Amount of samples} = \text{Transmission time} * \text{ADC sampling frequency} \tag{1.1}$$

$$\text{Transmission time} = \frac{EPCsize}{Datarate} \tag{1.2}$$

$$\text{Data rate} = \frac{\text{Back link frequency}}{\text{Miller number}} \tag{1.3}$$

The tag used in this thesis has a back link frequency of 250KHz with a 128 bit EPC, the communication protocol uses 4 as the Miller number. This results in the transmission time of 2.048ms, combining this time with the 3Msps sampling frequency allows for 6144 samples to be taken during the communication between the tag and the receiver.

The sample size used in this report is 1024 since it offers a good trade-off between reduction of the noise effect and the input data size. The value of 1024

is chosen for reducing the effect of outliers in the sample data set and is a power of 2 which is beneficial for the, on hardware implemented, calculations to be done in this thesis.

Agility The implementation of this thesis is a testbed for future products and research, this requires development iterations to be implemented quickly. The final processing steps in the MUSIC algorithm are subject to optimization in the implementation of this thesis.

Resource utilization The resources of the SoC are shared among other modules/processes. This limits the utilization of the resources on the SoC for the implementation of this thesis.

Power consumption Power consumption is a constraint that weighs less but should be considered when making choices for the platform for processing. This is, however, no hard constraint and is considered as an additional benefit. Given two options equal in performance the least power hungry option is preferred.

1.1.2 Hardware platform

This report focusses on the implementation of the MUSIC algorithm on an existing hardware platform, the DE10-standard development kit with a Cyclone V [2], a System on a Chip(SoC) with a Field Programmable Gate Array(FPGA) and an ARM Cortex-A9 32bit dual-core processor [1]. This is the given hardware platform as it is the current measurement setup, the overview is shown in the block diagram of Figure 1.2.

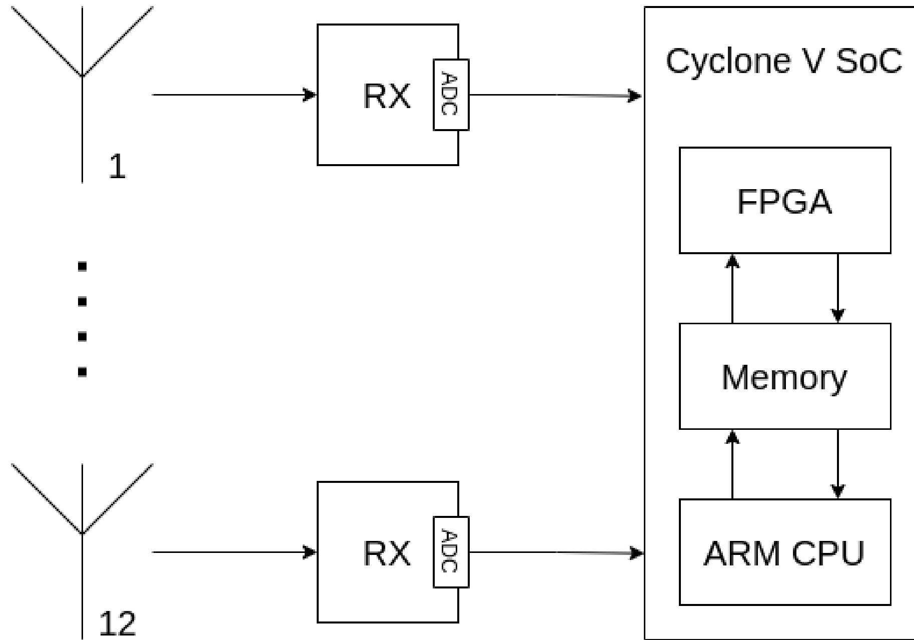


Figure 1.2: The existing hardware platform

Considering the hardware platform used, the DE10-standard with a Cyclone V, two processing categories are available:

- FPGA
- Processor
  - Soft Core on the FPGA (NiosII) [3]
  - ARM Cortex-A9 dual-core [1]

The separate processing platforms will be discussed in depth in chapter 3.

1.1.3 Things to take into account

Large dynamic range of values The calculations have a large dynamic range in the resulting values, this gives rise to a possible problem for implementation on the FPGA as the FPGA does not support native floating point values. This would require to have a fixed point value or have a separate implementation for working with floating point numbers. This is only a problem if fixed point values do not provide sufficient precision or range.

Number of antenna elements Scaleability is not going to be discussed in this work. The maximum amount of antennas that will be considered in this thesis are 12. This is due to the fact that adding more antennas increases the cost such that it would not be feasible in the field of this implementation.

1.2 Related work

There is work already done that is related to this thesis, most notable is the paper on the MUSIC algorithm [19]. This paper introduces the MUSIC algorithm and is used as the basis for the MUSIC algorithm parts in this thesis. The literature also shows the combination of CPU and GPU platforms, where the GPU is used for the highly parallel spectrum calculation and peak search in the spectrum for the many processing cores on the GPU [14]. However, this is not possible in this thesis as the hardware does not include a GPU. Furthermore literature shows implementations purely on the FPGA with low latency implementation of the MUSIC algorithm [24]. There is also a paper that uses Virtual Array Reduction to reduce the complexity of the calculations by limiting the input for the MUSIC algorithm on FPGA [7]. The paper is published close to the end of this thesis and was not used, but is recommended for future research. To the best of our knowledge, the implementation of the MUSIC algorithm on a SoC utilizing an FPGA and a CPU has not been investigated. Therefore, the hardware-software co-design for MUSIC algorithm on a SoC including an FPGA and a CPU is the main contribution of this thesis.

1.3 Research questions

The main research question in this work is as follows.

- How is the MUSIC algorithm implemented with optimal performance on an Embedded system using hardware and software co-design?
  To find the answer for this research question, it is divided into four smaller research questions related to design and implementation of different blocks of the processing chain of the MUSIC algorithm.
  - Which processing platform is most suitable for covariance calculation?
  - Which processing platform is most suitable for eigendecomposition?
  - Which processing platform is most suitable for localization?
  - How do hardware choices propagate in the resource usage of the Embedded system?

The above mentioned questions are investigated in this thesis and the processing chain of the MUSIC algorithm is designed, implemented and evaluated based on the decisions.

1.4 Structure of thesis

Chapter2 will introduce the theoretical background on which this thesis is built to determine the optimal solution. Chapter3 takes the introduced platforms and elaborates on the characteristics of each platform to aid in the choice where to

process the algorithm parts. Chapter4 uses the theory and platform information to provide a design decision for the processing platform for the MUSIC algorithm processing parts. Chapter5 discusses the implementation choices made and how the design decisions are implemented. Chapter6 performs timing analysis on the implemented algorithm to validate if the timing complies to the constraints. Chapter7 checks the output of the implementation to be the desired output. Chapter8 draws the conclusion for this thesis and includes recommendations for future research.

Chapter 2

Theory

As was described in the introduction, this research will focus on finding the optimal implementation of the MUSIC algorithm for Angle of Arrival estimation on an embedded system. This chapter will provide the theoretical background regarding angle-of-arrival determination and the MUSIC algorithm specifically [19].

As mentioned in section 1.1, the problem to be solved by the MUSIC algorithm is the localization of a radio transmitter, in the case of this project a replying RFID tag. The MUSIC algorithm is used for radio direction finding, which makes it an ideal algorithm for solving our problem. If the hight of the tag is known and fixed the location of the replying tag can be estimated using the direction of the incoming signal. When multiple antennas are located at different distances from a transmitter, a delay in the received signal can be seen between the closest and the more distant antennas. This information can be used to find the direction of the transmitting source.

In the simple 2D illustration, figure 2.1, at a point in time the phase between the incoming radio wave and the antenna array is dependent on the location of the transmitter. This allows the system to determine the delay of the signal arriving at each antenna as shown in Figure 2.1 where the $d\ sin(\theta)$ indicates the delay of the signal arriving at antenna 2 compared to antenna 1. The θ in the $d\ sin(\theta)$ is the angle-of-arrival of the signal.

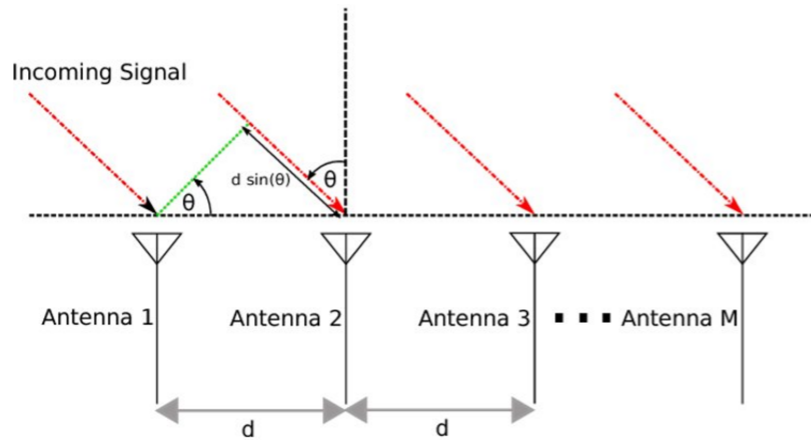


Figure 2.1: The receiving antenna array with the angle of arrival of the signal

The signal which is received by the antenna passes through a IQ-demodulator to retrieve the In-phase and Quadrature data. This IQ demodulation is shown in Figure 2.2, where the RF signal is split in the separate IQ data. The plot shows two data points, this is because the fact the antenna switches between two antenna states as the data transmitted is binary. This switching between antenna states, '1' and '0', appears as 2 positions in the IQ plot of Figure 2.2.

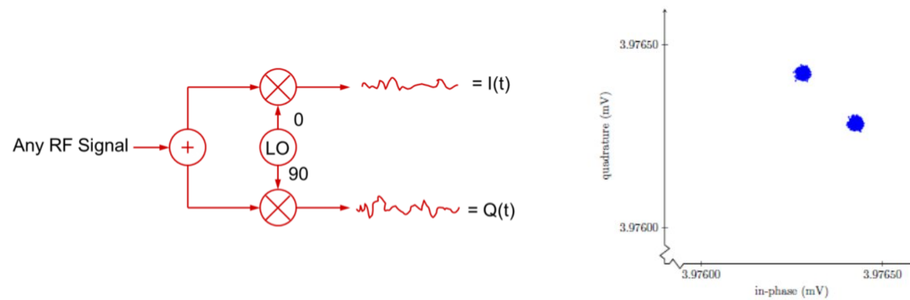


Figure 2.2: IQ demodulation of the antenna signal

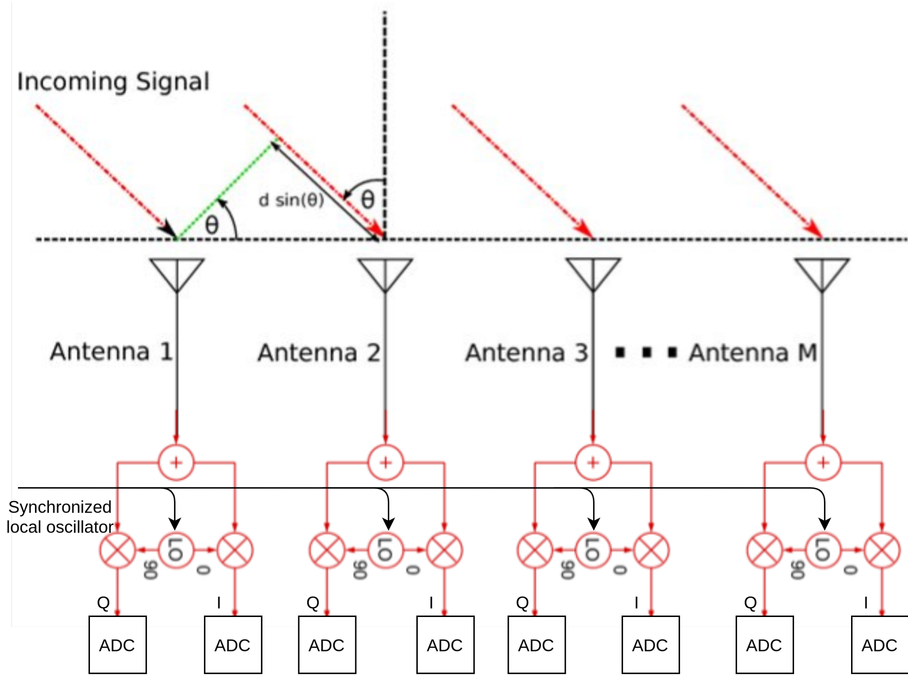


Figure 2.3: Multiple antenna array setup with IQ demodulation

For the multi antenna array setup it is crucial to synchronise the local oscillators to be certain a snapshot of the IQ signal of each of the antennas is taken at the exact same time, shown in Figure 2.3. This is crucial for the delay in signal arriving at the antennas to be determined from the data snapshot.The data received by the different antennas (channels) after IQ demodulation can be shown in a scatter plot, the scattered data can be seen in figure 2.4.

The MUSIC algorithm, 2D MUSIC, elaborated above is for the far field scenario. However, this is not always the case for this implementation as the far field approach is only valid from a certain distance from the antenna. This distance is the Fraunhofer distance calculated using equation 2.1, where the D is the length of the antenna array [6].

$$d_F = \frac{2D^2}{\lambda} \tag{2.1}$$

The length of the antenna array in this thesis is 0.4 m and the wavelength is 0.32577284 m, wavelength of 920.25MHz resulting in a Fraunhofer distance of $\frac{2*0.4^2}{0.32577284} = 0.98m$. All tag reads within 0.98m of the receiving antenna array are in the near field of the antenna array. When the near field is used for the MUSIC algorithm the situation changes from a planar wave to a wave front that has a curvature, as shown in Figure 2.5,and requires a different formula to determine the expected delay between the antennas.

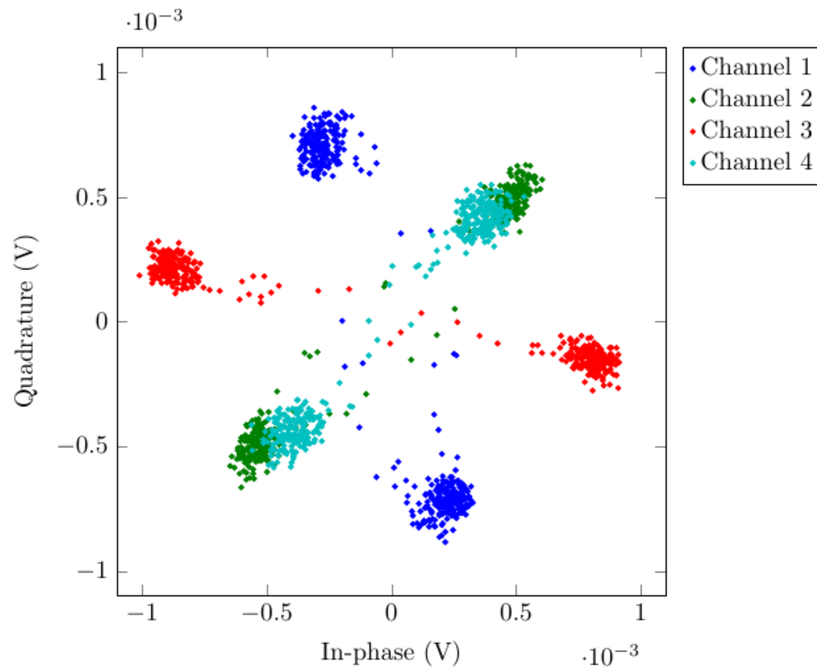


Figure 2.4: IQ scatter plot of 4 channel ADC data [20]

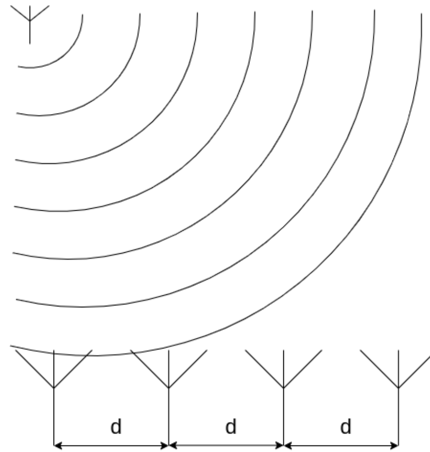


Figure 2.5: The receiving antenna array for the transmitting source to be located

When observing figure 2.4 the phase difference between the different channels is present as rotation in the IQ plot. To extract this rotation angle between the

data channels, the covariance, of the received signals of the antennas, is used. This is done by comparing the signals sampled by the different antennas in the antenna array in relation to each of the other antennas in the antenna array. A simple example is given now as the complex valued and multi dimensional nature of the antenna data is too complex to be presented in this thesis. The covariance of two simple variables can be shown as the shape of a cloud of data points where a data point is a certain input for both variables and the result is plotted versus two different axis (X and Y) see figure 2.6.

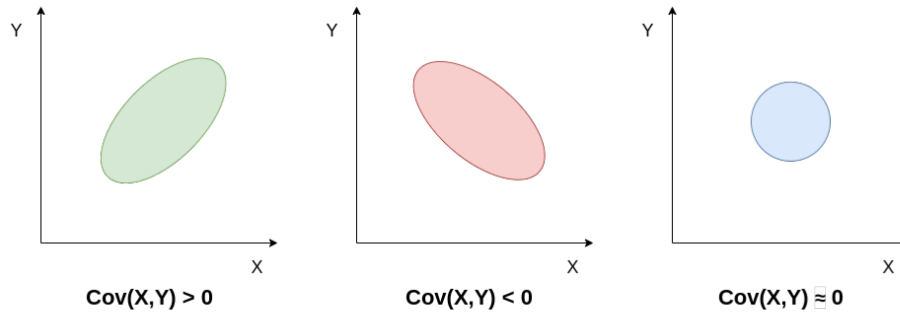


Figure 2.6: Simple covariance plots for different covariance values of X and Y variable

The value of the covariance shows the relation between the two different signals, figure 2.6. Positive values for the covariance show a relation in the same direction, if the value of signal 1 increases, the value of signal 2 increases as well. Negative values for the covariance result in an opposite direction of change of the two signals, when the value of signal 1 increases the value of signal 2 decreases. The opposite is also true i.e. when the value of signal 1 decreases the value of signal 2 increases. This shows the important role of the sign of the covariance value. As can be seen the main relationship between the signals is determined by the sign of the covariance value. If the covariance is zero the two signals are varying with no relation, between the two signals.
Using the covariance matrix of the received data, the next step is to extract the direction of the correlated data change received by the antenna array. The covariance matrix is determined by the shape of the cloud of data points (figure 2.6), as it depends on the correlation between the signals. To extract this data from the covariance matrix, an eigendecomposition is used. The result of the eigendecomposition provides us with the eigenvalues and eigenvectors of the covariance matrix. When applying the eigendecomposition to the covariance matrix of figure 2.6 we can plot the resulting eigenvectors in the same plot resulting in figure 2.7

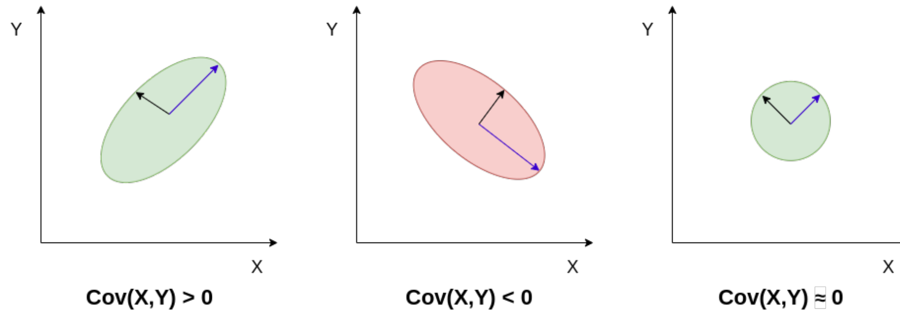


Figure 2.7: Simple visualisation of the result of the eigendecomposition of the covariance matrix of figure 2.6

From figure 2.7, one of the most clear observations is the direction of the largest eigenvector and the second largest eigenvector. The largest eigenvector is in the direction of the relation between the signals which were used to calculated the covariance. The second largest vector is in the direction of the noise of the data set which was used to calculate the covariance. When the covariance is zero, no location can be detected as there is no correlation between the received signals of the antennas. The MUSIC algorithm is a subspace based algorithm [15]. When the eigenspace is calculated the next step of for the MUSIC algorithm is to split the eigenspace into two subspaces, the signal and noise subspaces. To determine the location of the tag steering vectors are required for all the possible locations the tag can be within the detection area. Using the steering vectors and the noise space, the location of the tag can be determined. Because the matrix is hermitian the eigenspace is orthogonal, all vectors are orthogonal to each other. This property of the eigenspace is used to calculate the matching grade of the steering vector. The steering vector that is most orthogonal to the noise space is the location that matches best with the signal space. This matching for each steering vector in the detection area creates a MUSIC spectrum that indicates the matching grade of the location to the signal subspace. This gives the possible signal transmission locations that span the signal space.
All the above mentioned steps result the MUSIC algorithm to consist of the the 3 processing blocks including covariance, eigendecomposition and localization shown in Figure 2.8, which are discussed in the following sections.

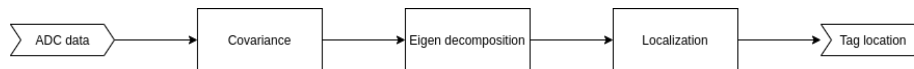


Figure 2.8: The processing blocks of the MUSIC algorithm

2.1 Covariance

Covariance is a measure of the correlation of changes in one variable with respect to another variable. In the case of this implementation, the correlation to be calculated is the correlation between antenna signals received by the antenna array to determine the direction the signal is transmitted from. This process can be seen in figure 2.9, where the antenna array of 3 antennas results in a covariance matrix of size $3 * 3$ with one axis the antenna signals and the other axis the complex conjugate of the antenna signals.

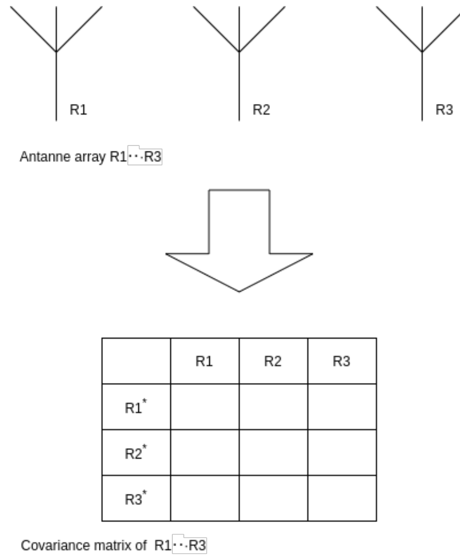


Figure 2.9: The relation between the receiving antenna array and the covariance matrix

The covariance between signal 1 and signal 2 is calculated using the formula in equation 2.2 [16]. The expected or mean value, $E[\ ]$, of the sample set which needed to calculate the covariance can be obtained from equation 2.3.

$$\begin{aligned} cov(S_1, S_2) &= E[(S_1 - E[S_1])(S_2^* - E[S_2^*])] \\ &= E[S_1 S_2^* - S_1 E[S_2^*] - E[S_1] S_2^* + E[S_1] E[S_2^*]] \end{aligned} \tag{2.2}$$

$$E[S_1] = \frac{\sum S_1}{\text{number of samples(N)}} \tag{2.3}$$

Since the inputs of the covariance calculation are discrete samples, the result of equation 2.2 becomes equation 2.4. This is done as the sample mean is used for the expected mean value which requires the Bessel's correction to be applied [11].

$$cov(S_1, S_2) = \frac{1}{N-1} \Sigma[S_1 S_2^* - S_1 E[S_2^*] - E[S_1] S_2^* + E[S_1] E[S_2^*]] \tag{2.4}$$

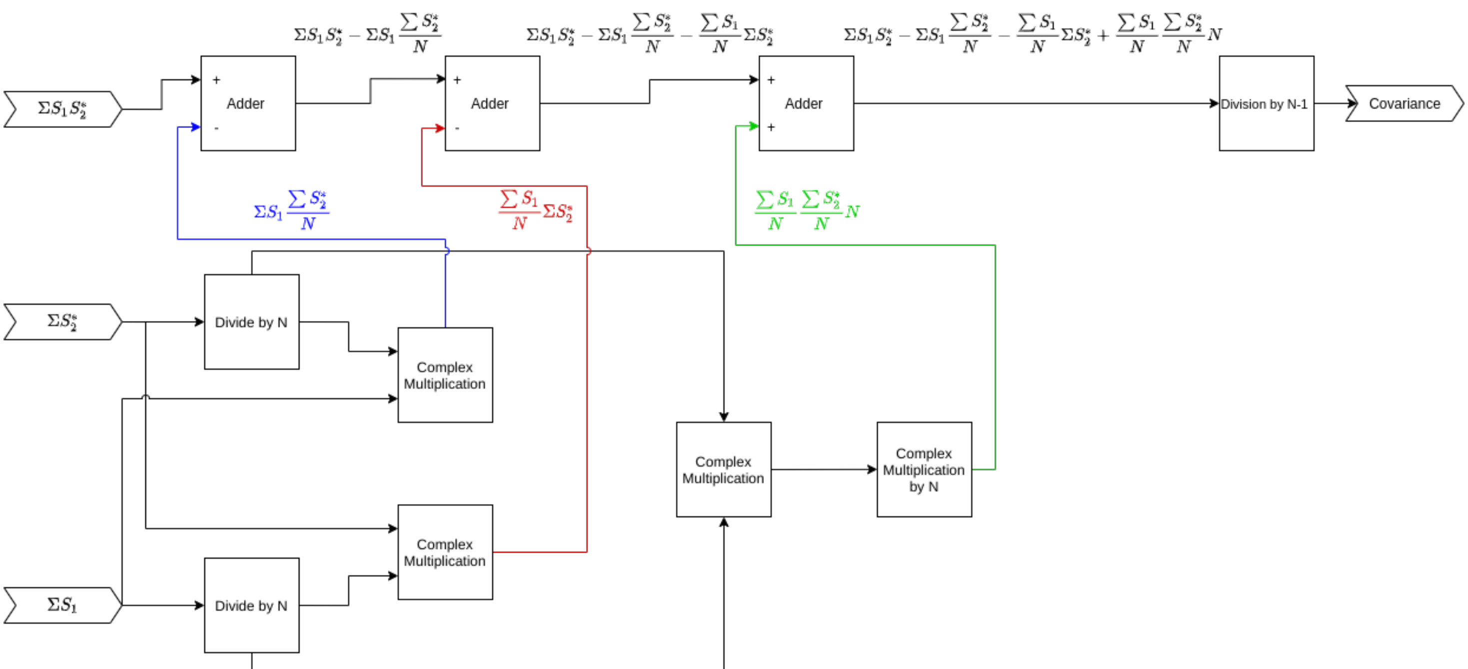


Figure 2.10: Fundamental calculation blocks of equation 2.5

To apply these calculations in an algorithmic format we can combine equations 2.4 and 2.3 into equation 2.5.

$$cov(S_1, S_2) = \frac{\sum(S_1 S_2^* - S_1 \frac{\sum S_2^*}{N} - \frac{\sum S_1}{N} S_2^* + \frac{\sum S_1}{N} \frac{\sum S_2^*}{N} N)}{N-1} \tag{2.5}$$

Taking equation 2.5 and breaking it into its fundamental calculation block we get figure 2.10.

Antenna number

Antenna number	1	2	3	4	5	6	7	8	9	10	11	12
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												

Figure 2.11: Covariance matrix format

The result of applying equation 2.5 to the data set received by the antenna setup is a matrix that shows the correlation between the antenna outputs. The covariance matrix is shown in figure 2.11, where in each cell of the matrix equation 2.5 is performed between the antenna on the column and the row. The shape of this matrix is square with the size $12 * 12$ since 12 antennas are used in this thesis. This matrix contains all the information required to determine the location of the tag.

2.2 Eigendecomposition

After having calculated the covariance matrix the next step in the music algorithm is using the covariance matrix and performing eigendecomposition on it. The eigendecomposition is a method of taking a diagonalizable matrix and decomposing the matrix into terms of its eigenvalues and eigenvectors. Where mathematically the transformation is expressed by $A\mathbf{x} = \lambda\mathbf{x}$ where λ are the eigenvalues, $\mathbf{x}$ the eigenvectors and A the input matrix. The eigendecomposition provides a space spanned by the eigenvectors, this is required for MUSIC as MUSIC is a subspace based algorithm [15]. The space resulting from the eigendecomposition is split into a signal space and a noise space. From a

physics perspective the eigenvalues represent the RMS values of the corresponding eigenvectors. The eigenvalues are used to split the space into the signal and noise space. This is possible since there is a difference in the order of magnitude of eigenvalues of the noise space and the signal space. The large eigenvalues correspond to the signal space and the small magnitude correspond to the noise space.
To calculate the eigenvalues and eigenvectors of the matrix, multiple algorithms exist. In this thesis the divide and conquer algorithm will be used as it is a common algorithm used in libraries e.g. Armadillo [18] [17].

The divide and conquer algorithm is based on the idea of splitting a larger problem into smaller sub problems and recombining the results for a faster calculation of the eigendecomposition [8]. For the divide and conquer algorithm to be used the input matrix first has to be reduced to a symmetric tridiagonal form, Equation 2.6 . This is done through orthogonal transformations.

$$T = \begin{pmatrix} a_1 & b_1 & & 0 \\ c_1 & \ddots & \ddots & \\ & \ddots & \ddots & b_{n-1} \\ 0 & & c_{n-1} & a_n \end{pmatrix} \tag{2.6}$$

Once the matrix is in a tridiagonal form, it can be divided in, usually 2, sub matrices. From this point the eigenvalues should be calculated for these sub matrices. When implementing this functionality it is done by recursive calls to the divide and conquer algorithm. If the sub matrix is sufficiently small, the QR algorithm can be called to calculate the eigenvalues and eigenvectors.
The QR algorithm is an iterative algorithm that decomposes an input matrix A_k into Q_k and R_k where is R_k is a right upper triangular matrix [12] [10]. The QR algorithm then creates A_{k+1} by using equation 2.7.

$$\begin{aligned} A_k &= Q_k R_k \\ A_{k+1} &= R_k Q_k \end{aligned} \tag{2.7}$$

Performing this operation iteratively results in matrix A_{final} which is converging to a diagonal matrix with the eigenvalues of matrix A. The matrix U_k converges to the matrix of eigenvectors, where $U_k = Q_0 Q_1 \cdots Q_k$.

2.3 Localization

Finally the last step of the music algorithm is localization. Localization uses the eigenvectors and eigenvalues obtained from the eigendecomposition and determines the location RFID tag. This is done by first generating a set of steering vectors which are a set of vectors that contain a given set of possible locations. The elements of the steering vector are the complex signals received by the specific antennas as can be seen in figure 2.12.

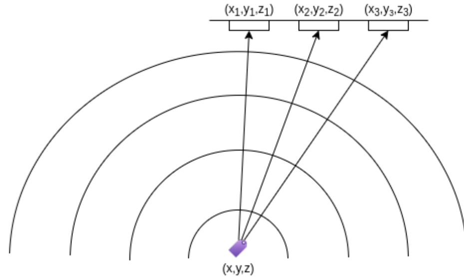


Figure 2.12: Visualisation of the steering vector

These steering vectors can then be used to determine, using the noise space of the eigenvectors, where the signal from the tag is located in the 3 dimensional space. When comparing the localization to the version from the "Multiple Emitter Location and Signal Parameter Estimation" paper [19] the main difference is that this thesis does not use the angle but the (x, y, z) coordinates of the location. The first step is to calculate the phase of the signal arriving at the antenna element. This is done using distance from the tag to the receiving antenna element(eq:2.8) and the wavelength for the specific frequency band. This is combined into equation 2.9, where $a(x, y, z)$ is the resulting steering vector for the location.

$$r_1 = \sqrt{(x_1, y_1, z_1)^2 - (x, y, z)^2} \tag{2.8}$$

$$a(x, y, z) = \begin{bmatrix} e^{(-2j*\pi*\frac{r_1}{wavelength})} \\ \vdots \\ e^{(-2j*\pi*\frac{r_{12}}{wavelength})} \end{bmatrix} \tag{2.9}$$

Using the steering vector, $a(x, y, z)$, the matching grade of the signal to steering vector can be calculated using equation 2.10 [19].

$$P_{mu}(x, y, z) = \frac{1}{a^*(x, y, z) E_N E_N^* a(x, y, z)} \tag{2.10}$$

When for all the X and Y position at a fixed Z the P_{mu} is calculated the MUSIC spectrum can be shown as shown in figure 2.13. Where the axis span the raster of the area for which steering vectors are generated and the colors show the matching grade of the steering vector to the signal space.

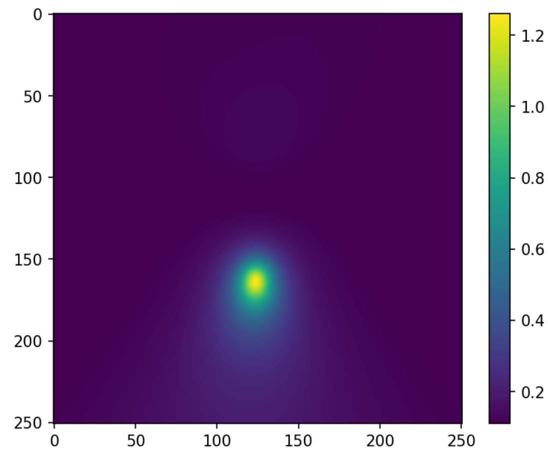


Figure 2.13: MUSIC spectrum example

Observing the MUSIC spectrum, figure 2.13, the estimated location of the signal origin can be determined which is shown by the yellow peak (the highest value in the spectrum). The spectrum in the case of figure 2.13 is clear and singular. However, since the MUSIC algorithm is designed for estimation of multiple emitters, the spectrum can show multiple and different peaks in the spectrum. This is even possible with a single signal due to the possibility of reflections, which will have a different direction of arrival and/or different travelled path length to the original signal.

Chapter 3

Platform

This Chapter will give the reader an insight into the strengths and weaknesses of the available processing platforms.

In chapter 1.1.2 three platforms were mentioned: the FPGA, the ARM core and the NiosII softcore. FPGA requires a hardware description language to program and programs all the required logic elements and wires them together as hardware. The NiosII and ARM are both referred to as processors. These can both be programmed in the middle-level programming language C. The ARM Cortex-A9 processor supports an operating system(OS) which makes it possible to use other middle level and higher-level languages while programming the application.

The performance of the ARM and NiosII processors is measured in Dmips/ MHz/ core. Where Dmips stands for Dhrystone million instructions per second and Dhrystone is a synthetic computing benchmark containing no floating-point operations.

The system introduced in section 1.1 has the ADC's connected to the FPGA and the final tag location is to be presented to the other parts of the eco system on the ARM processor.

3.1 FPGA

An FPGA is a processing platform that has the advantage of being significantly faster in some applications due to its parallel nature and the ability to optimize the number of gates used for certain processes. This property is very advantageous in the implementation case at hand. The data with high throughput has to be highly parallel so that the system complies with the constraints set in chapter 1.1.1.
The complexity of using an FPGA is in the programming. The FPGA programming language is a hardware description language(HDL), which requires more development time to program algorithms when compared to middle and high level languages that are used by most processors. Using a hardware description

language, requires a more in-depth understanding of the algorithm and the ability to translate that understanding to a hardware implementation of the desired algorithm.

Performance The resources of the FPGA [2] are:

- 25K logic elements or 9,430 Adaptive Logic Modules (ALMs)
- 400 MHz max clock frequency
- 36 DSP blocks

The performance of the FPGA is deterministic, it has no background running tasks or other processes that can change the timing of the algorithm processing. What is implemented on the FPGA is executed at the same time every single execution as the algorithm is implemented as hardware. This deterministic property is exploited to have a low and fixed latency from data acquisition to processing.

3.2 Processors

The C programming language is more suited for algorithmic programming than a HDL, as it allows the programmer to take a more mathematical approach. This flexibility and less complexity in programming is generally preferred over the complexity of the HDL if the resources and performance requirements allow the use of processors.

It is worth noting that the execution time of algorithms on a processor is variable due to background processes and other overhead in the system if an operating system is used to manage multiple running programs. However, if the program is running on a more real-time operating system or bare metal, it becomes increasingly deterministic and fixed execution times arise. However, for this project, we assume the scenario with an operating system running on the ARM with multiple other background tasks as this is a given for this thesis due to other modules are already present on the OS.

NiosII

The NiosII softcore is a processor that runs on the FPGA. This is an implementation provided by Intel. The benefit of the softcore is that it can run on the FPGA and have fast and easy interfacing with the FPGA for accelerating algorithms. This benefit is further emphasized by the 256 custom instructions that can be implemented leveraging the use of the FPGA and VHDL code resulting in an FPGA accelerated processing core.

Implementation The NiosII is implemented by using logic elements, 600 logic elements for the economic NiosII and between 3k and 4k logic elements for the fast NiosII. [3] The Cyclone V SoC used in this thesis contains 25k logic elements [2], this makes the utilization of the logic elements by the NiosII range from 2.4% to 16% of the total amount of logic elements on the SoC.

Performance The performance of the NiosII is limited by the clock speed(400 MHz) of the FPGA and the implementation on the FPGA resulting in 0.753 Dmips/ MHz/ core for the fast NiosII and 0.107 Dmips/ MHz/ core for the economic NiosII. [3]

ARM Cortex-A9

The ARM platform is a standard 32bit processor running on the RISC instruction set and is capable of doing floating-point calculations with a floating-point unit.The ARM Cortex-A9 dual-core has an performance of 2.50 Dmips/ MHz/ core and is running at 925MHz with 2 cores. [1]

Choice of processor

Considering both the NiosII and the ARM it can be seen that the performance of the NiosII(0.753 Dmips/ MHz/ core) is significantly less than the performance of the ARM processor(2.50 Dmips/ MHz/ core). Even considering the benefit of the custom instructions, it is not feasible to use the NiosII instead of the ARM as the ARM is capable of communication/sharing data with the FPGA (it is connected on the SoC) and hence is able, albeit in a less integrated way, to use the FPGA for hardware accelerating certain processing tasks. Due to the reasons mentioned above the NiosII is dismissed for the current application as it is not advantageous when the ARM processor is available.

Chapter 4

Design

The design chapter will combine the knowledge of chapters 2 and 3 into design decisions. The design specified in this chapter is implemented and tested in the remainder of this thesis. The block diagram of the MUSIC algorithm discussed in the previous chapters is repeated in figure 4.1.

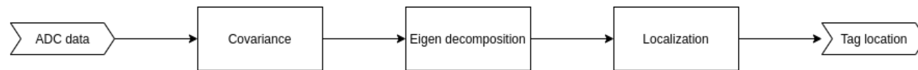


Figure 4.1: The design to be implemented

The requirements for the design are:

- Maximum latency of 0.5 seconds
- Support 12 antennas
- Process the covariance in real-time
- Sample size of 1024

These requirements are the basis for the choices made in this design chapter for the processing blocks in figure 4.1.

4.1 Covariance

In this section, it is assumed that not using floating point precision but staying in integer precision has a negligible impact on the further processing in the processing chain. Which is discussed in more detail in chapter 7, where the implication of this assumption is discussed.The covariance block consists of the sub blocks defined by section 2.1 figure 4.2 shows the composition of these sub blocks with the corresponding rate of new data arriving at the input of the block.

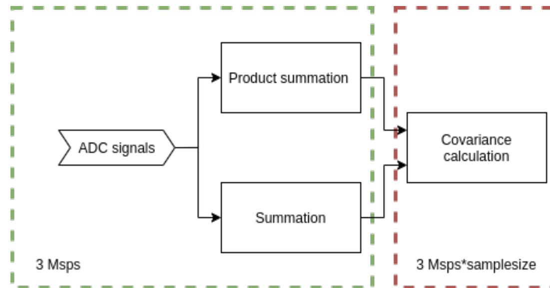


Figure 4.2: The covariance calculation blocks

In section 2.1, it was mentioned that the processing of the covariance calculation has to be real-time as the data is only available for the time between two ADC samples. Figure 4.2 shows the region where this data is used in green with the corresponding data rate . This implies a clear preference for the FPGA platform as it is the only platform in the options that offers true real-time deterministic behavior. CPU processing has the problem of running an OS with multiple other processes, this makes it not possible to guarantee the real-time processing of the data. The usage of an OS is a given for this platform as other modules are already present on the OS. This is problematic since the input data has a crucial latency window in which the data has to be processed before the new input data arrives.
This is why the design decision is made to use the FPGA as the processing platform for the covariance calculation.

4.2 Eigendecomposition

The eigendecomposition is a calculation step in the chain that does not have a hard latency requirement but has a large number of individual operations. This generates a list of the characteristics for the eigendecomposition:

- Soft latency limit
- High number of individual operations

This list favors the CPU implementation. As previously mentioned in chapter 3, the FPGA is highly deterministic which is not a necessity for the eigendecomposition. On the other hand, FPGA processing platform requires every part of the processing chain to be implemented on hardware which is quite detrimental for the resource utilization. This is emphasized by the high utilization of multiplications by the eigendecomposition algorithm. Multiplications are implemented using DSP blocks which are the most limited resource for the FPGA. A significant portion of the resource utilization can be resolved by reusing the individual processing blocks multiple times, this, however, limits the throughput. The implementation of complex valued fixed point eigendecomposition has

high resource usage on the FPGA as paper [4] shows for a 4*4 matrix, where the implementation of this thesis would require 12*12 with larger fixed point numbers. This reduces the feasibility of the eigendecomposition on the FPGA. The CPU does not have these characteristics in the configuration used in this thesis, the CPU is non deterministic due to the operating system running on the processor and the resource utilization is expressed in clock cycles. The non deterministic behavior is no disadvantage in this processing step since the data is present for a longer time and the processing latency can be variable. The benefit of the CPU is in the resource utilization being expressed in clock cycles and not in area as it is for an FPGA. Clock cycles are a generic resource and can be used in all the operations required independent of the type of operations (addition, multiplication, etc...).

The ARM CPU has a Floating Point Unit(FPU), a mathematical coprocessor which is specifically designed for doing mathematical operations on floating point numbers. This is an integrated hardware coprocessor in the ARM CPU, the use of the this FPU does not require additional hardware or large additional software resources. This allows the use the floating point precision required for the eigendecomposition without additional resource usage and makes the CPU the choice for the eigendecomposition processing platform. The CPU also allows the use of pre-existing libraries for the mathematical operations. Which speeds up the development time and introduces more optimized implementation of the mathematical operations.

4.3 Localization

The localization is, as seen in section 2.3, a mathematical operation that relies on floating point precision, multiplications, division and additions. The additions do not impose difficulty on either processing platform, however, the floating point precision and multiplications impose a higher resource usage on the FPGA. This is problematic since the implementation of the covariance relies heavily on the FPGA for the real-time deterministic execution. The covariance calculation used most of the available DSP slices on the FPGA, consuming resource so the localization could not be implemented using DPS slices. The use of DSP slices increases the speed and efficiency of the multiplications on the FPGA, in the case of this thesis the use of complex numbers increases the efficiency of the DSP slices as the available IP blocks prefer the DSP blocks for the implementation. The more problematic mathematical operation for the localization on the FPGA platform is the division by a floating point number. This is not trivially done on the FPGA since the divisor is not a guaranteed power of 2. If the number was a guaranteed power of 2, the division could be done by a bit shift operation, which would lose the decimal numbers. This bitshift operation is a low resource resource operation. The ARM CPU allows, using the FPU, to use the floating point precision required for the Localization.
Combining the above points, the design choice can be made to use the ARM CPU for the localization calculations. The main reason for this is the presence

of the FPU on the ARM CPU which provides the mathematical operations with floating point precision for low additional resource usage. An additional benefit of the ARM platform for the localization is to not have to crossover into the FPGA fabric and return the data to the ARM platform. This removes some complexity and overhead for the implementation.

Chapter 5

Implementation

This chapter will discuss the problems and implications of the different platforms and their implementations

In this chapter the process of implementing the design choices in chapter 4 is documented. This includes the choices how to implement required processing parts, data storage and performance choices. For readability the processing blocks to be implemented are repeated in Figure 5.1

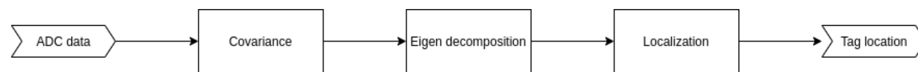


Figure 5.1: The processing blocks to be implemented

5.1 Covariance

This section will elaborate decisions on choices and how the problems related to the implementation of the covariance calculation on the FPGA and CPU platforms were addressed. The covariance system to be implemented is repeated in figure 5.2. Here the green part is where the input is only present for the time between the ADC samples.

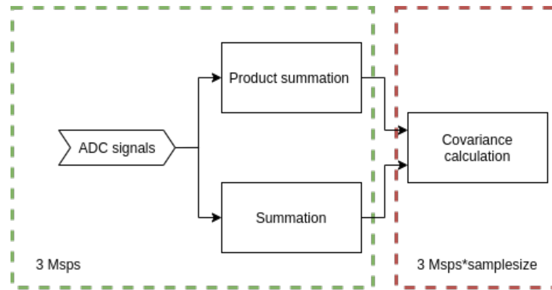


Figure 5.2: The covariance processing diagram

The different aspects of the covariance implementation on the FPGA will be discussed and the choices are elaborated. The end of this section will provide a description and conclusion on the final chosen implementation.

5.1.1 Structural process

Structural in the FPGA context means connecting pre-verified processing blocks with known resource usage to create the required process. This approach results in a very fine-grained control and insight into where the resources are utilized. The summation of the incoming ADC signals is handled in a structural method and is shown in figure 5.3. Here the process can be seen as connections between processing blocks. The summation has only one drawback, it requires its resulting adder signal as input for the adder. This is a problem as when there is no buffer in between the output and the input there is a possibility for metastability problems to occur. This happens because a change at the input changes the output which in turn changes one of the inputs of the adder. However, this is not the only problem as the summation IP block is to be re-used to lower the resource utilization. This re-using of the same IP block introduces an additional requirement for the data to be stored for 1 sampling period before being presented at the input of the adder. To solve this data dependency a shift register is used between the output and input of the adder block, as shown in figure 5.3. This shift register retains the result of the adder calculated using the previous sample, which is to be used in the next (partial) sum calculation for the incoming signals.

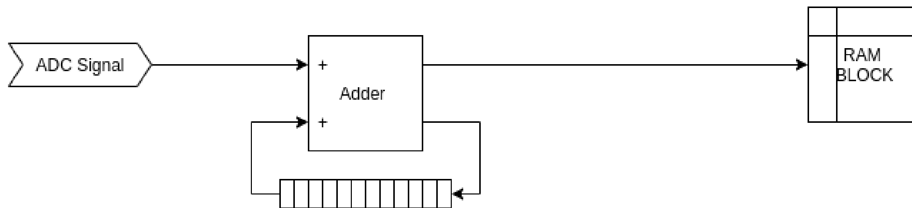


Figure 5.3: The FPGA structural summation with shift register to store the partial sums

The product implementation using the Complex multiplication IP block from Intel is implemented as shown in figure 5.4. Where a multiplication of two signals is added before the adder system of figure 5.3.

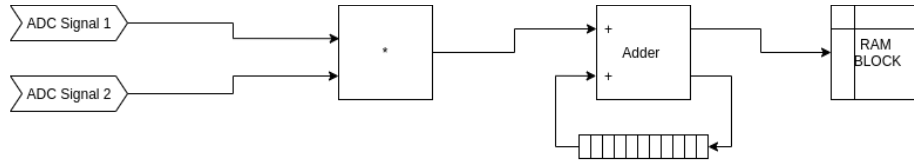


Figure 5.4: The FPGA structural summation of the product with shift register

The complete implementation of the covariance calculation in a structural format is shown in figure 5.5. Where teal is used to show the most crucial latency, those calculations must be done within the time between ADC samples. This is due to the inputs only being present for this time before new signals are present. Blue shows the covariance calculation. This is less crucial since the covariance has the maximum latency of (clock cycles between adc samples $*$ samplesize) clock cycles. Red is used to show the least time crucial part, this contains supplying the complete covariance matrix to the next part of the total angle of arrival processing chain.

5.1.2 Partial sums

For the summation of the products and the ADC signals, partial sums are required to store the data of the previous cycle. To do this a shift register is implemented, the implemented shift register is an IP block from Intel. This shift register's clock enable input controls a Finite State Machine(FSM) to control the clock signal, the FSM determines when the clock signal is passed through so smaller shift registers can be used than would be necessary when the clock signal is connected directly to the clock input of the shift register.

5.1.3 Clock frequency

Clock frequency has an impact on the processing speed, heat production and power consumption. The choice of the clock frequency is to be well balanced to provide the required processing speed and maintain a limited heat production and power consumption to keep the cooling solution as minimal as possible.
The clock frequency used as the starting point is 60Mhz as this is the current systems clock frequency and is known to require no external cooling in the form of active or passive cooling. The maximum clock frequency to be used is 400MHz, the maximum of the FPGA on the SoC. This frequency range results in a range of $60Mhz/3Msps = 20$ clock cycles to $400Mhz/3Msps = 133$ clock cycles between ADC samples arriving at the system, this is a significant range in the processing time available between the ADC samples.

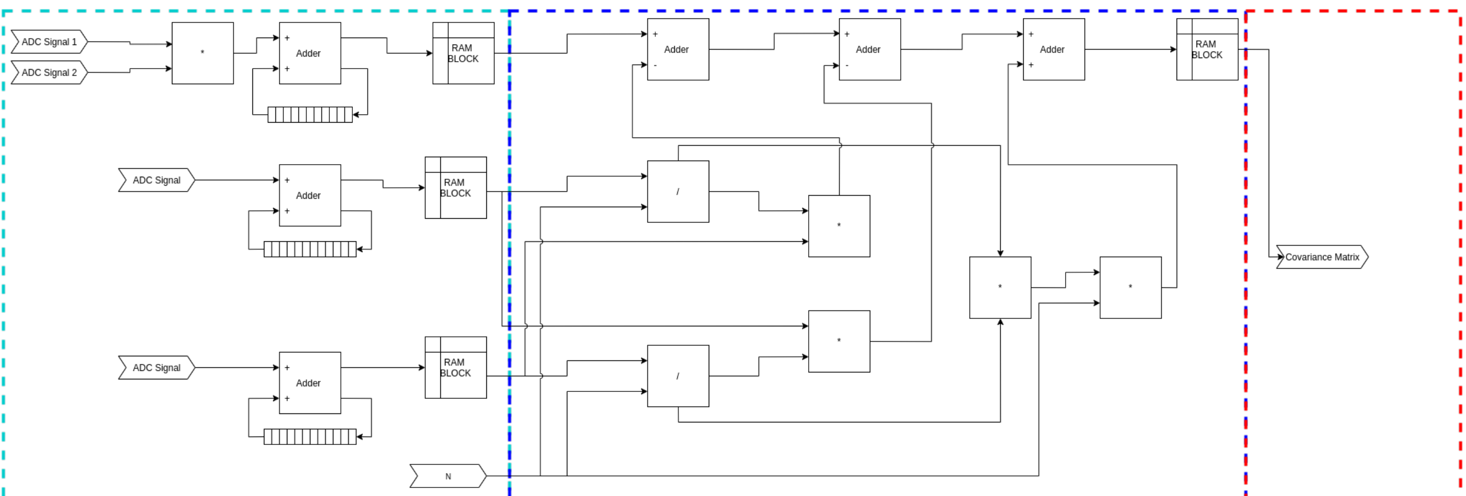


Figure 5.5: The implementation with colored dotted boxed around the parts that share the same timing constraints. Teal is per ADC sample, Blue is 78 times per 1024 ADC samples and Red is once per 1024 ADC samples

Clock frequency is constrained by more factors in the design outside the generation of heat and the required clock cycles. Namely another factor in the selection of clock frequency is the longest combinatorial path between clock cycles. The timing of the longest combinatorial path consists of propagation delays of the largest chain of consecutive non clocked logic.

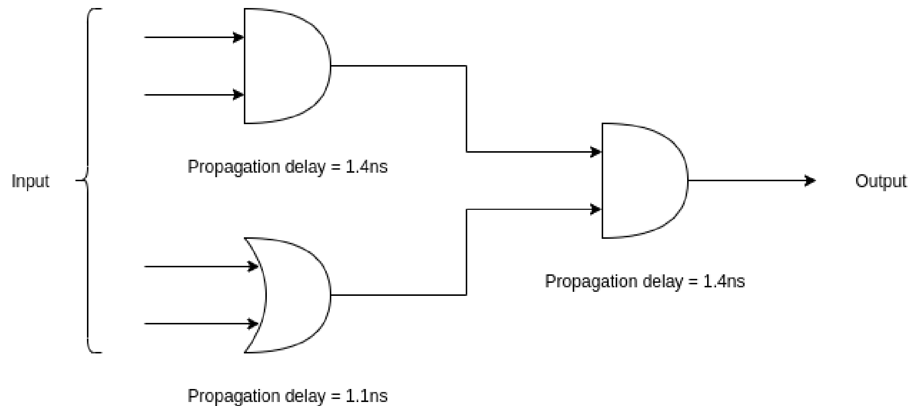


Figure 5.6: Illustration of the largest combinatorial path

An example of the longest combinatorial path can be given using figure 5.6. If the propagation delays of the 2 AND gates are added the total propagation delay is $2.8ns$ which results in a max clock frequency of $\frac{1}{2.8ns} = 357.1429MHz$. However, if the calculation would have been made using the path of the OR gate and the AND gate the total propagation delay would have been $2.5ns$ which results in a max clock frequency of $\frac{1}{2.5ns} = 400MHz$. This shows the critical path of this very small and simple system to be the path of the 2 AND gates as it has the lowest requirement for the maximum clock frequency to produce a correct value at the output.

Resource limitation The main resource limitation of the FPGA is the "area" (logic elements) on the FPGA. The constraint in our design is the maximum of 25k logic elements as this is the amount of logic elements in the FPGA. This is one of the units used to show resource utilization, another unit is ALMs which are used by the synthesis tool provided for the Cyclone V QuartusII. The amount of ALMs available on the Cyclone V is 9430. This limitation indicates the ALM usage for covariance calculation should be less than 9430. There are other tasks implemented on FPGA in addition to the covariance calculations. The FPGA is used by the entire algorithm and the communication between peripherals and the ARM chip. This requires a limit to be set beyond which the implementation is no longer feasible, this resource limit is chosen to be 1000 ALMS.
Resource limitation would also apply to the RAM storage, as this is a finite resource on the Cyclone V chip. However, this resource is in abundance for this

application and thus not discussed in this report.

Cycles The main part of the program that has to be executed every sampling cycle is the summation of all the incoming ADC signals and the summation of the required products for the covariance matrix calculation. In the current set up the clock of the FPGA runs at 60Mhz and the ADC generates 3 Msps, this results in having 20 cycles to handle the above-mentioned summations before the next samples are at the input of the covariance calculation block. This results in the processing block diagram shown in figure 5.7. In this figure the two timing domains are shown by there dotted boxes, the input is presented at a rate of 3 Msps and the output presents data once every 1024 samples. The output is the right upper triangle and the diagonal of the covariance matrix. This is possible because the covariance matrix is a Hermitian matrix. This gives the output matrix the property of the lower left triangle being the Hermitian conjugate of the right upper triangle. This property reduces the amount of products required for the covariance matrix form $12*12 = 144$ to $12*12*\frac{1}{2}+6 = 78$ where the +6 is required to fill the diagonal. This is a reduction of the amount of calculation of 45.8% compared to the full matrix.

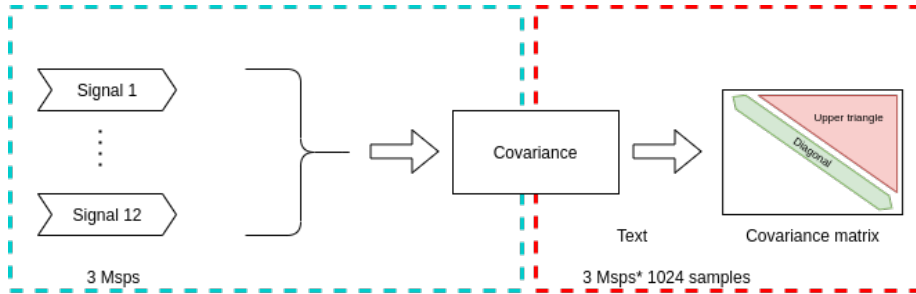


Figure 5.7: Covariance calculation block diagram

Summarized, the required calculations are:

- Summing the 12 incoming ADC signals
- Summing the 78 required products of the incoming ADC signals

Each summation can be done in a separate clock cycle, however, for the products, this is not possible as 78 is significantly more than the 20 cycles available. To solve this two solutions are available. Parallel calculation, in each clock cycle 4 products are calculated, since $20 * 4 = 80$ there are sufficient calculations for the required 78 calculations. Raise the clock speed, for this amount of calculations it should be higher than $calculations *$ sampling speed in this case $78 * 3Msps = 234MHz$. The Parallel solution implies that 4 times the same calculations have to be implemented on the FPGA and not a single implementation that is reused 78 times, the 4 implementations will be reused 20 times. The raised clock speed implies that a single implementation can be reused 78

times. This makes raising the clock speed the least resource intensive choice and the chosen solution in this thesis.
The final calculation, figure 2.10, of the covariance matrix is done spread out over all of the clock cycles required to obtain all the samples to determine the covariance matrix. This is not a bottleneck in the system as this is far greater amount of cycles than are required to do the calculation, namely $1024 * 78 = 79872$ clock cycles.

Number of bits Bit depth plays a significant role in the resource usage of the FPGA as the number of bits required propagate to the output. 16 bits input results in a 44-bit output of the complete covariance system. The ways the number of input bits propagate to the output bits are shown in figure 5.8 and 5.9. Figure 5.8 shows that when a 16 bit input is summed for 1024 samples the maximum size of the output is the maximum size of input times the amount of samples. The bit representation of 1024 samples is 2^{10}, the calculation of the number of output bits results in equation 5.1. An important remark is the data is presented in a signed format, this means that the 16 bit signals carry 1 sign bit and 15 data bits, meaning an multiplication of two 16 bit signals is $2^{15} * 2^{15} = 2^{30}$ data bits with one sign bit resulting in a 31 bit output.

$$2^{16} * 2^{10} = 2^{26} \tag{5.1}$$

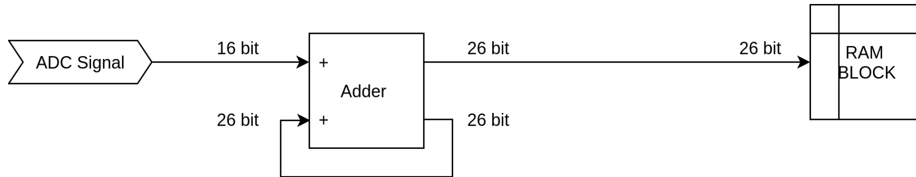


Figure 5.8: Bit depth for summation

For figure 5.9 the same calculation holds as for the the summation in addition to the multiplication before the summation. In the situation of the multiplication it can be observed the lowering of 1 bit at the input results in the output number of bits lowering with 2.

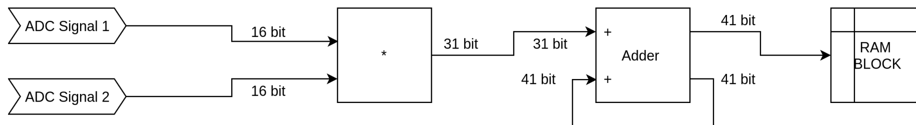


Figure 5.9: Bit depth for the product calculation

For the effect of the bit depth on the covariance calculation figure 5.10, where for all the mathematical calculations the input and output number of bits are shown.

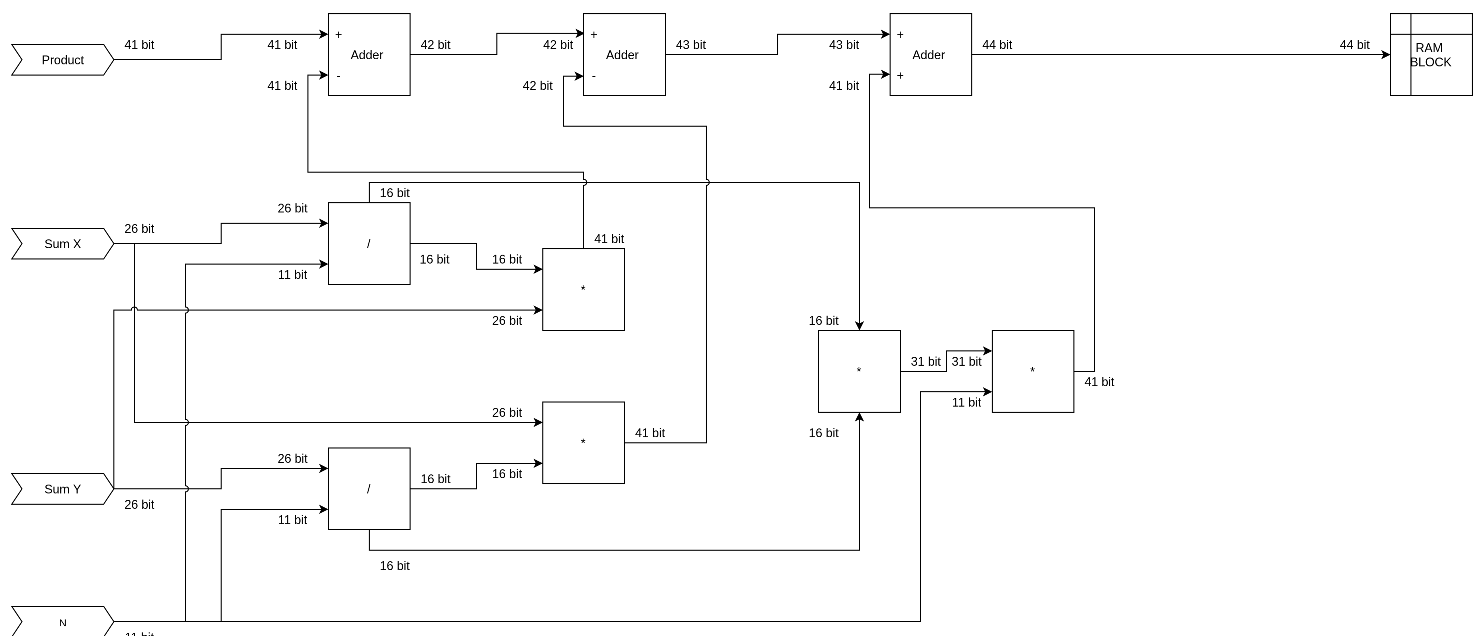


Figure 5.10: Bit depth for the covariance calculation

Reducing the bit depth by 1 lowers the output number of bits by more than 1 at the covariance output. This has a significant impact on the required calculation resources implemented on the FPGA. When considering an implementation similar to the one presented in this thesis, the bit depth of the input should be considered to be lowered to reduce the FPGA resources.

To combat the loss of dynamic range of less bit depth an automatic gain controller(AGC) can be used to always utilize the full range of the ADC. This would allow the system to, for example, work with an 8 bit ADC and have sufficient resolution whilst reducing the resource cost and complexity of the processing done on the FPGA.

For this thesis it was not possible to change the ADC hardware and implementing an digital AGC was outside of the scope of this thesis. This results in not implementing the AGC mentioned in this paragraph.

Storage The results of the calculations have to be stored, this has to be done in RAM blocks as otherwise to store the signals logic elements are used, which are the computing resources of the FPGA. Using the logic elements is a solution to be avoided when possible since the FPGA has dedicated storage resources, M10K blocks. Such dedicated storage solution is utilized by the RAM block IP for the FPGA provided by Intel. The RAM block uses the dedicated M10K storage resources on the FPGA SoC, this way the storage of the results is handled without increasing the logic utilization of the design. Intermediate signals are not being stored using the RAM blocks due to the time limitation, the time that it takes to send all the signals into the RAM block and retrieve them for the calculations exceeds the latency limit by a significant margin. However for the intermediate signals the shift registers are used which utilize the M10K storage blocks.
RAM blocks, when the 2 port configuration is used, can be used to cross clock domains within the FPGA, which enables us to have different clock domains on the FPGA. In this way higher clock frequency is provided when it is necessary for the time critical operations and a lower clock frequency can be used to reduce power consumption.

Complex calculations Complex multiplications, divisions and additions are required for covariance calculation. Complex addition is simple. For complex product calculation, 2 separate sets of calculations of real numbers. To calculate complex multiplication we have two choices: writing it from scratch in VHDL or using an IP block provided by Intel. The choice of the implementation depends on the requirements. The Intel IP implementation comes with a reduction in development time since the IP block implementation is generated using a design wizard in QuartusII. The design from scratch implementation comes with more insight into the actions taken in the block and freedom to change certain aspects and behaviour of the calculation.
In the implementation for this report the Intel IP block implementation is chosen. The IP block was tested and the performance was observed to be within one

clock cycle. Furthermore, correct values were achieved with minimal resource usage. Considering these, the Intel IP blocks were chosen for the complete design of the covariance implementation on the FPGA.

5.2 Eigendecomposition

Implementation choices and problems faced for each the implementation of the eigendecomposition are presented in the next sections.

5.2.1 CPU

The implementation on the ARM CPU is made using the programming language C++.

FPGA covariance data translation

The covariance matrix that is received from the FPGA is not complete, two operations have to be applied to the matrix. Dividing all the values by 1023 to complete the covariance calculation and filling the lower triangle of the matrix as shown in figure 5.11.

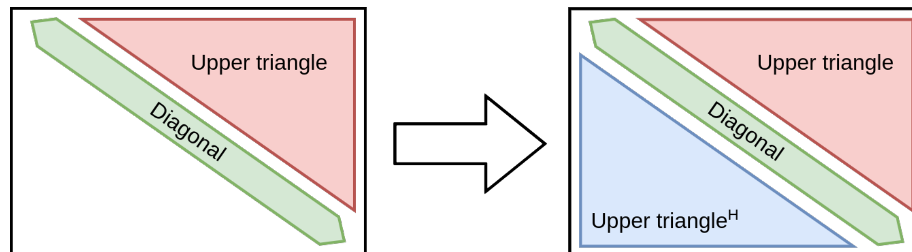


Figure 5.11: Filling the covariance matrix using the FPGA data by applying the Hermitian conjugate to the upper triangle

Eigendecomposition

For the eigendecomposition the armadillo library for C++ is used [18] [17]. The armadillo library provides functions for linear algebra operations and uses different back-ends (LAPACK [5]and Openblas [22] [23]) for the calculation. This library has a special eigendecomposition function for symmetric/Hermitian matrices. This function takes advantage of the characteristics of such matrices to increase the speed of the eigendecomposition. This function in the library implements the divide and conquer technique discussed in section 2.2 to calculate the eigenvalues and eigenvectors of the covariance matrix.

5.3 Localization

Implementation choices and problems faced for the implementation of the localization are discussed in the next section.

5.3.1 CPU

The implementation of the localization on the ARM CPU is programmed in the C++ programming language.

Steering vector generation The first step for the localization is to generate the required steering vectors which can be saved to a binary format by the Armadillo library used. This is done by a function that for each point in an XY raster on a fixed Z calculates the corresponding steering vector to the antennas using equation 2.9. In this thesis the raster is of size $251 * 251$. The raster spans a physical space of 5*5 m, resulting in an maximal accuracy of 2cm for the localization.

Isolating the noise space To calculate the MUSIC spectrum a noise space is required. This noise space is isolated from the eigenvector space by splitting the eigenvector space into a signal space and a noise space. This is done by splitting of the largest eigenvectors of the eigenvector space, the amount is equal to the amount of expected signal sources. This leaves the remaining eigenvectors to be the noise space.

Calculating the spectrum The spectrum is calculated using equation 2.10, where $a(x, y, z)$ is the steering vector for the specific point on the raster and E_N is the noise space.

Libraries The matrices and matrix operations are implemented using the "Armadillo" library [18] [17]. This is a library that performs efficient linear algebra operations using multiple different possible back-ends. The back-ends used in this implementation are LAPACK [5] and OpenBLAS [22] [23].

Chapter 6

Analysis of the solutions

In this chapter the final solution for implementation of the complete algorithm is presented on separate processing platforms. Now that the MUSIC algorithm is implemented on a SoC platform, the data gathered from the complete chain and individual processing blocks can be analysed to determine the feasibility of the solutions. If the feasibility is not adequate other solutions will be recommended. The areas which are to be analysed are:

- performance
- resource utilization
- maximum latency of 0.5 seconds for the processing chain

The combination of the above mentioned factors give a measure of the feasibility of the solution for the processing block of the MUSIC algorithm on the SoC platform

6.1 Covariance

This section provides analysis on the implementation discussed in section 5.1. The performance will be analysed to determine wether the processing is done within in the required time and how the resources are utilized.

Performance The performance of the solution in for the covariance implementation is determined by the ability to process the required calculations within the time between two ADC samples from the RX boards. The FPGA clock for this processing block is $240MHz$ resulting in $\frac{240MHz}{3Msps} = 80$ cycles to process the samples of the 12 ADCs. The processing time in cycles for the different processing blocks mentioned in figure 5.2 is shown in table 6.1. The product summation and summation should be processed withing the 80 clock cycles. As can be seen in table 6.1 these meet this requirement. The covariance matrix meets the requirement since it is, with significant margin, within

$80 * 1024 = 81920$ clock cycles to calculate the matrix. The 81920 clock cycles are because it has to process the previous product summations and summations before the next values are stored in the RAM block to calculate the covariance matrix. Meeting the requirement with this significant margin is beneficial for the processing blocks following the covariance in the block diagram of figure 4.1 to be within the total latency of 0.5 seconds. The total processing time of the covariance on the FPGA is $341.7\mu s$ calculated using equations 6.1 and 6.2.

$$\text{processing time} = \text{required clock cycles} * \frac{1 \text{ second}}{\text{clock frequency}} \tag{6.1}$$

$$82000 * 4.16666667e^{-9} = 341.7\mu s \tag{6.2}$$

	Cycles required to calculate
Product summation	78
Summation	12
Covariance Matrix	80

Table 6.1: Processing time in clock cycles for the covariance on the FPGA

Resource utilization In this paragraph the resource usage of the implementation on the FPGA is analysed. The resources used by the implementation on the FPGA compared to the available resources of the FPGA is shown in table 6.2. DSP block utilization is an area where the usage is large, i.e. 55.56% of total available DSP blocks. This was expected with the complex multiplications required and the realtime constraint of the covariance calculation. However, for the other categories the overall resource usage of the covariance is below 10% of the available resources. The low resource usage is beneficial, as mentioned in section 5.1.3, since the resources are not exclusive to the Angle of Arrival implementation in this thesis.

	Used by covariance	Available on the FPGA	Percentage
Logic utilization (ALMS)	764	9,430	8.1%
Registers	202	37,736	0.5%
Block memory bits	15,080	1,400,000	1.1%
DSP blocks	20	36	55.6%

Table 6.2: Resource utilization by the covariance implementation of the FPGA

6.2 Eigendecomposition

This section will analyse the processing time required by the eigendecomposition implementation of section 5.2. The processing time for the eigendecomposition has a significant spread, this is due to the initialization of the variables in the first iteration of the eigendecomposition. This is seen by the average processing time of table 6.3 is close to the minimum processing time. This shows the

maximum value of table 6.3 is one of 3 outliers in the processing timings as the average processing time is 0.013 ms slower compared to the minimum processing time. These outliers are due to the OS scheduling another task instead of the eigendecomposition. The spread of computation is large due to outliers . When taking the outliers out, the computation time spread is shown in Figure 6.1, where the distribution of the computation times is clearly visible. The spread is due to the OS performing other tasks on the background and the application not having exclusive access to the CPU.

	Avg	Min	Max
Processing time(ms)	0.352	0.339	1.164

Table 6.3: Processing time for the Eigendecomposition on the ARM CPU over 3000 iterations

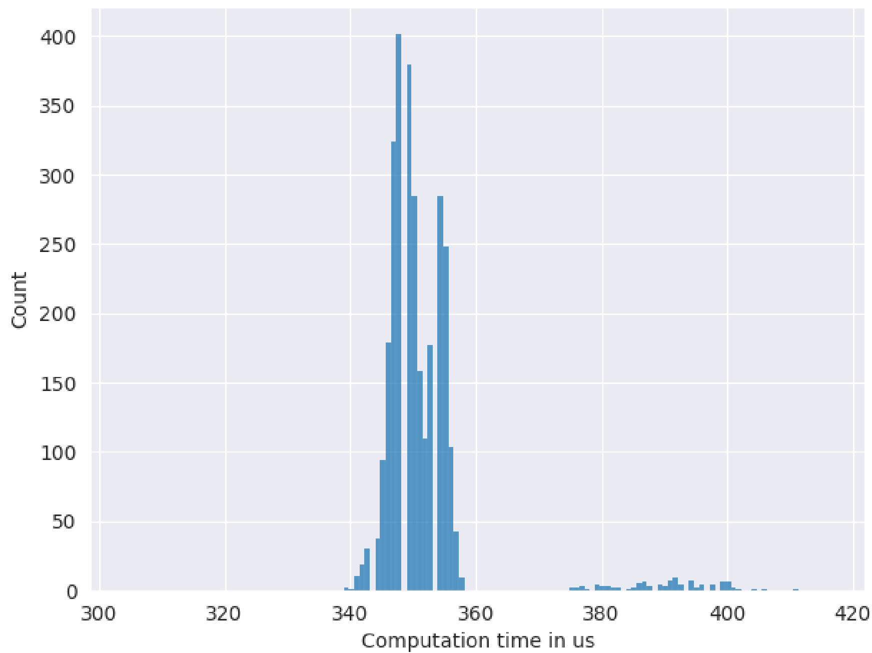


Figure 6.1: Histogram of the eigendecomposition computation times zoomed in

6.3 Localization

This section will analyse the processing time required by the localization implementation of section 5.3. The spread of the processing times can be seen in table 6.4, where the minimum, maximum and average processing times are

shown. The spread of the values in table 6.4 shows the maximum value is an outlier as the average over 3000 iterations is 1.559 ms slower than the minimum processing time and 26.687 ms faster than the maximum processing time. This is a trend for the ARM CPU platform if section 6.2 is taken into account as well. The spread of the computation time is shown in Figure 6.2 using a histogram, the figure shows how the average computation is higher due to the outliers. The spread is due to the OS performing other tasks on the background and the application not having exclusive access to the CPU.

	Avg	Min	Max
Processing time(ms)	365.855	364.296	392.542

Table 6.4: Processing time for the localization on the ARM CPU over 3000 iterations

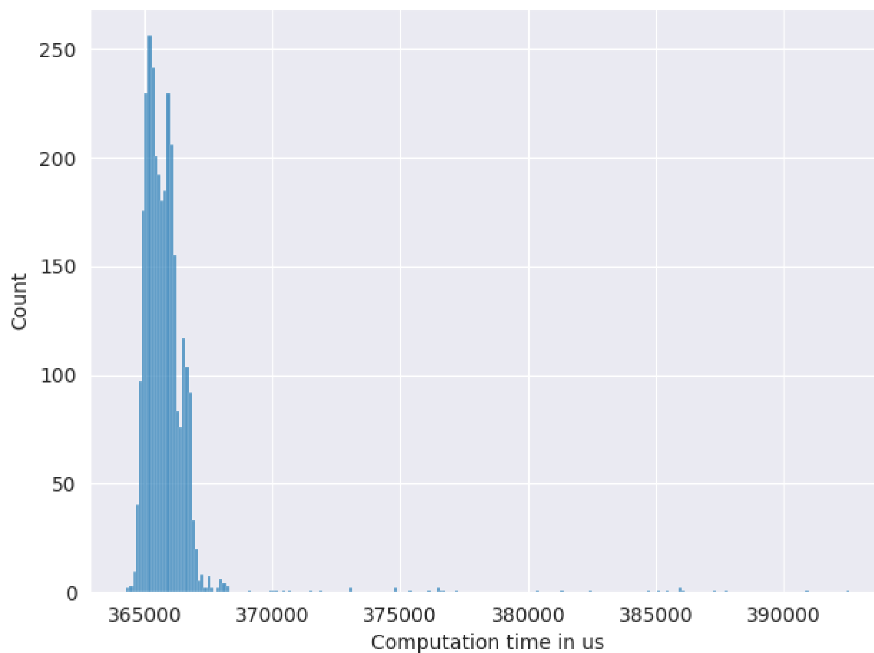


Figure 6.2: Histogram of the localization computation times

6.4 Complete signal processing chain

The complete processing chain, consists of the two processing platforms: FPGA and ARM CPU where the division of processing done in chapter 4 is shown in

Figure 6.3. The complete latency from ADC data to tag location should be within 0.5 seconds constraint.

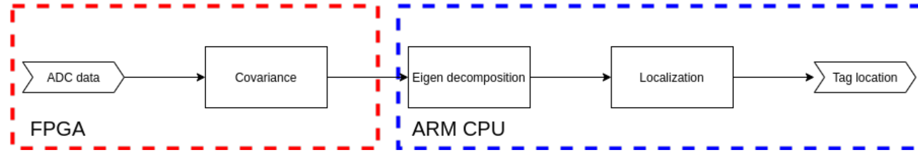


Figure 6.3: The complete processing chain

The complete processing chain can be split in two separate processing times, the processing time on the FPGA and the ARM CPU.

FPGA processing time The FPGA processing time is deterministic and fixed since the FPGA processing is done as hardware. The processing time is calculated in section 6.1 to be $341.7\mu s$ or $0.3417ms$.

ARM CPU processing time The ARM processing time consists of the eigendecomposition and the localization combined. This combination is tested over 3000 iterations and the processing time is shown in table 6.5. The spread of the computation time is shown in Figure 6.4 using a histogram. The figure shows how the average computation for combination of localization and eigendecomposition is higher due to the outliers which is similar to the localization.

	Avg	Min	Max
Processing time(ms)	366.000	364.655	390.861

Table 6.5: Processing time for the processing chain on the ARM CPU over 3000 iterations including outliers

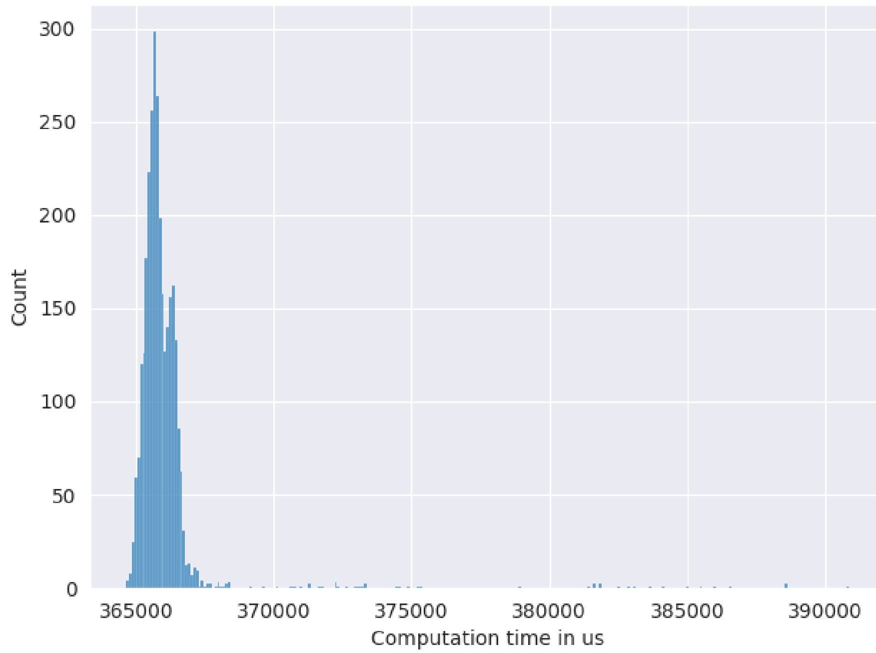


Figure 6.4: Histogram of the completer ARM computation times

Complete chain processing time The complete chain's processing time is adding the processing times of the FPGA and ARM CPU. This results in a processing time of an average 364.997ms and a maximum 391.203ms including the outliers. The average and maximum processing times lie, with a minimum margin of 100ms, within the constraint of 0.5s or 500ms.

Chapter 7

Results and discussion

Here the results of the implementation of the complete algorithm are presented. To validate the answers of the processing blocks python with a widely used library, Numpy, is used [21] [13].

7.1 Covariance

7.1.1 Results

The result of the covariance implementation on the FPGA is discussed in this section. The Numpy covariance function will be supplied with the same input data as the FPGA implementation and the values will be compared. What will be observed is the output of the Numpy function is a floating point number and the output of the FPGA implementation is an integer. This results in an error that will be discussed below. First, the output of Numpy is shown in table 7.1. For the readability a 3*3 sub array is taken from the upper left corner of the output matrix. The same sub array is taken from the output of the FPGA and is shown in in table 7.2. The previous statement on the difference in number representation is observed when comparing the two tables.

	S1	S2	S3
S1	39881.02735139+0j	-5391.83977311+35908.66478991j	49423.97414162+10372.74660542j
S2	-5391.83977311-35908.66478991j	33981.6564772+0j	2736.91463717-46390.16252177j
S3	49423.97414162-10372.74660542j	2736.91463717+46390.16252177j	65013.19838633+0j

Table 7.1: Covariance Numpy

	S1	S2	S3
S1	39881+0j	-5392+35909j	49423+10373j
S2	-5392-35909j	33982+0j	2737-46390j
S3	49423-10373j	2737+46390j	65013+0j

Table 7.2: Covariance FPGA

When taking a closer look at the tables 7.2 and 7.1, it can be observed that the difference in values is no more than 1, as the value is floored to the nearest integer of the floating point number shown in table 7.1. The maximum error for the covariance due to the truncation done in the division is calculated by the effect of the propagation through the covariance calculation shown in Figure 2.10, this calculation is shown in equations 7.1 and 7.2. The maximum error for the implemented covariance system on the FPGA is 1024. Moreover, a division by 1023 is done on the ARM CPU processor to preserve floating point values. This fact can be seen when comparing the covariance matrices from python and FPGA (tables 7.2 and 7.1), the error does not exceed 1.001.

$$\text{Max error} = -\text{Max summation value} - \text{Max summation value} \\ +\left(\left(\frac{\text{Max summation value}}{\text{sample size}} + 1\right)^2 - \left(\frac{\text{Max summation value}}{\text{sample size}}\right)^2\right) * \text{sample size} \tag{7.1}$$

$$-2^{27} - 2^{27} + \left(\left(\frac{2^{27}}{1024} + 1\right)^2 - \left(\frac{2^{27}}{1024}\right)^2\right) * 1024 = 1024 \tag{7.2}$$

7.1.2 Discussion

For the covariance matrix calculation implementation on the FPGA, it is observed that the error is due to the use of integers and requiring a division. This could be solved by taking the covariance calculation and adjusting it to either avoid division on the FPGA or move the division to the end of the calculation which reduces the error in the covariance matrix. Since most elements in the covariance calculation (see equation 2.5) rely on the division, this is the place to increase the accuracy most for the covariance implementation.

7.2 Eigendecomposition

7.2.1 Results

The eigendecomposition implementation is done in both python and C++ where python is the baseline for validation of the C++ eigendecomposition. The eigenvalues acquired by Python and C++ are shown to be identical by table 7.3. However, when looking at the eigenvectors in table 7.4 the vectors are not the same element wise. When projecting the two eigenvectors on each other the projection shows the vectors to be identical due to the projection value being

1.0. This is possible because the eigenvectors are normalized, being unit vectors. The projection value of the two vectors is calculated by using equation

$$Projection = \mathbf{vector1}^T \cdot \mathbf{vector2}^H \tag{7.3}$$

Python Eigenvalues	C++ Eigenvalues
3.1863e+05	3.1863e+05
2.0313e+03	2.0313e+03
9.3701e+02	9.3701e+02
6.3122e+02	6.3122e+02
3.5468e+02	3.5468e+02
2.9582e+02	2.9582e+02
1.6448e+02	1.6448e+02
1.3200e+02	1.3200e+02
8.0242e+01	8.0242e+01

Table 7.3: Eigenvalue comparison between Python and C++

C++ eigenvector	Python eigenvector
-0.33737613+0.10133277j	0.35226409+0.j
-0.39408206+0.21520513j	0.43933252-0.09274676j
-0.35878056+0.08726183j	0.36871753+0.01963406j
0.05021557+0.23421448j	0.01928504-0.23876082j
0.13762553+0.29380102j	-0.04729243-0.3209719j
0.12833544+0.30085998j	-0.03636325-0.32506106j
0.27690357+0.06280886j	-0.24713071-0.13981038j
0.33482636-0.12070866j	-0.35539705+0.01928813j
0.24704778+0j	-0.23660437-0.07106667j

Table 7.4: Eigenvector comparison between Python and C++, with a projection value of 1

7.2.2 Discussion

When designing the assumption made in the covariance calculation, the integer based covariance calculation does not impact the accuracy of the system, it is shown the projection value of the eigenvectors lies between 1 and 0.9996. This shows the assumption to be correct, the integer based calculation does not impact the eigendecomposition is negligible.

7.3 Localization

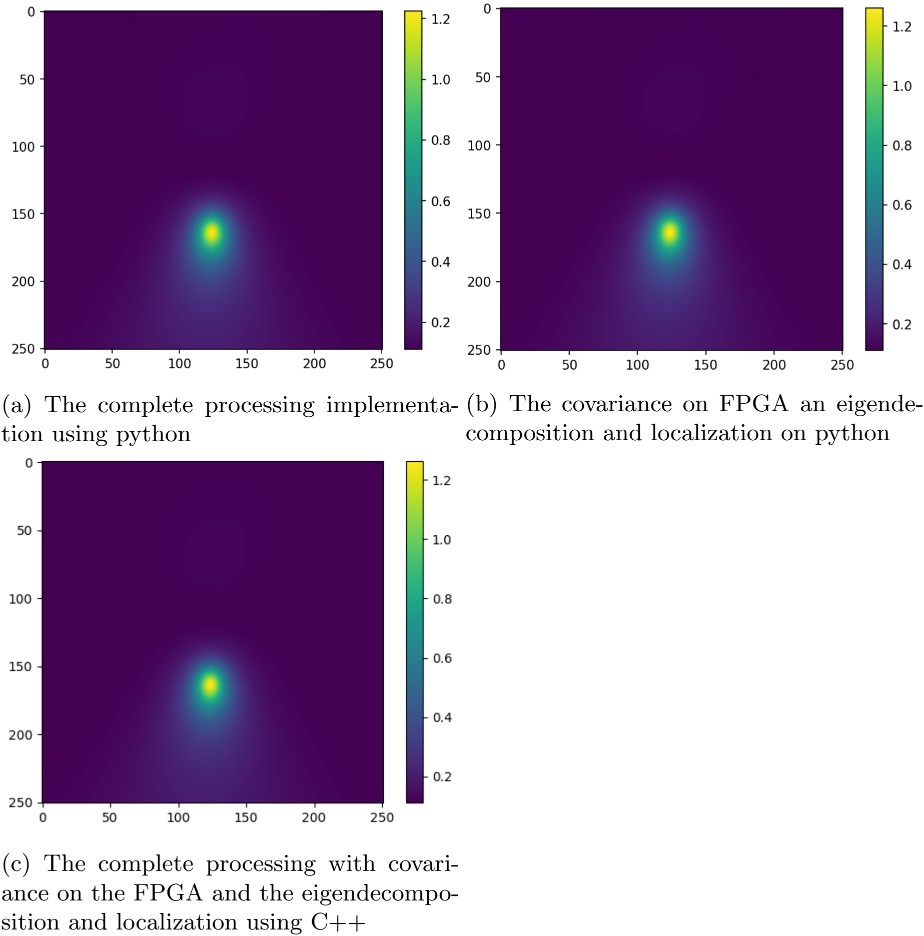
7.3.1 Results

The localization has two results in the context of this thesis: the music spectrum and the location of the RFID tag that is transmitting the response. For this section several comparisons are done to show validity of the result, this resulted in three scenarios:

- a pure python implementation, from covariance to localization
- a FPGA covariance implementation with python eigendecomposition and localization
- a FPGA covariance implementation with C++ eigendecomposition and localization

This shows the effect of the covariance error due to the FPGA implementation on the localization result. It also shows the result of the C++ implementation of the eigendecomposition which is compared to the python implementation using Numpy.

The spectra generated by the implementations are shown in Figure 7.1. Observing the spectra it is clear they calculated spectra are very similar. Figure a and Figure b of Figure 7.1 are almost identical, indicating the error introduced by the FPGA implementation of the covariance is negligible in the current implementation of the processing chain. One remark in to be made is the maximum height of the spectrum in Figure b is slightly higher than Figure a, showing the difference in the input covariance matrix between the two implementations. Figure c is almost identical to Figure b.



(a) The complete processing implementation using python

(b) The covariance on FPGA an eigendecomposition and localization on python

(c) The complete processing with covariance on the FPGA and the eigendecomposition and localization using C++

Figure 7.1: MUSIC spectra generated by the different implementations

The next result of the localization is the actual determined location of the tag, the locations with their corresponding implementation are shown in table 7.5. The stability of the location determination across the three implementation is high, having a maximum deviation of 1 location on the raster in either X or Y direction. When comparing the results with the actual physical tag location, the X location for all three implementations is correct with a maximum error of 1. The Y location of the three implementations has a larger deviation than the X location with an error between 3 and 4 locations on the raster. Knowing the raster has a resolution of 2cm per index the localization error by the implementation is 8cm. However, the results are inconclusive to what platform is the best to implement the algorithm on as the implementations have either better Y localization or X localization. This shows the algorithm to be insensitive to the platform it is implemented on.

	X-index	Y-index
Python	124	164
FPGA and Python	123	163
FPGA and C++	123	163
Physical location	124	160

Table 7.5: The location of the tag in the raster given by the different implementations

7.3.2 Discussion

The error in the implementation of the localization is due to lack of precision in the steering vectors. For more precise steering vectors, the characteristics of the receiving antennas have to be well known, this is proven to be a difficult task when antenna suppliers do not provide the characteristics. This is a direction in which a significant increase in accuracy of the localization of the tag can be gained especially in the Y direction.

Chapter 8

Conclusion

In this thesis we investigated an efficient implementation for the MUSIC algorithm on an embedded system using hardware and software co-design. The summary of the constraints this thesis was required to comply with are shown in table 8.1.

Total latency	<0.5 seconds
FPGA resource usage	<1000 ALMs
Throughput	180 tag reads
Sample amount	1024
Supported antennas	12

Table 8.1: Constraints of this thesis

The research questions that have been set in chapter 1 to be answered in this thesis are as follows

- How is the MUSIC algorithm implemented with optimal performance on an Embedded system using hardware and software co-design?
  - Which processing platform is most suitable for covariance calculation?
  - Which processing platform is most suitable for eigen decomposition?
  - Which processing platform is most suitable for localization?
  - How do hardware choices propagate in the resource usage of the Embedded system?

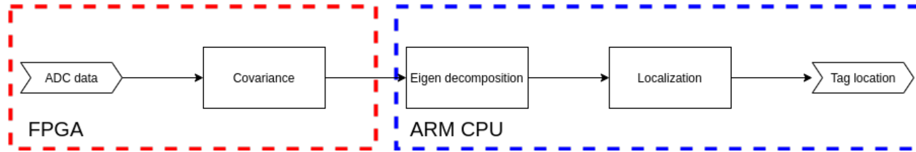


Figure 8.1: The complete processing chain

The available platforms for processing were introduced in chapter 3. Based on design decisions explained in chapter 4, the complete processing chain was designed and implemented as shown in Figure 8.1. Design decisions were made such that the constraints mentioned in chapter 1 are met. The platform to process the covariance is the FPGA because the FPGA's characteristics are suitable to adhere to hard timing constraints. Since the eigendecomposition and localization require floating point precision, for which a FPU is located on the ARM CPU, the ARM CPU was chosen to be the processing platform for the eigendecomposition and the localization. This choice concerning processing platform has the additional benefit of allowing the use of libraries which have an efficient implementation and are simple to use for the solution implementation.

The choice of implementation was tested and it was confirmed that it can comply with the timing constraints while providing the desired output data. The timing validation was done in chapter 6, where it was shown that the covariance implementation meets the hard deadline set by the constraints and the eigendecomposition and localization meet the softer latency by a margin of 100ms. The validation of the output was done in chapter 7. It was shown that the covariance output is correct according to a reference implementation and the possible maximum error is lower than 1% of the expected covariance values. The eigendecomposition and localization were validated in combination through the MUSIC spectrum and the estimated location. The accuracy of the eigendecomposition and localization algorithms in the embedded platform was validated compared to a reference implementation in Python which runs on a PC. The location estimate is within 10cm of the physical location and which is within 2cm accuracy of the Python estimate.

With the implementation of this thesis the throughput of the ARM processing chain is not sufficient to meet the 180 tag reads per second that is set as the constraint. However, this is due to the implementation of this thesis being a test bed the localization calculates the entire MUSIC spectrum instead of just finding the location. This requires $251 * 251 = 63001$ calculations to find the location. If the system was not a testbed an optimal peak finding algorithm could be used to find the peaks in the spectrum instead of calculating the entire spectrum.

The conclusion of the thesis is that the current implementation of the MUSIC algorithm strikes a good balance between performance and resource usage, whilst performing within the constraints of the application. Also, it has elaborated why the platforms are well suited for their processing parts of the MUSIC algorithm.

8.1 Recommendations

This section will show directions for future research to build upon this thesis.

8.1.1 ADC number of bits

For the resource usage due to hardware choices, the number of bits of the ADC is discussed in section 5.1.3. However, this thesis did not have hardware changes in its scope, therefore no implementation results have been presented to support the resource reduction by lowering the number of bits of the ADCs. As a future research, the effect of lowering the number of bits should be investigated to use the minimal number of bits that are required to achieve the accuracy needed by the implementation and lowering the resource utilization.

8.1.2 Throughput

To increase the throughput an optimization algorithm can be implemented to find the peaks in a more efficient way. If an algorithm for localization is used that finds the location in, for example using divide and conquer which has a worst case $O(nlog(n))$, 603 calculations are required lowering the computation time by a factor of 104 . This optimization results in a computation time of $\frac{391ms}{104} = 3.76$ for the localization which results in a processing time that is below the RFID reader read speed of $\frac{1 \text{ second}}{180\text{tag reads}} = 5.5ms$ This satisfies the constraint by a significant margin as the throughput is limited by the communication in this case.

8.1.3 Amount of antennas

This thesis used 12 antennas to determine the location of the RFID tag. Future research should investigate the use of less antennas to lower the complexity and data to be processed by the signal processing while maintaining the accuracy. This could allow the implementation to meet the requirements with a higher margin and gives more room for optimizing the implementation.

Bibliography

[1] White paper: The arm cortex-a9 processors. https://web.archive.org/web/20141117060156/http://www.arm.com/files/pdf/ARMCortexA-9Processors.pdf, Sep 2009.

[2] Cyclone v device datasheet. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_51002.pdf, Nov 2019.

[3] Nios® ii performance benchmarks. https://www.intel.la/content/dam/www/programmable/us/en/pdfs/literature/ds/ds_nios2_perf.pdf, May 2020.

[4] Abdulrahman Alhamed and Saleh Alshebeili. Fpga implementation of complex-valued qr decomposition. *2016 5th International Conference on Electronic Devices, Systems and Applications (ICEDSA)*, 2016.

[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[6] Constantine A. Balanis. *Field regions*. Wiley, 1997.

[7] Uzma M. Butt, Shoab A. Khan, Anees Ullah, Abdul Khaliq, Pedro Reviriego, and Ali Zahir. Towards low latency and resource-efficient fpga implementations of the music algorithm for direction of arrival estimation. *IEEE Transactions on Circuits and Systems I: Regular Papers*, page 1–12, 2021.

[8] Jan Cuppen. A divide and conquer method for the symmetric eigenproblem. *Numerische Mathematik*, 36:177–195, 06 1980.

[9] Daniel Dobkin. *The RF in RFID: passive UHF RFID in practice*. 01 2007.

[10] J. Francis. The qr transformation. *The Computer Journal*, 4, 04 1962.

[11] Upton Graham J G. and Ian Cook. *variance*. Oxford University Press, 2002.

[12] WALTER GANDER. Algorithms for the qr-decomposition. 01 1980.

[13] Charles R. Harris, K. Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585:357–362, 2020.

[14] Qinghua Huang and Naida Lu. Optimized real-time music algorithm with cpu-gpu architecture. *IEEE Access*, 9:54067–54077, 2021.

[15] Vijay Madisetti. *Subspace-Based Direction Finding Methods.* CRC Press, 2010.

[16] Kun Il Park. *Fundamentals of probability and stochastic processes with applications to communications.* Springer, 2018.

[17] Conrad Sanderson and Ryan Curtin. Armadillo: a template-based c++ library for linear algebra. *The Journal of Open Source Software*, 1(2):26, 2016.

[18] Conrad Sanderson and Ryan Curtin. A user-friendly hybrid sparse matrix class in c++. *Mathematical Software – ICMS 2018*, page 422–430, 2018.

[19] R. Schmidt. Multiple emitter location and signal parameter estimation. *IEEE Transactions on Antennas and Propagation*, 34(3):276–280, 1986.

[20] R.G.J.F.O.J. van Lakwijk. Tracking of uhf rfid tagged stock in retail. Jun 2013.

[21] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual.* CreateSpace, Scotts Valley, CA, 2009.

[22] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. Augem: Automatically generate high performance dense linear algebra kernels on x86 cpus. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2013.

[23] Zhang Xianyi, Wang Qian, and Zhang Yunquan. Model-driven level 3 blas performance optimization on loongson 3a processor. *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, 2012.

[24] Zhou Zou, Wang Hongyuan, and Yu Guowen. An improved music algorithm implemented with high-speed parallel optimization for fpga. In *2006 7th International Symposium on Antennas, Propagation EM Theory*, pages 1–4, 2006.

Appendices

Appendix A

Behavioral process

The first implementation of the covariance matrix calculation on the FPGA is done in behavioural format. The resulting process is shown in figure A.1, which contains optimizations to adhere to the required latency of the input data being available for only 20 clock cycles. This was shown to be not feasible as the resource usage was high and the nature of the behavioural format does not allow for fine-grained control over the resource usage of the calculation and implementation.

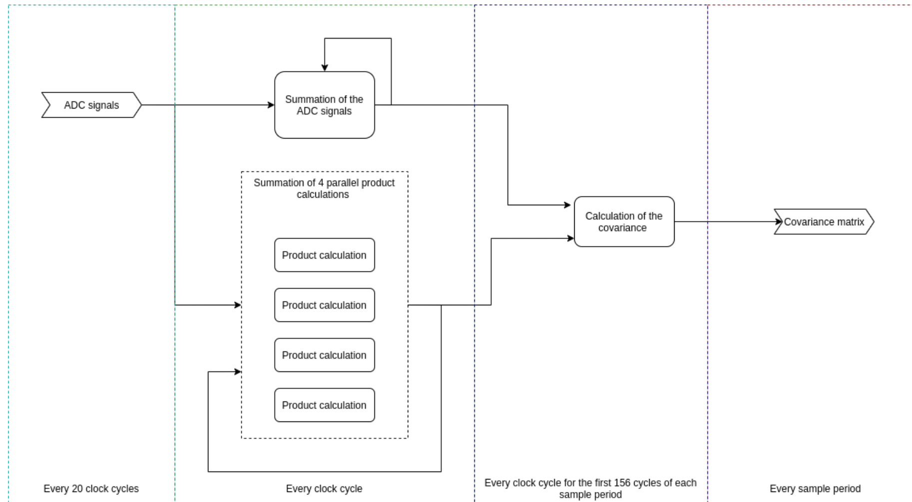


Figure A.1: The FPGA behavioral covariance process

Appendix B

Eigendecomposition

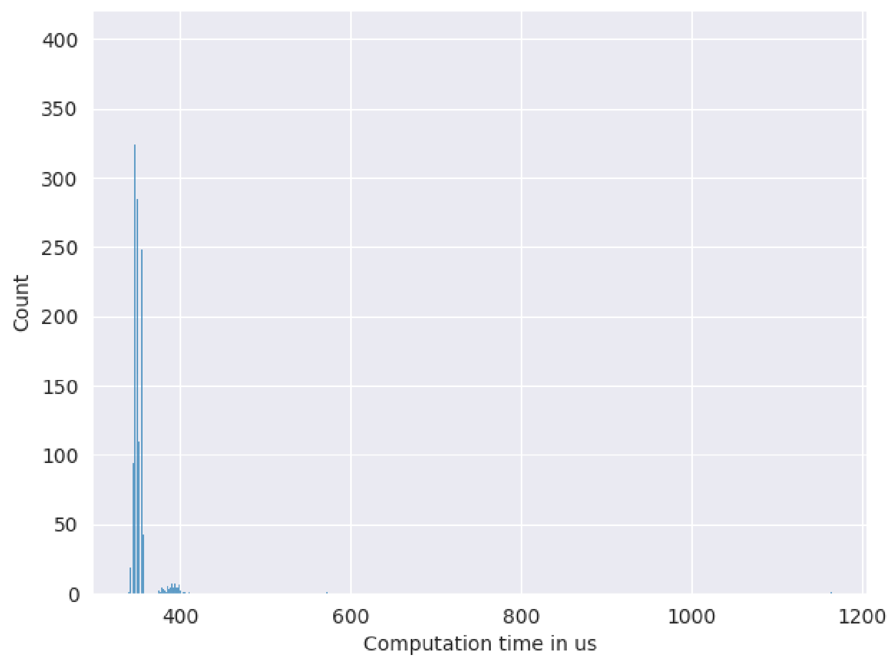


Figure B.1: Histogram of the eigendecomposition computation times