University of Twente
Faculty of Electrical Engineering, Mathematics and Computer Science
Computer Architecture for Embedded Systems

Phased Array Antenna Processing on Reconfigurable Hardware

M.Sc. thesis by

Rik Portengen

Graduation committee:
prof. dr. ir. Gerard J.M. Smit
dr. ir. André B.J. Kokkeler
ir. Marcel D. van de Burgwal
ir. Kenneth C. Rovers

Enschede, December 2007

Preface

This thesis presents the results of my work in the research of beam forming and the creation of a validation platform. During this project the development with an evaluation board is experienced. The interface with external modules delivered some challenges but eventually started to work.

The audio receiving array, the program source codes and this thesis are part of my master project at the Computer Science department of the University of Twente. The assignment was part of the Beamforce project at the chair Computer Architecture for Embedded Systems and Thales Hengelo.

I would like to thank my graduation committee for their support. For getting me this project and to be able to cooperate to get this final result. Marcel van de Burgwal was of great importance to my work for implementing a Montium version on the evaluation board and to help me with numerous questions about the interface. Also thanks to the people at Recore Systems which gave fast updates of the simulator and answers about the Montium architecture. Further I would like to thank everybody of the CAES group and students for a really nice time.

Finally I would like to thank Linda for her unconditional support during my master course and this graduation.

Contents

Introduction

In this document the research concerning digital processing of phased array antenna signals is described. A study on which algorithms will be suitable for implementing, how well these perform on a reconfigurable processor and how fast the throughput will be in different scenarios. This thesis will cover the mathematical approaches of beam forming and the design decisions taken to perform this task on reconfigurable hardware. In the chapter 1 the general phased array antenna concept is explained. In chapter 3 reference designs from other papers are treated. In chapter 4 the methods for beam forming are described. An algorithm and implementation are made in chapter 5 and 7, respectively.

Possible applications and estimated requirements for beam forming scenarios are given in chapter 8.

A verification platform of beam forming for audio has been designed and implemented on a development board. This design will be shown in chapter 6.

Phased array antenna processing

For reception of electro-magnetic signals an antenna is used. In case of a simple antenna it will receive this signal equally strong from all directions[1]. In many cases this is a usable approach. However, other systems like for example a satellite communication system or a radio telescope, often a directivity signal is required. The use of antennas which suppress interference and noise is then preferred. Traditionally, satellite dishes were used for this but now also phased array antennas are slowly introduced as receivers [6, 9].

Phased array antennas consist of multiple antennas spaced from each other. The use of multiple antennas has a number of advantages. It can be used to improve signal to noise ratio. The phased array antenna has a higher

[1]A monopole or dipole antenna placed vertical receives all signals equally strong in the horizontal plane

sensitivity in the perpendicular direction, this is called a beam. When performing processing on the individual antenna signals, it is also possible to steer the sensitivity of the antenna. This is called beam steering. Effectively, you can 'look' in different directions without mechanically moving the antennas. This processing is done digitally in this project. Performing beam forming digitally is commonly referred as Digital Beam Forming (DBF).

Reconfigurable hardware

Hardware can be developed to perform a fixed task. An example is a soundcard in the computer. This hardware is developed to perform the task of audio processing; it can perform this task possibly very fast and it could perform it energy efficiently. Reconfigurable hardware is developed to perform a variety of tasks. The goal of most producers [4] of reconfigurable hardware is to get an comparable performance with respect to a specific application domain as application specific hardware. The configuration of reconfigurable hardware can be altered such that the hardware can execute other tasks. In this way, one can use hardware to execute different tasks and take advantage of the reconfigurability.

Introductie

In dit document wordt het onderzoek over digitale verwerkering van fase array antenna signalen omschreven. Een studie over welke algorithmes geschikt zijn voor implementatie, hoe goed deze presteren op een reconfigureerbare processor en hoe snel de doorvoer capaciteit is in verschillende scenarios. Dit verslag zal de wiskundige aanpak van beam forming uitleggen en de ontwerp beslissingen die genomen zijn om deze taak op reconfigureerbare hardware uit te voeren. In hoofdstuk 1 is het concept van de fase array antenna uitgelegd. In hoofdstuk 3 zijn referentie ontwerpen van andere verslagen behandeld. In hoofdstuk 4 worden de methodes van beam forming omschreven. Een algorithme en implementatie worden gemaakt in de hoofdstukken 5 en 7.

Mogelijke applicaties en verwachte requirements voor verschillende beam forming scenarios worden gegeven in hoofdstuk 8.

Een verificatie platform voor beam forming met audio is ontworpen en gemaakt op een ontwikkel bord. Dit ontwerp wordt in hoofdstuk 6 omschreven.

Fase array antenne verwerking

Om radiogolf signalen te ontvangen worden antennes gebruikt. In het geval van een simpele antenne zal deze het signaal even sterk ontvangen vanuit alle richtingen[2]. In veel gevallen is dit een bruikbare aanpak. Echter, andere systemen zoals een sateliet communicatie systeem of een radio telescoop hebben vaak een signaal nodig dat richtings gevoeliger is. Het gebruik van antennes welke stoorsignalen en ruis onderdrukken is dan gewenst. Traditioneel werden hiervoor satellietschotels gebruikt maar tegenwoordig worden ook vaker fase array antennes gebruikt [6, 9].

Fase array antennes bestaan uit meerdere antennes die verdeeld zijn. Het gebruik van meerdere antennes heeft een aantal voordelen. Ze kunnen ge-

[2]Een monopool of dipool antenna die verticaal geplaatst is ontvangt alle signalen even sterk in het horizontale vlak

bruikt worden om ruis te onderdrukken. The fase array antenna heeft een hogere gevoeligheid in de loodrechte richting, dit is een beam. Wanneer de individuele antennes apart worden verwerkt is het ook mogelijk om de gevoeligheid van de antenne te sturen. Dit wordt beam steering (sturing) genoemd. Effectief, kun je 'kijken' in verschillende richtingen zonder het mechanisch bewegen van de antenne array. De verwerking gaat digitaal in dit project. Het digitaal verwerken van beam forming signalen wordt vaak Digital Beam Forming (DBF) genoemd.

Reconfigureerbare hardware

Hardware kan ontworpen worden om een vaste taak uit te voeren. Als voorbeeld hiervan een geluidskaart van een computer; deze hardware is ontworpen voor de taak audio verwerking. Het kan deze taak mogelijk heel snel en bijvoorbeeld heel energie efficient uitvoeren. Reconfigureerbare hardware is ontworpen om een verscheidenheid aan taken uit te voeren. Het doel van de meeste producenten [4] van reconfigureerbare hardware is om een vergelijkbare prestaties te behalen in een specifiek applicatie domein in vergelijking met applicatie specifieke hardware. De configuratie van reconfigureerbare hardware is te veranderen en kan dan worden gebruikt voor andere taken. Hierdoor kan hardware meerdere taken uitvoeren en zijn voordeel doen van de reconfigureerbaarheid.

Chapter 1

Phased array antenna processing

A phased array antenna can be designed for a number of applications. Phased array antennas are used for example in mobile base stations, radio astronomy receivers and radar systems. In these applications phased array antennas can apply beam forming to change the sensitivity of the antenna in specific directions and to suppress interference.

1.1 Signal Model

A schematic representation of a phased array antenna system is shown in figure 1.1. This systems shows a standard setup of a possible array. The array is placed in the horizontal plane with antenna elements from west (left) to east (right). A signal coming from the north direction is coming perpendicular to the array. A signal from the west direction is coming parallel to the array, the array is build of equally spaced antennas. All signals drawn in the figure travel along the horizontal plane.

The signal arriving at the k^{th} antenna has a delay of

$$(d/p)sin(-\theta_0) \times k \tag{1.1}$$

seconds relatively to the first antenna. The symbols used in this equation are shown in table 1.1, these symbols will be used throughout this document. The angle θ_0 is measured relatively to the perpendicular of the array.

1.2 Processing

The individual antennas of a phased array antenna require processing to create a beam in a given direction. A beam represents a signal from a specific

k	Index of antenna
d	Distance between antennas
p	Propagation speed of a wave
θ_0	Direction of a wave, positive orientation is clockwise

Table 1.1: Symbols used

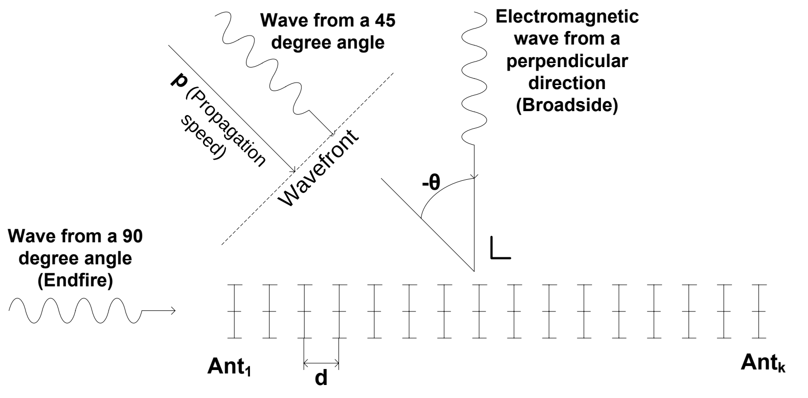

Wave from a 45 degree angle
Electromagnetic wave from a perpendicular direction (Broadside)
p (Propagation speed)
Wavefront
-θ
Wave from a 90 degree angle (Endfire)
Ant₁
d
Antₖ

Figure 1.1: Schematic representation of a line antenna array, top view

direction. By adapting parameters in the beam forming process, the direction can be steered. The processed signals of the antennas are added together to form this beam. This processing can be performed in different ways.

First a decision is made in which domain this processing is done. Signal processing can be done in both the analog and the digital domain. Analog signal processing requires devices such as phase shifters or delay lines for beam forming. A disadvantage of these devices is that they introduce signal loss, which results in less signal power and, after amplification, a lower signal to noise ratio. This signal loss gets worse when more devices, such as phase shifters, are used or many beams are created. Processing in the digital domain gives a number of advantages. After the signal is sampled and digitized at the analog to digital converters, no signal loss will occur. Multiple beams can be made without power loss. Digital processing gives flexibility in the steering direction. And the steering direction can be changed quickly when software processors are used.

The goal of this assignment is to find opportunities to use reconfigurability from processors for fast switching between different methods and different beam configurations. Therefore the focus will be on strategies which can be implemented digitally. For digital processing of phased array antenna signals these signals are converted by analog-to-digital-converters (ADC). The frequency of conversion is called sampling frequency (F_s).

A schematic representation of a beam forming system with the location of the processing algorithms is shown in figure 1.2. Different strategies for processing are searched and are explained in the chapter 4.

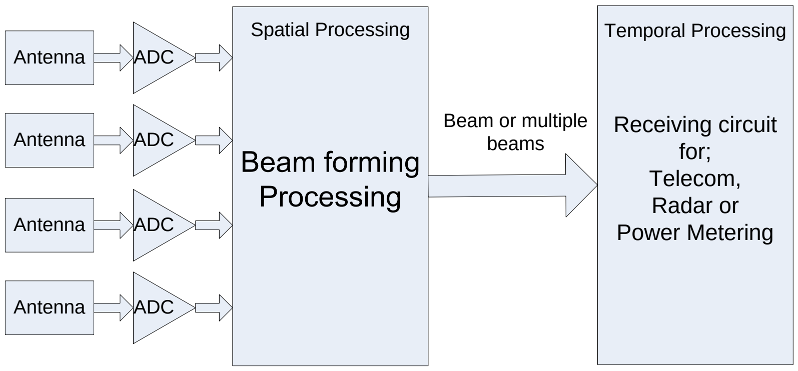


Figure 1.2: Total system with processing stages

1.3 Problem description

The assignment of this thesis is about the research of current techniques in beam forming and to build a validation platform for beam forming with the use of reconfigurable hardware. The reconfigurable hardware is the Montium. It is expected that this processor will be able to efficiently process phased array signals. Digital Beam Forming is very calculation intensive and the Montium is a energy efficient processor. A beam former system which implements this processor rather than a general purpose processor or FPGA will be more energy efficient.

In the following chapter some techniques for beam forming are explained from literature. In chapter 3 designs are discussed which have been used digital techniques for beam forming. Furthermore, a study is presented on which algorithms are suitable for implementing, how well these perform on a reconfigurable processor and what the throughput will be in different scenarios. This thesis will cover the mathematical approaches of beam forming suitable for digital processing and the design decisions taken to perform this task on reconfigurable hardware.

Chapter 2

Literature

2.1 Introduction to Radar Systems

The book "Introduction to Radar Systems" [1] explains the basics of radar and radar processing. It covers radar systems and different technologies to design radar systems. The book also covers noise and clutter (weather and environmental distortion), which can be observed in practical radars. Chapter 9 explains possible antennas that can be used to create a radar system. In this chapter the application of beam forming is explained and how this can be done in an analog and a digital manner.

In this chapter also a discussion about Baseband and IF Digitizing is made. When IF Digitizing is used with in-phase and quadrature signals, conversion with two analog to digital converters (ADC) can be done with a minimum sampling rate of 1.4 times the signal (half-power) bandwidth. It was stated by [11] that with direct digitizing of the baseband signal with only one ADC channel the minimum sampling rate becomes 5.4 times the signal bandwidth. The sampling rate has to be higher than the theoretical Nyquist rate of two times the signal bandwidth for avoiding distortion of the signal spectrum caused by folding.

2.2 Array and Phased Array Antenna Basics

The book "Array and Phased Array Antenna Basics" [2] deals with the basics of electromagnetic waves and antenna radiation. The first three chapters explain how single antennas work and introduces their sensitivity. Chapter 4 covers the 'standard' linear broadside array. This is a standard phased array build up of antennas equally spaced on a straight line. The chapter studies its performance and adjustable parameters. It is stated that the first side lobe is around -13 dB of the main beam for a phased array. The remaining chapters are about different phased array topologies and discusses their designs and performance. Also how antenna measurements can be performed is written. This book focuses highly on electrical engineering of antennas.

2.3 Smart Antennas

The book “Smart Antennas” [3] introduces array antenna models for different situations. Narrowband processing, adaptive and broadband processing are the main chapters. Chapters 2.1 and 2.2 explain conventional beam forming with a steering vector. This book follows a mathematical approach for the explanation of beam forming.

Chapter 3

Related work

3.1 Radio Astronomy Receivers

In radio astronomy, the universe is studied about solar systems and stars. One of the methods for observation is to receive electromagnetic waves send out by stars. The classical approach for this is to use large dish antennas. However, in the search of higher reception quality now also the use of phased array antennas is studied.

One example of such an antenna is developed by ASTRON, in [6] a phased array antenna telescope demonstrator is described. This demonstrator consists of 256 elements and is used for evaluation of the phased array antenna concept for astronomical research. In this paper a brief description of the thousand element array and the square kilometer array concept is given as well as results from the demonstrator.

The concept consists of tiles with 64 antennas. These tiles first perform analog RF beam forming to create two beams. From there, the result is down converted, digitized and transported over glasfiber to a digital beam former. This digital beam former is able to sum different beams and the result is passed through a Digital Signal Processing (DSP) board. This DSP board performs the calculations needed for evaluations of the radio astronomy signals.

This design represents certain aspects of the problems encountered in Digital Beam Forming (DBF). The digitalization is done with a sample rate of 40 MHz. ASTRON has chosen to equip the DBF with FPGA's. Currently this is a method which is widely used for digital beam forming. [8, 9]

The conclusion is that the demonstrator delivers comparable results with the current 25m reflector telescope. Phased array antennas are concluded to be a well suited technology for radio astronomy telescopes.

3.2 Optical Beam Forming Networks

At the research group “Telecommunication Engineering” of this faculty a beam forming network is developed with the use of laser optics. This is called an optical beam forming network (OBFN) [10], the system uses optical ring resonators (ORRs) to establish a continuously tunable time delay. The OBFN is created by using a binary tree-based hierarchy of ORRs and by using optical combining/splitting circuitry.

In theory this system can beamform broadband signals because it uses time delay rather than a phase shift. Such an approach can be useful in a number of applications. An actual design of an OBFN is designed with one input and 8 outputs, measurements are performed on a stage of 4 outputs. The design is tuned such that three linearly increasing delays are obtained over 1.5 GHz bandwidth. The largest delay value is approximately 0.5 ns (corresponding to 15 cm of physical distance in air) and a delay ripple of approximately 20 ps (6 mm).

3.3 Mobile Satellite Reception

For the reception of satellite signals often dish antennas are used. These antennas need to be setup precisely because of the high angular reception. When pointed directly to a satellite, a signal is received which can be used for television or communication. The setup is fixed and can therefore not be moved, in a mobile situation such high angular reception could be performed with the use of a phased array antenna.

In “Digital Beam Forming Antenna System for Mobile Communications” [8], which is written in combination with [13], the feasibility of a Digital Beam Former (DBF) for satellite communication is evaluated. A DSP system is built for the evaluation of reception capabilities of this system. As a reference a Japanese test satellite is used for the reception of an unmodulated signal. The system is built up of 16 antennas and with 128 KHz sampling ADC. Processing is done with FPGA’s. These FPGA’s are used to implement a DBF using Fast Fourier Transforms. For the creation of quadrature signals a digital local oscillator is used in combination of a FIR filter.

The system shows a succesfull implementation of a beam former processor built up of FPGA units. This systems shows a possible implementation of a DBF with the use of a FFT algorithm. This project also shows an adaptive beam former with a Constant Modulus Algorithm.

3.4 Base Station Communication

In the last decade mobile communication has rapidly grown. For mobile communication parts of the spectrum are used to transmit and receive signals. Because this is getting used more intensively the spectrum occupation grows, one solution can be to separate transmission signals in space.

A possible implementation of such a solution is written in [7]. It handles mobile base communications for ground stations. A cyclic phased array antenna is used with patch antennas and an analog beam former network is used for feeding this array. The goal of the project is to increase coverage radius and reduce transmit power of a base station, the cyclic phased array antenna has a high gain which is steerable and can be used to accomplish these goals.

In a satellite communication system separation in space is introduced in [9], supported by ESA/ESTEC in Noordwijk, the Netherlands. The communication system deals with the problems at the satellite site. A phased array antenna is mounted on a satellite and uses a system for multiple access from the earth. The proposed system features a frequency division multiplexer demultiplexer with a beam forming network.

This system has high specifications, the resources used are limited as only one ASIC is used to handle multiple channels. The proposed solution is a highly integrated system of filters and Fourier transforms.

3.5 The Montium, a coarse-grained reconfigurable processor

The Montium is a processor developed at the University of Twente as part of the Ph.D. thesis of P. Heysters [4]. The Montium can be used as a part of a system on chip. In such a chip, several processors communicate and exchange data with each other. The Montium is therefore also referred to as tile processor. Currently the development of this chip and development tools is handled by Recore Systems [5] which sells this Montium as an Intellectual Property Core (IP core).

The Montium is developed for streaming applications. These applications use streams of data as input and/or output. The architecture and processing units are developed to support this kind of applications. The Montium is equipped with 5 ALU's and 10 memories, these memories have a small AGU unit which can generate memory addresses. A complete description of the Montium tile processor can be found in Appendix C.

Chapter 4

Methods for beam forming

The beam forming explained in Chapter 1 is studied in detail and reference designs to process signals from phased array antennas. The number of digital implementations is limited. In this chapter we restrict to; time delay, phase shift and Fast Fourier Transform.

4.1 Time delay

One method to create a beam is to compensate for the time delay experienced by the different antennas. This time delay can be compensated relatively to the antenna which receives the signal as last one, a reference antenna. The antenna first receiving the signal buffers this signal until the wavefront reaches the last antenna. The time delay (τ_k) in seconds experienced by the antenna for the k^{th} antenna is (equation 1.1):

$$\tau_k = (d/p)sin(-\theta_0) \times k \tag{4.1}$$

k	Index of antenna
d	Distance between antennas
p	Propagation speed of a wave
θ_0	Direction of a wave, positive orientation is clockwise
λ	Wavelength of a signal

Table 4.1: Symbols used

To perform beam forming, individual array signals need to be equipped with a delay line or buffer to compensate for the delay of an incoming wavefront. The compensation is $-\tau_k$ seconds and is calculated for each antenna individually. The outputs of the individual delay lines are summed together and form one beam. The resolution of this method is dependent on the smallest time delay, which can be realized by the delay lines.

In the case the signal is compensated relative to an antenna which does not receive the signal last, this $-\tau_k$ will be negative and a negative delay line should be constructed. Such a delay line should contain future signals and is not feasible. A way to solve this is to add a constant delay equal for all the antennas, which effectively compensates relative to the last receiving antenna again.

Time delay works for wideband signals built up of arbitrary frequencies, not only narrowband signals. This is due to the fact that it compensates for the real experienced differences between antennas. This makes the approach a good solution for processing audio signals, because these signals are typically wideband. The response of beam forming methods also depends on the spacing (d) between antennas. In [2] a limit is calculated for the distance d. It is required that $\frac{d}{\lambda} \leq 1$ should be satisfied otherwise grating lobes appear. In this formula, λ is the wavelength of the signal. Grating lobes are duplicate beams with the same sensitivity as the main beam, but from unwanted directions. The spacing d is taken $\lambda/2$, this spacing generates the most number of nulls without creating ambiguity in the main beam.

As explained in the previous paragraph, beam forming responses are dependent on the locations of the antennas with respect to the wavelength of the signal. Compensating time delay is said to work for arbitrary frequencies, however, the virtual distance between antennas vary. This is a result from changing frequencies and therefore changing wavelengths. The result is that the beam width of the main beam depends on the frequency.

4.2 Phase shift

A signal which has only a small bandwidth, can be simplified by a single sinusoidal signal. This is called the “the narrowband assumption”. For a sinusoidal signal, a momentarily value can be recreated by shifting this signal with the right part of the periodic length. Such part of a period is called phase. Thus, by changing the phase of a signal it can be shifted in time.

To calculate this phase shift a few values are needed, the distance that needs to be compensated and the wavelength of the signal. The wavelength of the signal is on its turn dependent on the frequency of its signal and the propagation speed of the wave in its medium. The wavelength (λ) is calculated by dividing the propagation speed by the frequency of the signal. The frequency will be f and the propagation speed p. In formula form this will be

$$\lambda = \frac{p}{f} \tag{4.2}$$

In figure 4.1 an example phased array is shown. The antennas are separated d meters. A wavefront which is traveling perpendicular to the array is received at all antennas at the same time.

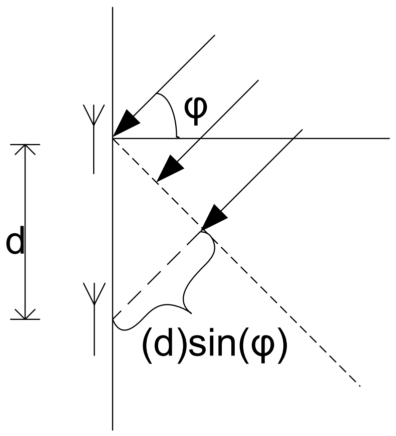


Figure 4.1: Schematic representation of two array elements with a wavefront

When a wavefront is coming from an angle like in the figure, the wavefront and the array form a triangle. At the time the wavefront reaches the upper antenna, the distance from the wavefront to the lower antenna can be calculated with a goniometric calculation:

$$\Delta x = d \times sin(\varphi)$$

The signal at the lower antenna needs to be shifted forward in time. This can be done by giving this signal a positive phase shift. This phase shift is equal to $2\pi \times \Delta x/\lambda$. This phase shift is calculated for a larger array in a linearly fashion for regularly spaced antennas. The distance that needs to be compensated grows linear for the k^{th} antenna, the phase shift (ψ_k) in radials also grows linear. For the total array it becomes:

$$\psi_k = 2\pi(d/\lambda)sin(\theta_0) \times k \tag{4.3}$$

The new symbols introduced in this chapter are summarized in table 4.2. Equation 4.1 and 4.3 will be used in following chapter to compute parameters of the beam forming algorithms.

k	Index of antenna
d	Distance between antennas
p	Propagation speed of a wave
θ_0	Direction of a wave, positive orientation is clockwise
λ	Wavelength of a wave in its medium
τ_k	Time delay
ψ_k	Phase shift
F_s	Sampling frequency

Table 4.2: Updated symbol list

4.3 Butler or FFT transform

The Butler Beam-Forming Array is explained in [1] and can be used to form N beams out of an N-element antenna array. The Butler matrix uses special electronic devices named hybrid junctions and static phase shifters. This Butler matrix is the analog version of the Fast Fourier Transformation (FFT). When the signals of the antennas are digitized, they can be fed into the FFT. A great advantage of this method is that after this processing, the output consists of multiple beams pointed in different directions. Specifically this transformation creates as many beams as input antennas fed into the Fourier transformation.

The Fast Fourier Transform was originally designed for transforming a time signal into a frequency response. The signal induced on an antenna array is also in the form of different frequencies, signals from different directions generate different frequencies when observed in the spatial domain. Let an antenna array consist of elements positioned $\lambda/2$ from each other. In case the signal is induced from the direction perpendicular to the antenna array it induces equal voltage over the antennas, because the signal is the same at each antenna at every moment in time. When a signal is induced in an small angle over the array, the signal is slightly different at each antenna. This results in an ac voltage in the spatial domain. Let the signal be induced in the direction of the array (end-fire), the signal differs $\lambda/2$ between all antenna elements, which results in the highest frequency possible. By using an FFT these frequencies which represent defferent angles can be separated and used as different beams.

The shape of the individual beams from such FFT is fixed and the relative position of the beams is also fixed. These constraints need to be considered when using a FFT as a beam former. In figure 4.2 a response plot shows these beam shapes and fixed positions.

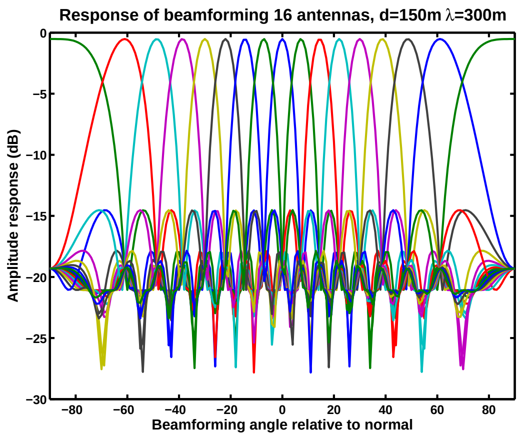


Figure 4.2: Response of phased array processing using fft processing, angle of -90 to 90 degrees in 16 steps

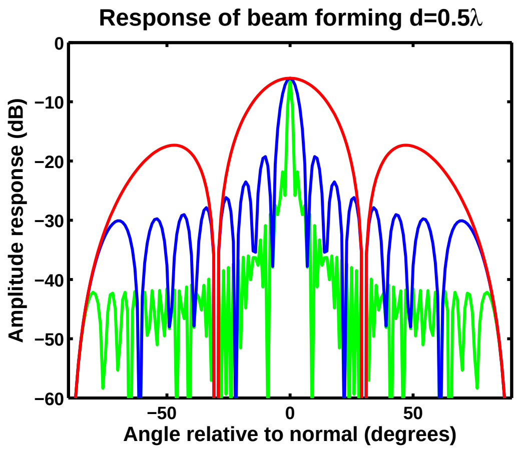


Figure 4.3: Response of an phase array antenna with 4 (red), 16 (blue) and 64 (green) antennas, using complex multiplication to perform beam forming

4.4 Antenna multiplicity

The directional sensitivity of the beams created by beam forming are dependent of the number of antennas used. By using more antennas the antennas receive more information about the direction of the signal, which results in a higher angular resolution. In figure 4.3 the response of three different phased array antenna systems is drawn. One system with four antennas, which has the lowest angular resolution. One with 16 antennas, the intermediate and one with 64 antennas, which has the best relative angular resolution. The -3dB bandwidth is 2 degrees.

4.5 Beam width and side lobes

The main beam width and side lobes are dependent on each other. By using amplitude weighting on the individual antenna elements the shape of the main beam and the side-lobes can be altered. In [2] it is stated that by using a binomial distribution over the elements, the side lobes can be suppressed all together. However, the resulting main lobe then gets wider. For phase shifting with complex multiplications this can be applied to all input signals. The shape of the main beam can be tuned. In case multiple beams are created with individual sets of coefficients, all these beams can be tuned individually. What is more interesting is that there is a trade-off between the width of the beam and the amount of suppression of the side lobs, depending on the used weights. The binomial distribution is one extreme of this. This is because of the uncertainly principle.

In a FFT approach complex multiplications are re-used. The consequence is that all beams get the same shape. Individual beam shaping in a FFT approach is not possible.

The beam width and side lobes also depend on the distribution of the antennas. In this document equal distance between antennas is assumed. Other configurations are possible to change the sensitivity of the array and to change beam width, however, this will not be taken into consideration in this thesis.

4.6 Advanced beam steering

Beam steering can involve a few advanced features, which are treated in [3]. For example one of these features involves dynamic nulling. This is a method to dynamically place the lowest sensitivity in the direction of interference. Adaptive algorithms exist which provide optimal beam steering. A beam steering algorithm is optimal with respect to an optimization criterium. An example criterium could be to produce the highest possible signal to noise ratio. Many of these algorithms work with some sort of digital feedback filter, in which case the optimal beam steerer dynamically changes the coefficients of the beam former.

Such processing can be performed separately of a beam forming algorithm. This document will be restricted to beam forming algorithms.

Chapter 5

Beam forming algorithms

The previous chapter explained how phased array antenna signals can be processed in theory. In this chapter, suitable digital algorithms will be introduced to process these signals on a computer: Time Delay, Complex Multiplication and Fast Fourier Transform.

5.1 Time delay

This method uses processing on the individual signals to create one beam at a time. Through multiple processing stages, multiple beams can be created. The time difference introduced by the different locations of the antennas is compensated with a time shift. After the compensated time shift the signal can be summed and a beam is created.

5.1.1 Algorithm

The approach is to control the delay signals from individual antenna, which can be done with the use of a buffer. The samples are first stored in a buffer and, when time expires, the samples can be read again. The buffer is filled with a rate equal to the sampling frequency (F_s) and the buffer is as long as equation 4.1 prescribes. Afterwards the samples are summed.

The buffer is filled at a fixed rate every $1/F_s$ seconds and the delay length can only be made an integer multiple of this time. To be able to construct all different τ_k values as needed, one could try to increase the sampling frequency F_s. However, in case F_s is already high, this solution is not feasible.

5.1.2 Interpolating

The resolution of the delay elements depends on the sampling frequency. For a typical beam forming application, it should be possible to point in randomly selected directions. This can result in delays which are not an integer multiple of the time steps of the sampling frequency. A naive solution is to round all delays of the antennas to the nearest integer. However this will introduce an error and will have consequences for precision. The response of such a solution is shown in figure 5.4(b).

A solution could be to combine a time delay buffer with interpolation. In this case, interpolation is used for time delays which are not an integer multiple of the sampling time. Interpolation is a technique to calculate values between sample moments. Higher order interpolation can be used to make the interpolation result better. Higher order interpolation uses more sample moments and results in better approximation of the original signal.

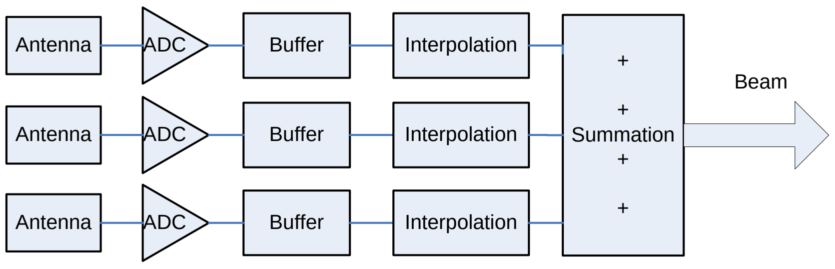


Figure 5.1: Schematic of signal flow with time delay using buffers and interpolating

The Shannon sampling theorem prescribes the minimum sampling frequency needed for signal reconstruction. This should be at least two times the signal bandwidth. For reconstruction the Whittaker-Shannon interpolation formula can be used;

$$x(t) = \sum_{n=-\infty}^{\infty} x[n] sinc\left(\frac{t - nT}{T}\right)$$

This interpolation formula uses a *sinc* function, which is shown in red in figure 5.2. The problem for practical implementation of this formula is that it uses an infinite summation and is therefore in practice not feasible. To create an algorithm which can be implemented an approximation of the reconstruction formula can be used. The approximation is done by summation over a finite interval instead of an infinite interval. The resulting frequency response of an approximation is shown in figure 5.3 for a first, a 32th and a 64th order approximation. As a reference the ideal frequency response of the *sinc* function is shown in red.

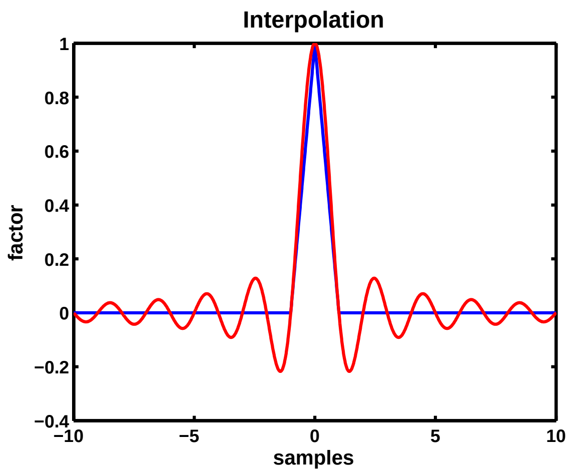


Figure 5.2: Impuls response of linear interpolation (blue) and an ideal sinc(x) interpolation (red)

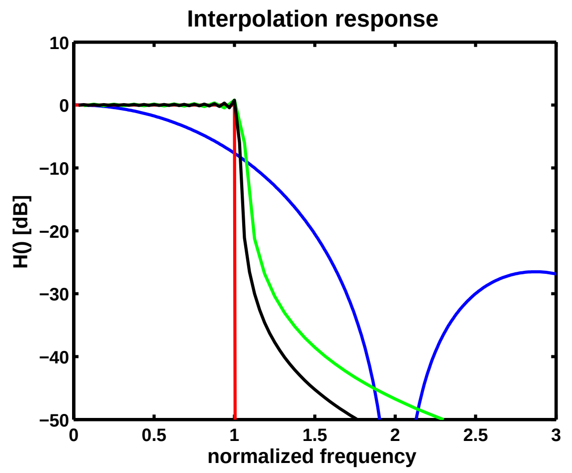


Figure 5.3: Frequency response of linear interpolation (blue), an ideal sinc(x) interpolation (red), a 32th order approximation (green) and a 64th order approximation (black)

5.1.3 Computational complexity

The time delay method consist of two elements, the buffer element and the interpolation element. The buffer element uses memory to store samples. The interpolation element uses only multiplication with a constant and additions, hence no memory is needed. The maximum memory depth required for buffering occurs when the signal travels along side the phased array, the first antenna encountered must store its samples until the wavefront reaches the last antenna. When the antennas are spaced d meters apart, the propagation speed is p and the sampling frequency F_s the maximum memory depth is

$$\frac{d}{p}F_s \tag{5.1}$$

samples. For one antenna, as the direction of the signal can be altered, each outer most antenna will need such a buffer depth. Reaching the middle of the array the needed buffer is half of that.

The number of multiplications required for interpolation is one multiplication for each interpolation coefficient. For N antennas and B beams this formula is;

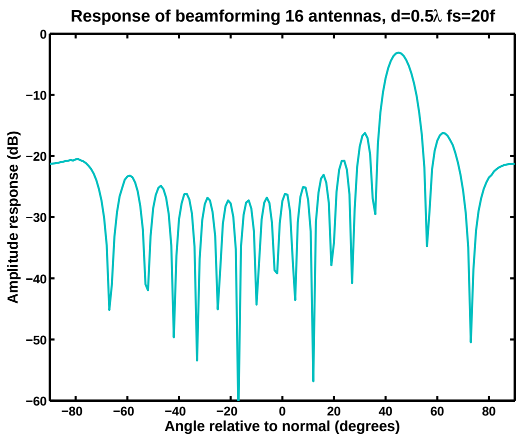
$$(1 \cdot Order + 1) \cdot N \cdot B \cdot F_s \tag{5.2}$$

multiply accumulate instructions per second.

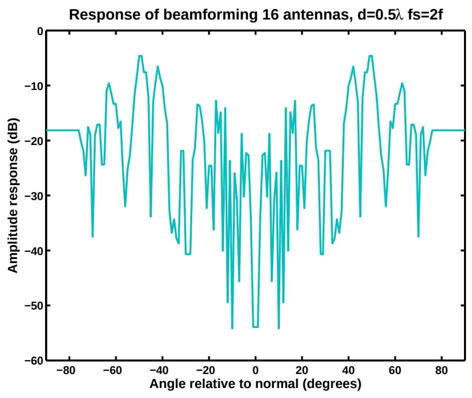
5.1.4 Simulation

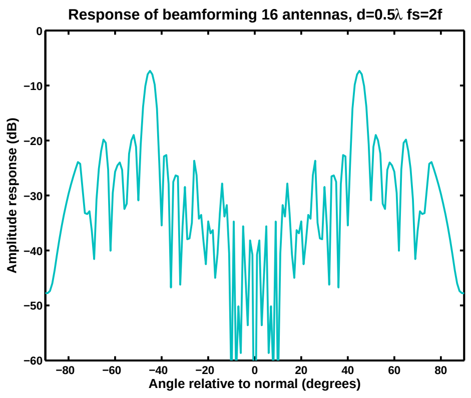
A Time Delay algorithm is simulated in MATLAB implementing an algorithm which rounds sample times and an algorithm which interpolates the samples with a first order interpolation. The simulation projects a beam on the phased array antenna as if it is received from a 45 degree direction of arrival. The parameters of this algorithm are varied with time delays τ_k, which are calculated with equation 4.1 to scan from -90 to 90 degrees. MATLAB simulates sinusoidal wave signals from the antennas. These signals are buffered and the simulation employs rounded time delays and interpolated time delays. Afterwards the signals are summed and the power of this beam is calculated. This calculated power is plot against different beam steering coefficients. The simulated received signal on the antennas is from a constant direction. In figure 5.4(b) the response without interpolation and in figure 5.4(c) the response with first order interpolation is shown.

It can be seen that the response without interpolation shows more noise and smaller suppression of the signal in the band outside 45 degrees. This emphasizes the need of additional processing when rounding errors in buffer delays becomes too large.



(a) Ideal response, Linear interpolation, $F_s = 20\ F_{signal}$





(b) No interpolation, $F_s = 2\ F_{signal}$ (c) Linear interpolation, $F_s = 2\ F_{signal}$

Figure 5.4: Response of an phase array antenna with time delay, beam direction of arrival 45 degrees

5.2 Complex multiplication

The complex multiplication method uses processing on the individual signals of multiple antennas to create one beam at a time. The phase shift introduced by the spacing of the antennas is compensated with a complex multiplication. After this multiplication all signals from one direction are in phase with each other and a cumulative signal can be made by adding all signals together. This is under the assumption that narrowband signals are processed.

5.2.1 Quadrature and in-phase signals

For the algorithm to work, every sample in the time domain needs to be manipulated in phase. The signal gathered by the ADC consists of a real signal, but does not yet contain phase information in its samples. This can be seen when a momentarily value is studied. When, for example, a real value from the ADC is sampled, its value could be '2'. With this information it is not possible to know what the phase of a sinusiodional is.

One way to represent complex signals which can include phase information is using quadrature signals. Together with a real signal, an extra signal is created which lags 90 degrees in phase. For example, when together with the real '2' an 90 degrees off '1' signal is present, they represent a phase of

$$tan\left(\frac{1}{2}\right) = 27$$

degrees. So to store phase information in samples a secondary signal is needed. This signal is called a quadrature signal. In the analog domain, this signal can be created with the use of a local oscillator (LO); at one side the direct LO signal and at the other side a 90 degrees shifted LO signal. These signals are multiplied with the received signal and because of this they are called in-phase and quadrature signals.

5.2.2 Hilbert transformer

A Hilbert transformer can also be used to construct a quadrature signal with the in-phase signal as input. This is done by shifting positive frequencies -90 degrees and negative frequencies 90 degrees. In [12] a transformation is made from the frequency domain to the time domain. The formula which describes the Fourier relation is;

$$\mathfrak{F}\left(\frac{1}{\pi t}\right) = -j \cdot sgn(f) \tag{5.3}$$

where

$$sgn(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

The right part of equation 5.3 represent the Hilbert function in the frequency domain. The positive frequencies get a -90 degrees shift through the multiplication in the frequency domain with $-j$, while the negative frequencies get a multiplication with j in the frequency domain.

The Fourier transform of the Hilbert function consists of imaginary values only. In the time domain, the function $1/\pi t$ can be approximated by a set of sine waves. By using this time domain function of the Hilbert transformer, it is possible to implement the Hilbert function using a Finite Impuls Response (FIR) filter. Filter coefficients can be calculated by MATLAB and an example of an impulse response is shown in figure 5.5.

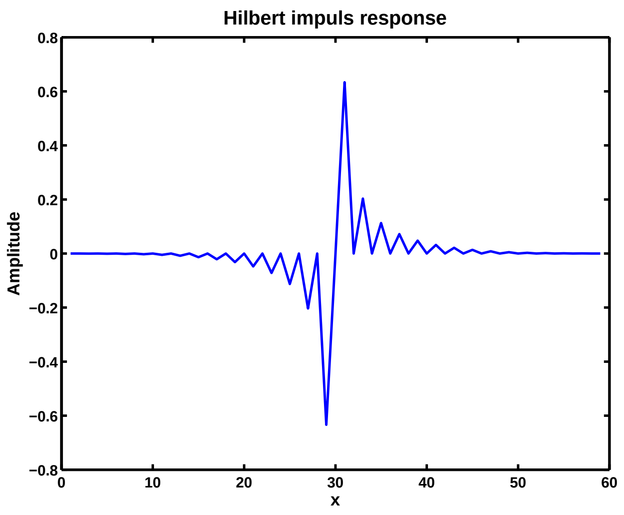


Figure 5.5: Impulse response of a Hilbert filter

The Hilbert filter is approximated using a FIR filter, which introduces extra calculation requirements for the algorithm to finish. The number of calculations required by a FIR filter depends on the number of coefficients used. For every coefficient a multiply accumulate instruction needs to be executed. In general the effects of using more coefficients for FIR filter design are: the delay of the signal increases, the approximation improves and more calculations are needed.

An example Hilbert FIR filter is designed with MATLAB. It is a 16th order filter which has 17 coefficients. The frequency response is shown in figure 5.6.

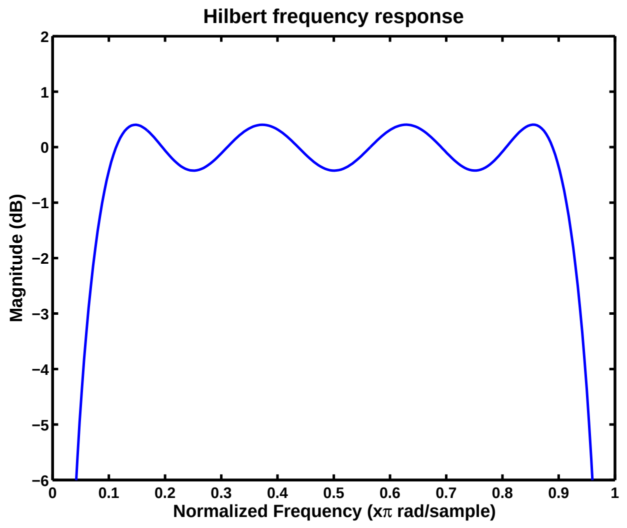


Figure 5.6: Frequency response of a Hilbert FIR filter

As seen in figure 5.5, half of the coefficients are zeros. In an optimal implementation, multiplications where these zero-coefficients are involved can be skipped as they do not influence the result, resulting in only half the multiply accumulate (MAC) instructions as normal. The example 16th order filter can with some added control be processed with 8 MAC instructions.

The signal which travels through the Hilbert filter experiences a group delay of half the filter length. This delay is introduced in FIR filter design, the FIR filter applies a convolution with the coefficients. The coefficients represent the impulse response of a desired frequency response. Because this impulse response is not causal, this impulse response is shifted in time over half the filter length.

The delay must also be given to the in-phase signal. To accomplish this, a group delay block is introduced in the signal path of the in phase signal.

5.2.3 Algorithm

The main algorithm consist of the multiplication of the in phase and quadrature signal (from now on the combination of these signals is called a complex signal) with a phase shifting vector (ρ_k). This vector is given a magnitude of one and a phase which is based on formula 4.3.

$$\rho_k = 1 \cdot e^{j \cdot \psi_k} \tag{5.4}$$

With the complex exponent this results in a complex vector. The signal needs to be multiplied with this constant complex vector (ρ_k). A complex value is a pair of real values. For a complex multiplication four multiplications of real values are processed.

A schematic overview of the total Hilbert filter + Complex Multiplication system is given in figure 5.7.

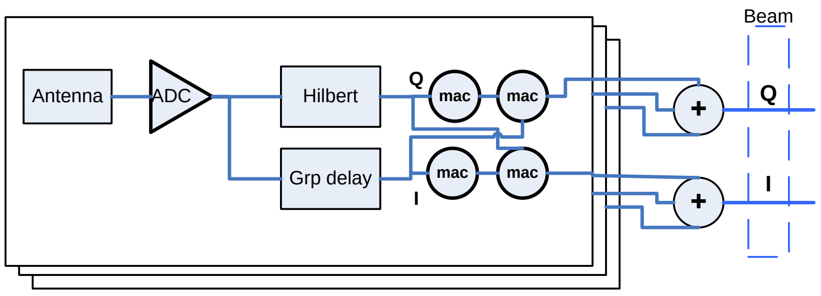


Figure 5.7: Schematic signal flow with complex multiplication

5.2.4 Computational complexity

The previous description consists of two steps. First, for each antenna a Hilbert process is started to create a complex signal, second, for each beam the complex multiplication has to be performed. The first step uses the FIR filter order (H) divided by 2 MAC instructions for each sample moment. This is done for all N antennas. The second step uses four MAC instructions for all N antennas, for each B beams and for each sample moment (F_s). In formula form this becomes

$$(H/2 \cdot N + B \cdot N \cdot 4) \times F_s \tag{5.5}$$

multiplications per second.

5.3 Fast Fourier transform processing

With the use of a Fast Fourier Transform (FFT) phased array signals can be processed in a single algorithm to multiple beams. The FFT therefore re-uses intermediate calculations. It is more efficient than the phase shift method.

5.3.1 Quadrature and in-phase signals

The FFT processing also requires a complex signal. This is because the FFT is an optimisation of the complex number multiplication and requires phase information. If this is not done, a FFT can not separate the negative and positive frequencies. These negative and positive frequencies form the left and the right intercept angles of the phased array antenna.

5.3.2 A spatial Fast Fourier Transform as beam former

The FFT processes all the antenna signal sampled at a specific point in time. This method creates N beams from an array of N antennas. The FFT algorithm computes the discrete Fourier transform (DFT) of a signal ($x[n]$), the equation of the DFT is:

$$X[k] = \sum_{n=0}^{N-1} W_N^{nk} x[n] \tag{5.6}$$

where

$$W_N^{nk} = e^{-j\frac{2\pi}{N}\cdot nk} \tag{5.7}$$

A schematic representation of the algorithm is presented in figure 5.8.

5.3.3 Computational complexity

In the case of N antennas and a sampling frequency F_s, to create the same number of beams as antennas the FFT algorithm needs [4]

$$4 \cdot (N/2) \cdot log_2(N) \cdot F_s \tag{5.8}$$

multiplications per second.

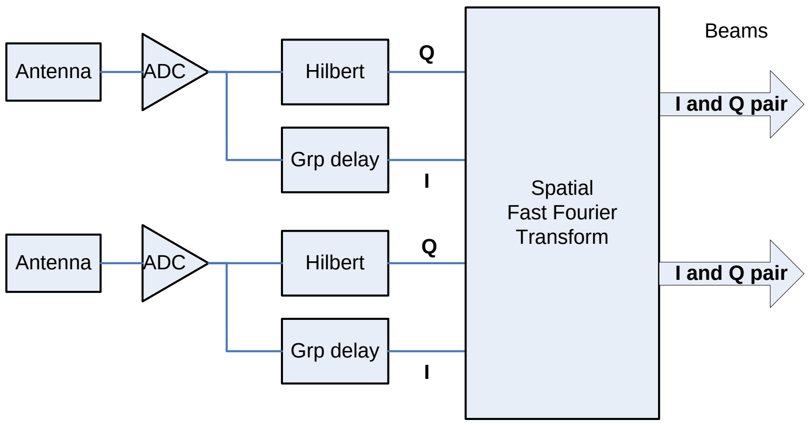


Figure 5.8: Schematic signal flow with FFT processing

5.4 Comparison of algorithms

The algorithms explained in this chapter use different amount of resources from a processor. To give an impression about the relative computational capacity needed by the three algorithms, a figure is made. Figure 5.9 shows such a comparison. On the vertical axis the number of multiply accumulate instructions needed on each sampling moment is given. On the horizontal axis the number of input antennas is given. The methods of complex multiplications and FFT uses a Hilbert pre-filter of 16^{th} order.

An example, the FFT approach with 128 antennas takes about $4 \cdot 10^3 \cdot F_s$ MAC instructions. This means that 4000 MAC instructions have to be executed each time the antenna ADCs take a new sample.

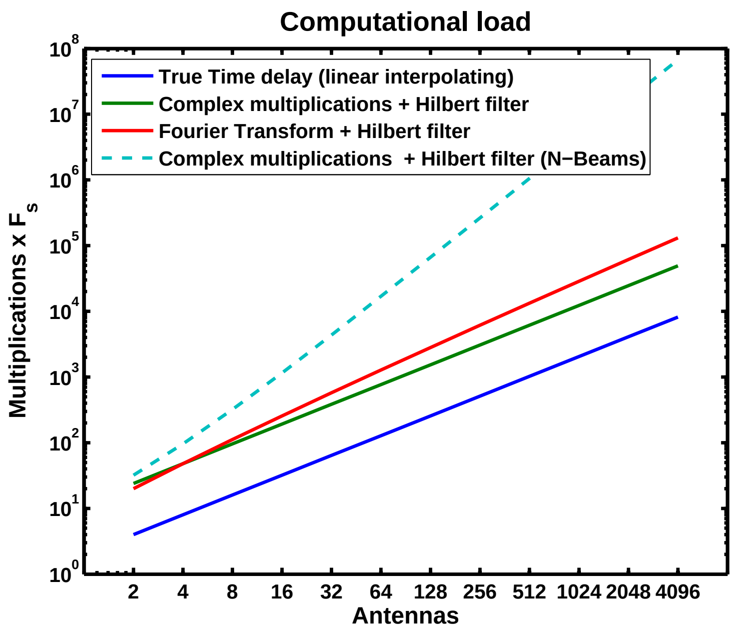


Figure 5.9: Computational complexity of different algorithms with respect to number of antennas. For complex multiplication processing and FFT processing a Hilbert filter of 16^{th} order is used.

Chapter 6

Testplatform design

6.1 Introduction

The test platform is built on a development board of Xilinx [17]. This development board has a FPGA, a number of input, output peripherals and is equipped with hardware to build an embedded system. The FPGA is a Virtex II Pro, this FPGA has next to the standard FPGA slices also block RAMs, multiplier slices and two PowerPCs. A PowerPC is a general purpose processor. For a beam forming testplatform audio signals are taken to be evaluated. Audio signal can be made with a predetermined spectrum and with the use of microphones these signals can be received. The received analog signal is converted with analog to digital converts (ADC) to a digital signal. These digital signals lines are connected to the FPGA.

6.2 Development

Developing a system on the development board can be done with the use of the Xilinx Platform Studio (XPS) software [18]. The studio delivers support for a hardware project with multiple software projects. The project is usually started with a “base system builder” wizards [16] which gives a foundation for the rest of the project. The wizard needs a “User Peripheral Repository” which gives a description of the development board. The output of the wizard consists of a complete (compilable) project which can be downloaded for evaluation.

This package of software delivers multiple tools for designing and debugging a hardware software integrated system. The XPS software is used to create a design for beam forming with the Montium and the ADCs. The two most used software programs in this package are;

Impact

Impact is a tool that can be used to program the development platform. This tool support all methods for configuring and handles the file translation between different formats. The board can be configured in a number of ways, for example, it can directly be programmed through the embedded platform USB connection, it can be configured with the use of the onboard flash PROM or it can be configured with the use of a Compact Flash card. Downloading software is done using Boundary Scan (IEEE 1149.1 /IEEE 1532).

Integrated Software Environment

The "Integrated Software Environment" (ISE) is the environment used by XPS to synthesize hardware designs. The hardware designs (for example VHDL descriptions) are managed by XPS and are compiled with this program. The input is a hardware description language and the output are net lists and place-and-route information.

6.3 System design

The system design is shown in figure 6.1, main parts are the Montium, the ADC interface and the PowerPC. The Montium TP is synthesized from its VHDL source and configured into FPGA space. For interfacing with the ADCs an interface is build in VHDL, this interface is described in appendix A. This VHDL interface is synthesized and configured into the FPGA next to the Montium.

XPS defines two kinds of busses: a Processor Local Bus (PLB) and an Onboard Peripheral Bus (OPB). The PLB is the (memory) bus of the PowerPC, the OPB is in turn connected to the PLB. The Montium and the ADC interface are interconnected with the OPB.

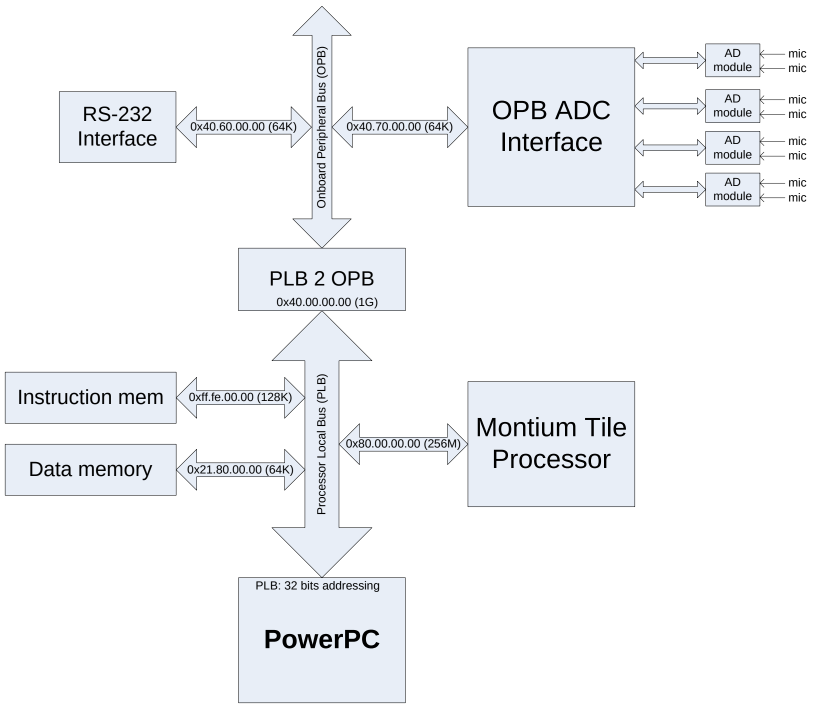


Figure 6.1: Hardware design of the testplatform

6.4 Beam former data flow

The testplatform design uses audio signals as source and applies beam forming on these audio signals. These input signals are made with the use of 8 microphones and are converted with 8 ADCs. The ADCs are embedded on an ADC-module, a single module consists of two ADC from National Semiconductors type ADCS7476 [19]. The modules digital signal lines are connected to the development board and are connected to FPGA pins. From here the processing is done on the FPGA chip, the data flow of the different processing stages are shown in figure 6.2 and 6.3 for the different methods.

The data flows are annotated with processing algorithms (above the blocks) and mapping information (beneath the blocks). The blocks are given a functional description.

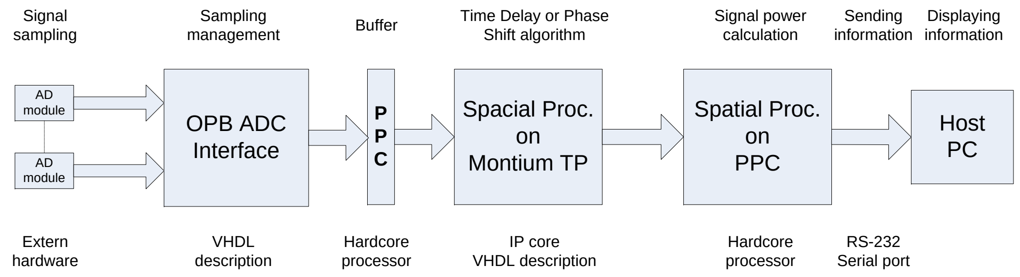


Figure 6.2: Data flow of a single stage (Time Delay or Phase Shift) process

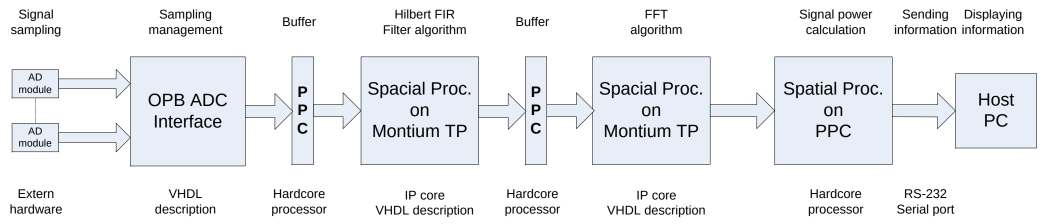


Figure 6.3: Data flow of a double stage (Hilbert and FFT) process

Chapter 7

Mapping beam forming algorithms to reconfigurable hardware

7.1 Introduction

This chapter will show how the previously explained algorithms have been implemented on the Montium processor. First the Time Delay algorithm implementation is explained. Then the implementation of a Hilbert filter together with the Complex Multiplication. Finally it is shown how the results of the Hilbert filter can be used for the Fast Fourier Transform.

The algorithm implementations are made for a system with 8 input antennas/channels.

7.2 Time Delay

The Time Delay algorithm consists of buffering, interpolation and summation. The following implementation is for a system which consists of 8 antennas, also called channels in this text. A trade off is made which combines interpolation with a feasible amount of processor power. The implementation made is an algorithm which uses a linear interpolation. Linear interpolation stores two samples in the registers and gives weights corresponding linearly to the distance of the required time and the sampling time. The resulting frequency response of this implementation is shown in figure 5.3.

The Montium receives samples, one sample $X_{k,i}$ for each channel (k) for each sampling time (i) and outputs a single value for each sampling time. The output sample Y_i is the calculated beam.

In the following sections these steps are explained. The steps are for a single channel. The Montium is a parallel machine and the steps can be performed for 5 channels in parallel; each Processing Part handles one channel at a time.

Buffering

Buffering is done with the use of a cyclic buffer. This buffer uses the memories of the Montium. Each channel is given its own memory, because the Montium is equipped with 10 memories it can serve 10 channels. A cyclic buffer is formed by incrementing the pointers with a modulo function. Two pointers are introduced, one writepointer which stores the location where the buffer is filled and one readpointer where the buffer is read.

The pointers are stored in registers of the ALU. When a sample is calculated these pointer are incremented. In each iteration, incoming samples (X) are stored at the position of the writepointer and a sample at the readpointer is forwarded to a register (R).

Interpolation

The sample stored in the register is multiplied with an interpolation coefficient and stored in a register, mathematical:

$$A = R[n] \cdot C_{interpolation}[1]$$

In the CDL[1] code the interpolation coefficients $C_{interpolation}[1]$ and $C_{interpolation}[2]$ are called `FIRST` and `SECOND`, respectively. In CDL code the interpolation is:

```
// Multiply NEWREGm1 (c1) sample with FIRSTm1  coeff. (b2)
alu p3c1 fmul p3b2 sadd p3es -> p3ws
```

In the next clock cycle the previous sample $X[n-1] = R[n-1]$ is used and multiplied with the second interpolation coefficient. In the same clock cycle this value is added to the previously calculated value A:

$$I = R[n-1] \cdot C_{interpolation}[2] + A$$

```
clock
// Multiply OLDREGm1 (b1) sample with SECONDm1 coeff. (c2)
alu p3b1 fmul p3c2 sadd p3es -> p3ws
```

Now an interpolated sample I is calculated for this channel. This sample ($I_{k,i}$) is used for summation.

[1]Montium source code

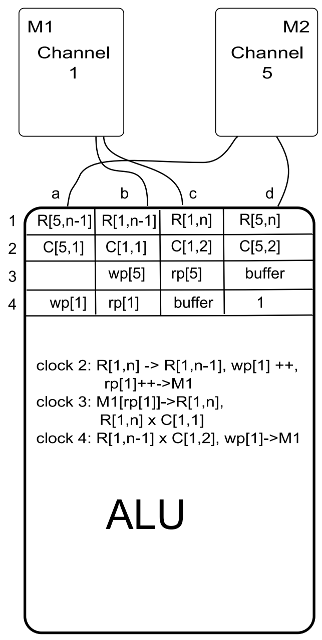

M1
Channel 1
M2
Channel 5
a
b
c
d
1
2
3
4
R[5,n-1]
R[1,n-1]
R[1,n]
R[5,n]
C[5,1]
C[1,1]
C[1,2]
C[5,2]
wp[5]
rp[5]
buffer
wp[1]
rp[1]
buffer
1
clock 2: R[1,n] -> R[1,n-1], wp[1] ++, rp[1]++->M1
clock 3: M1[rp[1]]->R[1,n], R[1,n] x C[1,1]
clock 4: R[1,n-1] x C[1,2], wp[1]->M1
ALU

Figure 7.1: Memory and ALU register mapping

Summation

Summation is done by adding all samples $I_{k,i}$ from the different processing parts together to form the resulting output sample Y_i. Summation is done with the use of the east-west connections of the ALUs. The summation of different processing parts is done in parallel with interpolation. The calculated value A becomes then a cumulative value for 4 channels and is calculated by adding $I_{k,i}$ for $k = 0$ to 3. In the second stage (channels 4 to 7) this value is added with $I_{k,i}$ for $k = 4$ to 7.
In the first stage;

$$Y_i = \sum_{k=0}^{3} Y_{k,i}$$

and in the second stage;

$$Y_i = \sum_{k=4}^{7} Y_{k,i} + Y_i$$

The output sample Y_i is a sample for the beam calculated.

Clock cycles

The algorithm on the Montium uses 10 cycles for one output sample[2]. This can only be seen in the source code, provided in appendix B. For the multiplication 4 cycles are used, 4 more for pointer updating and 2 for input, output communications.

[2]The Montium processor is a complex processor which can do a lot in parallel. The explanation of the mapping is not a one to one mapping in clock cycles. Because of performance enhancement, re-ordering and re-scheduling of processing is performed.

7.3 Hilbert filtering

The complex multiplication implementation is made with a preprocessor which filters the incoming data. This filter is a Hilbert FIR filter which generates information about the phase of the signal received. This Hilbert filter is implemented as a FIR filter in an optimized way. And is capable to process multiple channels in parallel.

The Montium receives as before a sample ($X_{k,i}$) for each channel (k) every sampling time (i). These samples are filtered and the Hilbert stage will output two values: the filtered quadrature (Im) value and the in-phase value (Re).

The number of samples which have to be calculated is preferred to be a power of 2. This results in an efficient mapping on the Montium AGUs. The implemented Hilbert filter is a 16th order filter. The filter coefficients (C_j) are calculated by MATLAB. This results in 17 coefficients of which 9 (rounded to the nearest 16 bits number representation) are zero, 8 coefficients remain to be calculated by the processor. The frequency response of a 16th order filter is given in figure 5.6.

Buffering

First the Hilbert filter buffers the incoming samples X_i in a cyclic buffer (which is a modified version of [14]). The algorithm is made for 8 input channels in the spatial domain which are stored in the first 8 memories of the Montium. These buffers are split in two sections of which the odd and even temporal index (i) of $X_{k,i}$ are separated. In the current implementation, these odd and even samples are split up in two partitions of the memory, each 8 samples long.

Filtering

The Hilbert coefficients (C_j) are stored in an additional memory, memory 9, of the processor. This memory is used by all processing parts to multiply the incoming data with.

$$Im_{k,i} = X_{k,(i-j)} \cdot C_j + Im_{k,i} \quad \text{for j} = 1 \text{ till } 8$$

Group delay

This implementation also generates a value for the in-phase signal. This value is calculated by addressing the cyclic buffer half the order of samples back.

$$Re_{k,i} = X_{k,(i-O/2)} \text{ for O = Order of filter}$$

Clock cycles

The algorithm on the Montium uses 21 cycles for one sampling time. For the multiplication $2 \cdot 8$ cycles are used, 2 for overhead of pointers and 3 for input, output communications.

7.4 Complex Multiplication

The output of the Hilbert filter is stored in registers. The complex multiplication (CM) continues processing with these values. It should multiply the complex signal pairs (Im and Re) with complex coefficients. It does this only to the extend that it calculates the resulting real values (Re). The result of the imaginary value (Im) is not calculated. This is done because the post-processing stage, power calculation, does not require a complex signal.

Clock cycles

The algorithm on the Montium uses 7 cycles for one sampling time. For the multiplication of 8 complex signal pairs with 8 coefficients 5 cycles are used, 1 for overhead and 1 for input, output communications.

A complex signal pair multiplication requires 4 clock cycles on one processing part, now only 2 clock cycles are used. Because one processing part handles two channels, the processing parts need 4 clock cycles. For the final summation of all 8 signals the accumulated results of the processing parts are added in a final summation clock cycle.

7.4.1 Results

The test setup is made with the development board (chapter 6) and the Montium configuration for CM.

The test is made with a signal generator, generating a sinus with a RMS value of $126mV$ (figure 7.3(b)). This signal is split in 8 and fed through the inputs of the ADC and is beam formed in the Montium processor. The PowerPC calculates the power received. This test signal should result in a power of

$$20 * log(126mV \times 8) = 60.069dBmV$$

In figure 7.3(a) the output signal in shown and a value of 59.925 in calculated by the PowerPC. This result shows a large resemblance with what is expected.

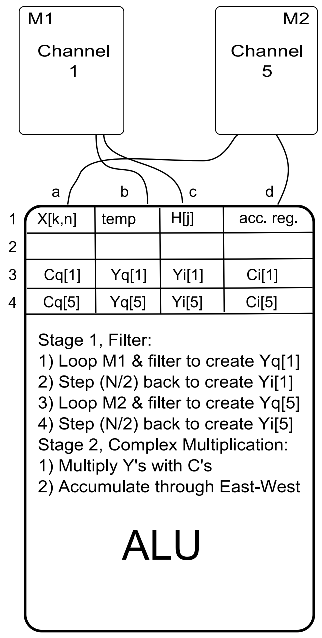

M1
Channel 1
M2
Channel 5
a
b
c
d
1
2
3
4
X[k,n]
temp
H[j]
acc. reg.
Cq[1]
Yq[1]
Yi[1]
Ci[1]
Cq[5]
Yq[5]
Yi[5]
Ci[5]
Stage 1, Filter:
1) Loop M1 & filter to create Yq[1]
2) Step (N/2) back to create Yi[1]
3) Loop M2 & filter to create Yq[5]
4) Step (N/2) back to create Yi[5]
Stage 2, Complex Multiplication:
1) Multiply Y's with C's
2) Accumulate through East-West
ALU

Figure 7.2: Memory and ALU register mapping

7.5 Fast Fourier Transform

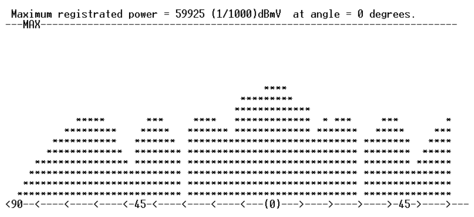
The Fast Fourier Transform (FFT) implementation uses the results of the Hilbert FIR filter as input data. The stream of $Re_{k,i}$ and $Im_{k,i}$ (complex) values of the Hilbert filter are first stored in a temporal buffer in the PowerPC (see section 6.4). When the buffer is filled, these complex values are streamed toward the FFT algorithm.

Implementation

The implementation of the Fast Fourier Transform is supplied by [5] as a binary configuration file. The implementation calculates the FFT of the input samples and does reordering of the output samples afterward. Reordering is a technique that enables a logical output order of the samples of the FFT. The core of a FFT routine delivers its output samples in a bit-reversed order.

The implementation uses 29 clock cycles in the Montium simulator (see section C.4). The implementation uses;

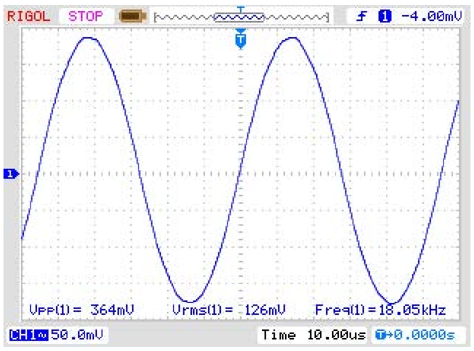
$$\left(\frac{N}{2}+2\right) log(N) \tag{7.1}$$



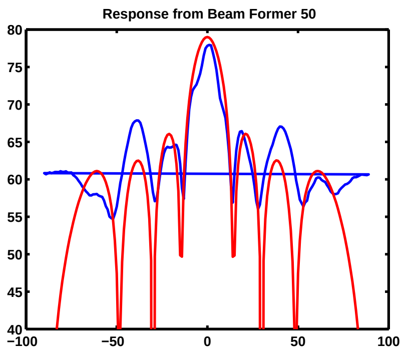
```
 Maximum registrated power = 59925 (1/1000)dBmV  at angle = 0 degrees.
---MAX-----------------------------------------------------------------------

                                            ****
                                        *********
                                       *************
            *****       ***     ****   *************  * ***     ***        *
          *********    *****   ******* ************* *******   *****     ***
        ***********   *******  ***************************** ********   ****
       *************  ******** ***************************** ********  *****
     **************** ******** ***************************** ******** ******
    ************************** ***************************** ***************
   *************************** ***************************** ***************
  **************************** ***************************** ***************
<90--<----<----<----<-45-<----<----<----<---(0)--->---->---->---->-45->---->---
```

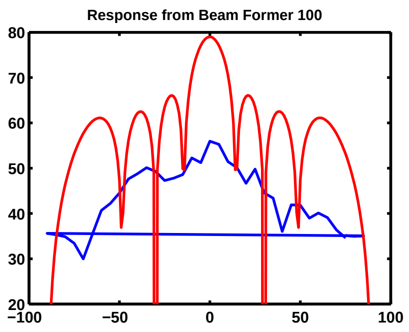
(a) Output spectrum of test setup, hyperterminal output



(b) Input signal for test setup, generated with a function generator



(c) Output spectrum of feed with signal generator (other setup), Matlab output



(d) Output spectrum of an audio source at a distance of around 2 meters, Matlab output

Figure 7.3: Response of an beam former implementation with Complex Multiplications

clock cycles for the FFT algorithm. In this case it should use $6 * 3 = 18$ cycles, the additional cycles are used for reordering.

7.6 Mapping results

A comparison of the different implementations is given. Table 7.1 shows the clock cycles minimum needed for calculating all multiply accumulate instructions. These are based on the formulas in chapter 5 and are the ideal values. The table shows the clock cycles needed in the implementations as given in this chapter.

The number of ideal clock cycles used are calculated based on the assumption that every processing part handles two channels. In the implementation this results in the most efficient mapping and a fair comparison of ideal and implemented values are possible then.

Algorithm	Ideal	Implemented	Remarks
Time Delay	4	10	1st order interpolation, 1 beam
Phase Shift	20	29	$Re/$in-phase output only, 1 beam
Phase Shift	24	33	Complex output, cycle usage is an estimation
Hilbert FIR	16	21	
FFT	12	29	Complex output, N beams, does also reordering in implementation

Table 7.1: Clock cycle usage of algorithms

Chapter 8

Applications

A beam former can be designed for different application purposes. For these different purposes, the design parameters differs considerably. First an overview of different Montium implementations with their operation speed will be given. Then a number of possible scenarios are given in which beam forming can be applied.

8.1 Montium processing throughput

The current Montium implementation on the BCVP[1] reaches clock speeds of around 6 MHz. For some applications mentioned in this chapter this is not sufficient high. In the Annabelle[2] chip clock speed of 25 MHz (worst case) to 100 MHz (best case) are reached. For this chip a 100 MHz Montium clock speed will be assumed. A future implementation of the Montium with pipelining could reach clock speeds till 500 MHz, this is hyphotetical. The current Montium features east-west connections which spread throughout 5 ALU's. In chip design, such long combinational paths slow down maximum clock speeds. In an improved design of the Montium, which handles these connections otherwise, such higher clock speed can be expected.

A Montium features 5 ALUs which can do 1 MAC/ALU. With these implementations a throughput in Million Multiply Accumulate (MMAC) instructions per second is given, in platforms with multiple Montiums, the total throughput in MMAC is given:

[1]BCVP is a first generation concept validation platform and houses three Montiums
[2]Annabelle is equipped with four second generation Montiums

Clock speed	MMAC·s^{-1}	Platform
6 MHz	90	BCVP
20 MHz	100	Xilinx Virtex II Pro implementation
100 MHz	2000	Annabelle chip
500 MHz	2500	Future expected implementations

In the following sections three scenarios are given and a estimation is made concerning the required processing speed.

8.2 Speech beam forming

In a situation of a room with multiple speakers and possible interferer sources a beam former system can be useful. To listen to individual speakers one beam can be used. This system can apply beam forming to separate different sources and only give one source amplification. The system can be equipped with a second beam to scan for interference.

For speech an upper frequency of 4 kHz is used. A case is sketched to perform beam forming with two beams. Speech occupies a large bandwidth from around 400Hz till 4kHz. For such a signal, a system with a Time Delay (TD) algorithm would be suitable because it can handle wideband signals.

Design parameters in such a system could be as follows:

Method	TD	
Maximum frequency	4	kHz
Sample rate	8	kHz
Reception	16	microphones
Interpolation	32	order
Number of beams	2	beam

The interpolation order is set to 32 to obtain a sufficiently low noise level. The reception equipment with the number of microphones depends on the room and the number of speakers. In this situation a processing requirement is given with the use of equation 5.2 and results in:

$$(1 \cdot 32 + 1) \cdot 16 \cdot 2 \cdot 8\text{k} = 8448\text{k MAC} \approx 9\text{MMAC} \tag{8.1}$$

instructions per second.

The resulting processing requirements for this scenario should be possible to implement on the current BCVP.

8.3 Quality audio beam forming

In a situation of a live concert for example, a beam former system which separates audio sources from different directions can be designed. Such audio system will be given the full frequency range of the human ear of around 20 kHz. For audio storage devices often a sampling frequency of 44.1 kHz is used. This sampling frequency will also be used for this proposed system.

Because of the use of a wideband audio signal, again the use of the Time Delay (TD) algorithm is preferred. In a live concert, multiple audio sources can be distinguished and processed for recording purposes. For such a situation the system will be given 32 beams and a sufficient interpolation order.

Method	TD	
Maximum frequency	20	kHz
Sample rate	44.1	kHz
Reception	32	microphones
Interpolation	32	order
Number of beams	32	beam

In this situation a processing requirement is given with the use of equation 5.2 and results in:

$$(1 \cdot 32 + 1) \cdot 32 \cdot 32 \cdot 44.1\text{k} = 1490\text{MMAC} \tag{8.2}$$

instructions per second.

The resulting processing requirements for this scenario has increased around a factor 150 with respect to the previous situation. These requirement should be possible to implement in the Annabelle chip. The algorithm is suitable to be divided over more processors.

8.4 Radar beam forming

For a radar application electromagnetic waves are used to recover objects. Radar sends out (high) power waves and objects which are present in these power waves reflect power back to the radar antenna. By processing the signals received back at the antenna, information about the size, distance and speed of the objects can be revealed.

Beam forming can be used in this situation to separate different signals from different directions. Radar applications use high frequency signals, in the order of GHz, for output waves. The signal received back (from the same order of GHz) is converted down to a suitable frequency for digitalisation. The required sampling frequency will then be in the order of MHz. For a high spatial resolution a high number of antenna elements is used. For such a system an algorithm like complex multiplications or Fast Fourier Transform can be used.

Method	CM	
Maximum frequency	20	MHz
Sample rate	40	MHz
Reception	256	antennas
Hilbert filter	16	order
Number of beams	1	beam

In this situation a processing requirement is given with the use of equation 5.5 and results in:

$$(16/2 \cdot 256 + 1 \cdot 256 \cdot 4) \times 20M = 61440\text{MMAC} \tag{8.3}$$

instructions per second.

The resulting processing requirements for this scenario are very high. For this 31 Annabelle chips or 25 future Montiums (500MHz clock speed) are required. When adopting a system which supplies an Annabelle chip for each antenna, each such a chip requires 240MMAC per antenna for beam forming. This could be done with a single Montium (500 MMAC$\cdot s^{-1}$) from one of the four Montiums in an Annabelle chip.

Chapter 9

Conclusion and Recommendations

9.1 Conclusion

Different mathematical methods are suitable for implementation on digital processors. The Montium processor is used in the verifications of these algorithms and can be configured efficiently. The implementation on the Xilinx development board shows that the Montium processor can apply these algorithms in real-time in an actual system. The Montium in a single setup performs well for audio applications. Also an advantage is that the algorithms do not use complex branch instructions but are mainly forward filter calculations.

In the Complex Multiplication method the Hilbert filter was originally not predicted. This extra filter greatly increases the amount of processing required by digital processors.

In chapter 8, the scenario of a radar beam former is described. This scenario puts high requirements for a beam former. Because of the low speed of the Montium, multiple processors are needed. For radar designs the current implementation of the Montium can be used for beam forming. A system with multiple cores or chips is suitable for processing.

The Montium's full design features a logic function array in its ALUs. This logic function array is currently not used in the methods of Time Delay or Complex Multiplications. The FFT method can use scaling in between different stages, this scaling is done with the use of the logic function array. The Montium can be adapted for specific purposes, in applications where no FFT processing is needed the logic function array can be left out of the design. This could save space in a final design.

9.2 Recommendations

9.2.1 Partial reconfiguration

The Montium is currently programmed with complete sets of configuration code. When the processing is switched to another algorithm the complete program memory is rewritten. The Montium also has the option for partial reconfiguration [15], which could be used to switch faster between different algorithms.

The option of partial reconfiguration also makes another change possible. For the Time Delay method, beam steering delay parameters and interpolation coefficients are in registers. Updating these coefficients takes two additions on the ALU at this moment. The coefficients are stored in registers to support multiple channels on one processing part. When these delay parameters are moved to the Montium AGUs, the AGU can take care of the additions and saves two clock cycles. In this new situation, the Montium will need to be reconfigured, by a master, with new coefficients.

9.2.2 Scalability

For high demanding beam forming applications and the possibility to support further post-processing a system with a lot of computational power is needed. Within the CRISP project a chip with 64 Montium will be developed. In the design of such a chip, to be suitable for a radar beam forming application, it will have to provide high speed input/output and interconnect, as well as high speed inter-processor connections.

The speed of the processors in such a chip should be suitable for high speed computations. A speedup of the current Montium clock is preferred above the possibility to be able to use the current east-west connections of the Montium ALUs. The east-west connections are not often used in the implementation. This can be a starting place for optimization.

Bibliography

[1] SKOLNIK, M.I., *Introduction to Radar Systems,* 3rd ed. New York: McGraw-Hill, 2001

[2] VISSER, HUBREGT J., *Array and Phased Array antenna basics,* John Wiley & Sons, Chichester, 2005

[3] GODARA, LAL CHAND, *Smart Antennas,* Boca Raton: CRC Press, 2004

[4] PAUL M. HEYSTERS, *Coarse-Grained Reconfigurable Processors, Flexibility meets Efficiency,* Ph.D Thesis, 2004

[5] *Recore Systems,* P.O. Box 77, 7500AB Enschede, The Netherlands

[6] J.G. BIJ DE VAATE, S.J. WIJNHOLDS AND J.D. BREGMAN, *Two Dimensional 256 Element Phased Array System for Radio Astronomy,* Technical report, www.astron.nl/tl/thea/publications, Astron, The Netherlands

[7] C. ALAKIJA AND S.P. STAPLETON, *A Mobile Base Station Phased Array Antenna,* Simon Fraser University, Canada, IEEE Wireless Communications, June 1992

[8] I. CHIBA, R. MIURA, T. TANAKA AND Y. KARASAWA, *Digital Beam Forming Antenna System for Mobile Communications,* IEEE AES Systems Magazine, September 1997

[9] T. GEBAUER, H.G. GÖCKLER, *Channel-individual Adaptive Beamforming for Mobile Satellite Communications,* IEEE Journal, Vol. 13 No. 2, February 1995

[10] L. ZHUANG, C.G.H. ROELOFFZEN, R.G. HEIDEMAN, A. BORREMAN, A. MEIJERINK, AND W.C. VAN ETTEN, *Single-chip optical beam forming network in LPCVD waveguide technology based on optical ring res-*

onators, Proceedings of International Topical Meeting on Microwave Photonics (MWP'2006), IEEE France F1.4., Oct 2006

[11] P. Barton, *Digital Beam Forming of Radar,* IEE Proc. 127 No. 4, August 1980

[12] *www.complextoreal.com,*

[13] *Advanced Telecommunications Research Institute International (ATR),* 2-2 Hikaridai, Kyoto 619-02, Japan

[14] Albert Molderink, *Mapping FIR filters on a Montium,* Technical report, University of Twente, 2006

[15] MSc. K.H.G. Walters, *Cognitive Radio on a reconfigurable platform,* M.Sc. thesis, University of Twente, 2007

[16] Xilinx, *EDK Base System Builder (BSB) support for XUPV2P Board,* Xilinx (2005), www.xilinx.com

[17] Xilinx, *Xilinx University Program Virtex-II Pro Development System, Hardware Reference Manual,* Xilinx (2005), www.xilinx.com

[18] xilinx, *Embedded System Tools Reference Manual, Embedded Development Kit EDK 8.1i,* Xilinx (2005), www.xilinx.com

[19] National Semiconductors, *Datasheet ADCS7476/ADCS7477/ADCS7478,* National Semiconductors (2007), www.national.com

[20] M.D. van de Burgwal, *Hydra Protocol Specification,* Technical report, University of Twente (2007).

[21] Recore, *Simsation Compiler Getting Started Guide,* Recore (2007)

[22] Recore, *Simsation Simulator Quick Reference Guide,* Recore (2007)

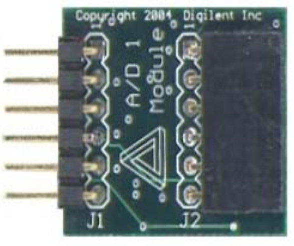
List of Figures

Appendix A

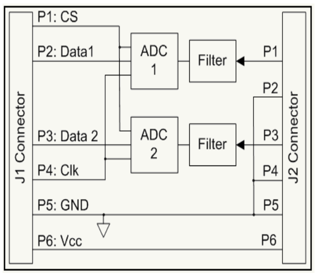
VHDL ADC interface design

The used analog digital converters are from Digilent Inc, modules PMOD-AD1. These modules features two 12 bit ADC with filtering and a 1 MSps sampling frequency. They are equipped with the ADC chips from National Semiconductors type ADCS7476. A picture and electrical diagram are shown in figure A.1.

The chips use a serial communication interface, which is not provided in the standard Xilinx development software. Therefore an interface is made separately as a VHDL module. This interface is connected to the OPB bus and uses the OPB clock of 10MHz to create a bit-clock and a sample-clock for the ADC's.



(a) Picture



(b) Electrical block diagram

Figure A.1: ADC module

The modules are connected with the use of a module interface board (MIB), this interface board is connected to the low speed expansion header 0 of the development board. The interface board has 8 6-pin connections, on each such connection an ADC module can be connected.

ADC protocol

The interface consists of four signal lines, the ADC is driven with a sample clock which initiates a sample/hold and conversion. The bit-clock is then used to clock the output of the different bits. 16 bits are transmitted the first 4 are leading zeros, the bits following is the sampled value. The pinning of the ADC modules is shown in table A.2. The protocol is verified with the use of a logic analyzer/digital scope (Rigol, DS 1102CD). The signal flow of two channels is recorded for 2 samples times, this is shown in figure A.2, consult table A.2 to recover the corresponding meaning of signal pins.

Memory interface

The VHDL interfaces with the OPB bus, in the hardware design the interface is given a base address. The interaction with the OPB is memory mapped. The mapping of different registers is shown in table A.1. The interface is free-running and is clocked from the OPB bus. Samples are in registers and are read out with a leading 0x50.00.0 or a leading 0x51.00.0 for hardware channel 0 or 1 respectively.

Channel	Offset addr.	MIB	Data format
1	4	1	0x50.00.0v.vv
2	8	1	0x51.00.0v.vv
3	12	2	0x50.00.0v.vv
4	16	2	0x51.00.0v.vv
5	20	3	0x50.00.0v.vv
6	24	3	0x51.00.0v.vv
7	28	4	0x50.00.0v.vv
8	32	4	0x51.00.0v.vv
15	56	8	0x50.00.0v.vv
16	60	8	0x51.00.0v.vv
Sattus reg.	64	-	0xAB.CD.xx.yy

Table A.1: Memory mapping of the interface, v.vv represent a 12 bits sampled value, xx and yy the status bits of channel 0 and 1.

Pin	Male/Digital Conn.	Direction	Description
Sample clock	1	I	Shown in figure A.2 as "D4"
Data line 0	2	O	Shown in figure A.2 as "D3"
Data line 1	3	O	Shown in figure A.2 as "D2"
Bit clock	4	I	Shown in figure A.2 as "D1"
Ground	5	I	
VCC	6	I	3.3 Power supply from dev. board
Pin	Female/Analog Conn.	Direction	Description
Analog inp 0	1	I	
Analog inp 1	3	I	
Ground	2,4,5	I/O	
VCC	6	O	

Table A.2: Hardware pin description of the ADC module

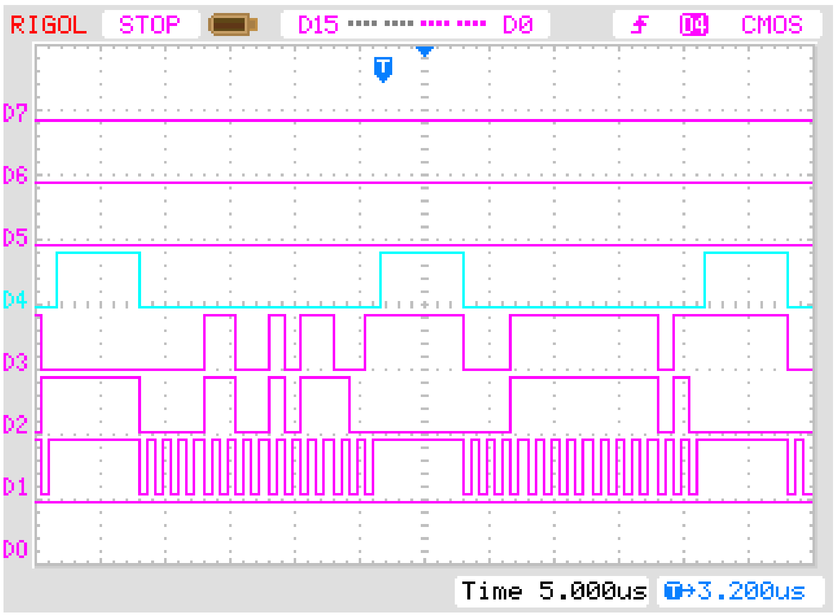


Figure A.2: Communication signals between ADC and development board

Appendix B

Source code of implementation

B.1 Time Delay

```
// Time Delay Implementation for Beam Forming
// Register mapping:
//
//    A                  B                  C               D
//1   OLDREGm2           OLDREGm1           NEWREGm1        NEWEREGm2
//2   FIRSTCOm2          FIRSTCOm1          SECONDCOm1      SECONDCOm2
//3                      writepointer2      readpointer2    buffer
//4   writepointer1      readpointer1       buffer          ONE
//
//
// Memory mapping:
// |   *ALU 1*     |   *ALU 2*     |   *ALU 3*     |   *ALU 4*     |   *ALU 5*
// |   X1   X5     |   X2   X6     |   X3   X7     |   X4   X8     |
// |   m1   m2     |   m1   m2     |   m1   m2     |   m1   m2     |
// |               |               |               |               |
//
// Lane Input/Output Mapping:
// Input:
//         Lane1: X1, X5
//         Lane2: X2, X6
//         Lane3: X3, X7
//         Lane4: X4, X8
//
// Output: result to lane ext4!

/**************************************************************
                  INITIALISATION
**************************************************************/
clock
agu p1m1 p1m2 p2m1 p2m2 p3m1 p3m2 p4m1 p4m2 p5m1 p5m2 |=0
agu p1m1 p1m2 p2m1 p2m2 p3m1 p3m2 p4m1 p4m2 p5m1 p5m2  =0
```

```
clock

/*************************************************************
                  PROCEDURE
*************************************************************/
rep i <- 1 2 3 4
   alu p.i.a4 -> p.i.o1, 0 -> p.i.o2   //Prepare readpointer1
end

nextSample:clock   //1
      mov ext1 -> p1m1     //Read input samples 1-4
      mov ext2 -> p2m1
      mov ext3 -> p3m1
      mov ext4 -> p4m1
      // Read sample from m1 at updated readpointer (b4) towards NEWREG (c1)
      rep i <- 1 2 3 4
         alu p.i.b4 add p.i.d4 -> p.i.o1
      end

   clock //2
      mov ext1 -> p1m2     //Read input samples 5-8
      mov ext2 -> p2m2
      mov ext3 -> p3m2
      mov ext4 -> p4m2
      rep i <- 1 2 3 4
         mov p.i.o1 -> p.i.b4 //Store updated readpointer
         mov p.i.o1 -> p.i.m1
         agu p.i.m1 int load
      end

      // Move NEWREG to OLDREG, creating 1 sample delay
      rep i <- 1 2 3 4
         alu p.i.c1 -> p.i.o2, p.i.a4 add p.i.d4 -> p.i.o1
         //writepointer++
         //alu p.i.a4 add p.i.d4 -> p.i.o1
      end

   clock //3
      rep i <-  1 2 3 4
         mov p.i.o2 -> p.i.b1    //Store previous sample in OLDREG
         mov p.i.o1 -> p.i.a4    //Store updated writepointer
      end
      rep i <- 1 2 3 4
         mov p.i.m1 -> p.i.c1    //Read new sample
      end

      // Multiply NEWREGm1 (c1) sample with FIRSTm1 coeff. (b2), total summed result to p4
      //Already prepare writepointer (a4) memories
      alu    p4c1   fmul p4b2              -> p4ws,     p4a4 -> p4o2
      rep i <- 2 3
         alu p.i.c1 fmul p.i.b2 sadd p.i.es -> p.i.ws,   p.i.a4 -> p.i.o2
      end
      alu    p1c1   fmul p1b2   sadd p1es   -> p1o1,     p1a4 -> p1o2

   clock //4
      //Store intermediate result in buffer (d3)
      mov p1o1 -> p4d3
      // Multiply OLDREG (b1) sample with SECOND coeff. (c2)
      alu    p4b1   fmul p4c2   sadd p4d3   -> p4ws
      rep i <- 2 3
         alu p.i.b1 fmul p.i.c2 sadd p.i.es -> p.i.ws
      end
      alu    p1b1   fmul p1c2   sadd p1es   -> p1o2      //This is the output sample!
      // Write sample at writepointer (a4) into (m1)
      rep i <- 1 2 3 4
         mov p.i.o2 -> p.i.m1             //This is not jet the updated pointer but the old pointe
         agu p.i.m1 int load
      end

   clock //5
      // move output from first 4 antennas to buffer p4c4
      mov p1o2 -> p4c4

      //update and set readpointer2
      // Read sample from m1 at updated readpointer2 (c3) towards NEWREG (d1)
      rep i <- 1 2 3 4
         alu p.i.c3 add p.i.d4 -> p.i.o1
      end


   clock //6
      //update writepointer2
      rep i <- 1 2 3 4
         mov p.i.o1 -> p.i.c3 //Store updated readpointer2
         mov p.i.o1 -> p.i.m2
         agu p.i.m2 int load
      end
      // Move NEWREG to OLDREG, creating 1 sample delay        //writepointer++
      rep i <- 1 2 3 4
         //alu p.i.d1 -> p.i.o2,       p.i.b3 add 1 -> p.i.o1
         alu                           p.i.b3 sub logic_true -> p.i.o1
      end

   clock //7
      // read mem2 into newreg2 (d1)
      // calculate newreg2*firstco2

      rep i <-  1 2 3 4
         mov p.i.o2 -> p.i.a1    //Store previous sample in OLDREG2
         mov p.i.o1 -> p.i.b3    //Store updated writepointer2
      end
      rep i <- 1 2 3 4
         mov p.i.m2 -> p.i.d1    //Read new sample
      end

      // Multiply NEWREG2 (d1) sample with FIRST2 coeff. (a2), total summed result to p4d1
      //Already prepare writepointer2 (b3) memories

      alu    p4d1   fmul p4a2   sadd p4c4   -> p4ws,        p4b3 -> p4o2
      rep i <- 2 3
         alu p.i.d1 fmul p.i.a2 sadd p.i.es -> p.i.ws,      p.i.b3 -> p.i.o2
      end
      alu    p1d1   fmul p1a2   sadd p1es   -> p1o1,        p1b3 -> p1o2

   clock //8
      // set writepointer2
      // calculate oldreg2*secondco2
      //Store intermediate result in buffer (c4)
      mov p1o1 -> p4c4

      // Multiply OLDREG2 (a1) sample with SECOND2 coeff. (d2)
      alu    p4a1   fmul p4d2   sadd p4c4   -> p4ws
      rep i <- 2 3
         alu p.i.a1 fmul p.i.d2 sadd p.i.es -> p.i.ws
      end
      alu    p1a1   fmul p1d2   sadd p1es   -> p1o2      //This is the output sample!

      // Write sample at writepointer (b3) into (m2)
      rep i <- 1 2 3 4
         mov p.i.o2 -> p.i.m2               //This is not jet the updated pointer but the
         agu p.i.m2 int load
      end

      set gpo1
   clock //9

      mov data p1o2 -> ext4                                      //Output the resulting s
      clr gpo1
      clock //10
      frz
jmp nextSample
```

B.2 Phase Shift

```
/*************************************************************
Beamsteering with Hilbert filtering and Complex multiplication
8 antennes

=> Real output only (Re x Re + Im x Im) (2MACs)
=> For demonstration purposes on the Xilinx Virtex-II Pro Development System
*************************************************************/
/* Register mapping:

A B C D
1 X[] temp_reg H[] accumulate register
2 p1d2:Endresultof8antennas
3 Qco_m1 YQ(i) (result Hilb m1) YI(i)(X(i-5)) Ico_m1 <= antennas 1-4
4 Qco_m2 YQ(i) (result Hilb m2) YI(i)(X(i-5)) Ico_m2 <= antennas 5-8
*/

// Memory mapping:
//     ext1        ext2 ext3 ext4
// |   *ALU 1*     |   *ALU 2*     |   *ALU 3*     |   *ALU 4*     |   *ALU 5*
// |   M1   M2     |   M1   M2     |   M1   M2     |   M1   M2     |   M1   M2
// |               |               |               |               |
// |   X1   X5     |   X2   X6     |   X3   X7     |   X4   X8     |   H[]  Input samples
// |   Odd |=0     |               |               |               |
// |   Even|=64    |               |               |               |

// Output result to lane ext4!!!

//agu p1m1 p1m2 p2m1 p2m2 p3m1 p3m2 p4m1 p4m2 p5m1 p5m2 =0
//agu p1m1 p1m2 p2m1 p2m2 p3m1 p3m2 p4m1 p4m2 p5m1 p5m2 |=0


def NANTENNAS
def NTABSHILBERT
def NSAMPLES
def ODD
def EVEN
def XMASK
def DEBUG

let ODD   := 0
let EVEN  := 64
let NSAMPLES    := 256
let NANTENNAS     := 8
let NTABSHILBERT  := 8
let XMASK         := (NTABSHILBERT-1)

clock
```

```
//Set agu's of X and of H on right positions

//Initialise
proc initialise
   agu p1m1 p1m2 p2m1 p2m2 p3m1 p3m2 p4m1 p4m2  = 0  //X1 till X8
   agu p1m1 p1m2 p2m1 p2m2 p3m1 p3m2 p4m1 p4m2 |= 0  //Odd or even X'range
   agu p5m1 = 0, p5m1 |= 0   //H
   agu p5m2 = 0, p5m2 |= 0          //Input samples, should be streaming, now for simulat
end

//Load sample
proc loadSample
      mov ext1 -> p1m1
      mov ext2 -> p2m1
      mov ext3 -> p3m1
      mov ext4 -> p4m1
      clock
      mov ext1 -> p1m2
      mov ext2 -> p2m2
      mov ext3 -> p3m2
      mov ext4 -> p4m2
      clock
end

proc load0
   rep i <- 1 2 3 4
      alu 0 -> p.i.o1
   end
end

proc calcHilbertm1
      rep i <- 1 2 3 4
         mov p.i.o1 -> p.i.d1 //Store accumulate register
         mov p5m1   -> p.i.c1 //Load H
         mov p.i.m1 -> p.i.a1 //Load X
         alu (p.i.a1 fmul p.i.c1) sadd p.i.d1 -> p.i.o1
      end
end

proc storeAccd3
      rep i <- 1 2 3 4
         mov p.i.o1 -> p.i.b3 // This is the momentarily YQ component //Store accumulate r
         agu p.i.m1 -= 4 & XMASK
      end
      clock
      rep i <- 1 2 3 4
         mov p.i.m1 -> p.i.c3     // This is the momentarily YI component
         agu p.i.m1 += 4 & XMASK
      end
end

proc calcHilbertm2
      rep i <- 1 2 3 4
         mov p.i.o1 -> p.i.d1 //Store accumulate register
         mov p5m1   -> p.i.c1 //Load H
         mov p.i.m2 -> p.i.a1 //Load X
         alu (p.i.a1 fmul p.i.c1) sadd p.i.d1 -> p.i.o1
      end
end

proc storeAccd4
      rep i <- 1 2 3 4
         mov p.i.o1 -> p.i.b4 // This is the momentarily YQ component //Store accumulate register
         agu p.i.m2 -= 4 & XMASK
      end
      clock
      rep i <- 1 2 3 4
         mov p.i.m2 -> p.i.c4    // This is the momentarily YI component
         agu p.i.m2 += 4 & XMASK
      end
end

proc mem2odd
      agu p1m1 p1m2 p2m1 p2m2 p3m1 p3m2 p4m1 p4m2 |= ODD
end

proc mem2even
      agu p1m1 p1m2 p2m1 p2m2 p3m1 p3m2 p4m1 p4m2 |= EVEN
      agu p1m1 p1m2 p2m1 p2m2 p3m1 p3m2 p4m1 p4m2 ++ & XMASK
end


proc multCV
      rep i <- 1 2 3 4
         alu (p.i.a3 fmul p.i.b3) -> p.i.o1 //calculate YQ * -Qcoefficient = Qr
(m1)
      end
      clock //1
      rep i <- 1 2 3 4
         mov p.i.o1 -> p.i.b1
         alu (p.i.c3 fmul p.i.d3) sadd p.i.b1 -> p.i.o1 //calculate YI * Icoefficient + Qr = CMres
      end
      clock //2
      rep i <- 1 2 3 4
         mov p.i.o1 -> p.i.d1
         alu (p.i.a4 fmul p.i.b4) sadd p.i.d1 -> p.i.o1 //calculate YQ * -Qcoefficient + CMr(m1) =
      end
      clock //3
      rep i <- 1 2 3 4
         mov p.i.o1 -> p.i.b1
         alu (p.i.c4 fmul p.i.d4) sadd p.i.b1 -> p.i.o1 //calculate YI * Icoefficient + Qr = CMres
      end
      clock //4
      mov p1o1 -> p1d1, p2o1 -> p2d1, p3o1 -> p3d1, p4o1 -> p4d1
      alu p4d1 -> p4ws
      rep i <- 2 3
         alu (p.i.d1 sadd p.i.es) -> p.i.ws
      end
      alu (p1d1 sadd p1es) -> p1o1
      clock //5
      mov data p1o1 -> gb02 -> ext4 // End result, total of 8(Q*Q + I*I)
end
```

```
clock
call initialise


nexttwosamples: clock

   call mem2odd

   clock //1
   call loadSample //2 3
   llc lc1 NTABSHILBERT - 2
   call load0 //0 clocks

   a: clock //4 5 6 7 8 9 10
   call calcHilbertm1 // X1 X2 X3 X4 //Calculate ant1-4, odd sample //0clocks
   agu p1m1 p2m1 p3m1 p4m1 ++ & XMASK
   agu p5m1 ++ & XMASK
   loop lc1 a

   clock //11
   call calcHilbertm1 // X1 X2 X3 X4
   agu p5m1 ++ & XMASK
   call storeAccd3 // 1 clock //12
   llc lc1 NTABSHILBERT - 2
   call load0

   b: clock //13 14 15  16 17 18  19
   call calcHilbertm2 // X5 X6 X7 X8 //Calculate ant5-8, odd sample
   agu p1m2 p2m2 p3m2 p4m2 ++ & XMASK
   agu p5m1 ++ & XMASK
   loop lc1 b

   clock //20
   call calcHilbertm2 // X5 X6 X7 X8
   agu p5m1 ++ & XMASK

   call storeAccd4 //21
   call load0

   clock //22
   call multCV   // Do complex vector multiplication //5clock cycles 23 24 25 26 27

   clock //28 output samples
   call mem2even

   clock //29
   call loadSample

   llc lc1 NTABSHILBERT - 2
   call load0
   c: clock
   call calcHilbertm1 // X1 X2 X3 X4 //Calculate ant1-4, even sample
   if DEBUG == 1
         agu p1m1 p2m1 p3m1 ++ & XMASK
   else
```

```
        agu p1m1 p2m1 p3m1 p4m1 ++ & XMASK
   endif
   agu p5m1 ++ & XMASK
   loop lc1 c
   clock
   call calcHilbertm1 // X1 X2 X3 X4
   agu p5m1 ++ & XMASK

   call storeAccd3

   llc lc1 NTABSHILBERT - 2
   call load0
   d: clock
   call calcHilbertm2 // X5 X6 X7 X8 //Calculate ant5-8, even sample
   if DEBUG == 1
        agu p1m2 p2m2 p3m2 ++ & XMASK
   else
        agu p1m2 p2m2 p3m2 p4m2 ++ & XMASK
   endif
   agu p5m1 ++ & XMASK
   loop lc1 d
   clock
   call calcHilbertm2 // X5 X6 X7 X8
   agu p5m1 ++ & XMASK

   call storeAccd4
   call load0
   clock
   call multCV    // Do complex vector multiplication
   clock


   set gpo1
   clock
   frz
   clr gpo1
   clock
   frz
jmp nexttwosamples
```

B.3 Hilbert Filter

```
/*****************************************************************************************
Beamsteering with Hilbert filtering and Complex multiplication
8 antennes

=> Real output only (Re x Re + Im x Im) (2MACs)
=> For demonstration purposes on the Xilinx Virtex-II Pro Development System
*****************************************************************************************
/* Register mapping:

A      B                C          D
1 X[]         temp_reg                   H[]               accumulate register
2                                p1d2:Endresultof8antennas
3 Qco_m1 YQ(i) (result Hilb m1) YI(i) (X(i-5)) Ico_m1 <= antennas 1-4
4 Qco_m2 YQ(i) (result Hilb m2) YI(i) (X(i-5)) Ico_m2 <= antennas 5-8
*/

// Memory mapping:
//     ext1        ext2 ext3 ext4
// |   *ALU 1*     |   *ALU 2*     |   *ALU 3*     |   *ALU 4*     |   *ALU 5*
// |   M1   M2     |   M1   M2     |   M1   M2     |   M1   M2     |   M1   M2
// |               |               |               |               |
// |   X1   X5     |   X2   X6     |   X3   X7     |   X4   X8     |   H[]  Input samples
// |   Odd |=0     |               |               |               |
// |   Even|=64    |               |               |               |

// Output result to lane ext1-4!!!

//Input
//ext1:  X_1  X_5  |                      |  X_1  X_5  |                      |  X_1  X_5
//ext2:  X_2  X_6  |                      |  X_2  X_6  |                      |  X_2  X_6
//ext3:  X_3  X_7  |                      |  X_3  X_7  |                      |  X_3  X_7
//ext4:  X_4  X_8  |                      |  X_4  X_8  |                      |  X_4  X_8
//Output:          |                      |            |                      |
//ext1:            |  Q_1  I_1  Q_5  I_5  |            |  Q_1  I_1  Q_5  I_5 |
//ext2:            |  Q_2  I_2  Q_6  I_6  |            |  Q_2  I_2  Q_6  I_6 |
//ext3:            |  Q_3  I_3  Q_7  I_7  |            |  Q_3  I_3  Q_7  I_7 |
//ext4:            |  Q_4  I_4  Q_8  I_8  |            |  Q_4  I_4  Q_8  I_8 |

def NANTENNAS
def NTABSHILBERT
def NSAMPLES
def ODD
def EVEN
def XMASK
def DEBUG

let DEBUG := 0 // This is for simulation in Matlab, use PP 1-3 in debug mode, instead of 1
let ODD  := 0
let EVEN := 64
let NSAMPLES    := 256
let NANTENNAS     := 8
let NTABSHILBERT  := 8
let XMASK         := (NTABSHILBERT-1)

//Initialise
proc initialise
   agu p1m1 p1m2 p2m1 p2m2 p3m1 p3m2 p4m1 p4m2  = 0  //X1 till X8
   agu p1m1 p1m2 p2m1 p2m2 p3m1 p3m2 p4m1 p4m2 |= 0  //Odd or even X'range
   agu p5m1 = 0, p5m1 |= 0
   agu p5m2 = 0, p5m2 |= 0
end

proc loadSample
      mov ext1 -> p1m1 //1
      mov ext2 -> p2m1 //2
      mov ext3 -> p3m1 //3
      mov ext4 -> p4m1 //4
      clock
      mov ext1 -> p1m2 //5
      mov ext2 -> p2m2 //6
      mov ext3 -> p3m2 //7
      mov ext4 -> p4m2 //8
end

proc load0
   rep i <- 1 2 3 4
      alu 0 -> p.i.o1
   end
end

proc calcHilbertm1
      rep i <- 1 2 3 4
         mov p.i.o1 -> p.i.d1 //Store accumulate register
         mov p5m1   -> p.i.c1 //Load H
         mov p.i.m1 -> p.i.a1 //Load X
         alu (p.i.a1 fmul p.i.c1) sadd p.i.d1 -> p.i.o1
      end
end

proc outputAccd3
      rep i <- 1 2 3 4
         //mov p.i.o1 -> p.i.b3
         // This is the momentarily YQ component //Store accumulate register
         mov data p.1.o1 -> gb0.(i+1) -> ext.i
         agu p.i.m1 -= 4 & XMASK
      end
      clock
      rep i <- 1 2 3 4
         //mov p.i.m1 -> p.i.c3
         mov data p.i.m1 -> gb0.(i+1) -> ext.i // This is the momentarily YI component
         agu p.i.m1 += 4 & XMASK
      end
end
```

```
proc calcHilbertm2
      rep i <- 1 2 3 4
         mov p.i.o1 -> p.i.d1 //Store accumulate register
         mov p5m1   -> p.i.c1 //Load H
         mov p.i.m2 -> p.i.a1 //Load X
         alu (p.i.a1 fmul p.i.c1) sadd p.i.d1 -> p.i.o1
      end
end

proc outputAccd4
      rep i <- 1 2 3 4
         //mov p.i.o1 -> p.i.b4 // This is the momentarily YQ component //Store accumulate
         mov data p.i.o1 -> gb0.(i+1) -> ext.i
         agu p.i.m2 -= 4 & XMASK
      end
      clock
      rep i <- 1 2 3 4
         //mov p.i.m2 -> p.i.c4    // This is the momentarily YI component
         mov data p.i.m2 -> gb0.(i+1) -> ext.i
         agu p.i.m2 += 4 & XMASK
      end
end

proc mem2odd
      agu p1m1 p1m2 p2m1 p2m2 p3m1 p3m2 p4m1 p4m2  |= ODD
end

proc mem2even
      agu p1m1 p1m2 p2m1 p2m2 p3m1 p3m2 p4m1 p4m2 |= EVEN
      agu p1m1 p1m2 p2m1 p2m2 p3m1 p3m2 p4m1 p4m2 ++ & XMASK
end

clock
call initialise
nexttwosamples: clock //
call mem2odd
   call loadSample // 1
   llc lc1 NTABSHILBERT - 2
   call load0

   a: clock // 2 3 4   5 6 7   8
   call calcHilbertm1 // X1 X2 X3 X4 //Calculate ant1-4, odd sample
         agu p1m1 p2m1 p3m1 p4m1 ++ & XMASK
   agu p5m1 ++ & XMASK
   loop lc1 a

   clock // 9
   call calcHilbertm1 // X1 X2 X3 X4
   agu p5m1 ++ & XMASK
   call outputAccd3 //10
   llc lc1 NTABSHILBERT - 2
   call load0

   b: clock // 11 12 13      14 15 16      17
   call calcHilbertm2 // X5 X6 X7 X8 //Calculate ant5-8, odd sample
```

```
         agu p1m2 p2m2 p3m2 p4m2 ++ & XMASK
   agu p5m1 ++ & XMASK
   loop lc1 b

   clock // 18
   call calcHilbertm2 // X5 X6 X7 X8
   agu p5m1 ++ & XMASK
   call outputAccd4 // 19

   clock // 20
   call mem2even
   call loadSample // ++1

   llc lc1 NTABSHILBERT - 2
   call load0
   c: clock // 21 22 23      24 25 26       27
   call calcHilbertm1 // X1 X2 X3 X4 //Calculate ant1-4, even sample
         agu p1m1 p2m1 p3m1 p4m1 ++ & XMASK
   agu p5m1 ++ & XMASK
   loop lc1 c

   clock //28
   call calcHilbertm1 // X1 X2 X3 X4
   agu p5m1 ++ & XMASK

   call outputAccd3 //29

   llc lc1 NTABSHILBERT - 2
   call load0
   d: clock //30 31 32     33 34 35         36
   call calcHilbertm2 // X5 X6 X7 X8 //Calculate ant5-8, even sample
         agu p1m2 p2m2 p3m2 p4m2 ++ & XMASK
   agu p5m1 ++ & XMASK
   loop lc1 d

   clock //37
   call calcHilbertm2 // X5 X6 X7 X8
   agu p5m1 ++ & XMASK

   call outputAccd4 //38
   call load0
   clock //39

   set gpo1
   clock //41 (++1)
   frz
   clr gpo1
   clock
   frz
jmp nexttwosamples
```

Appendix C

Montium tile processor

Here an explanation of the Montium tile processor is given. This chapter is copied from the report of [15] with permission from the author.

C.1 Introduction

The Montium tile processor is an embedded processor primarily used for streaming applications. It was developed as part of the Ph.D. thesis of Paul Heysters [4]. It is part of a system on chip template called Chameleon in which several tiles of different types are combined in a network on chip. Currently the Montium processor only exists as a VHDL description and is implemented on several FPGAs. This chapter describes the assets of the Montium that play a major role in the implementation of the sparse FFT. It is formost a reference, it provides some background information in order to understand the following chapters.

C.2 Coarse grain reconfiguration

Coarse grain reconfigurability is a term which addresses the amount of reconfigurability the hardware has. This ranges from a completely rigid architecture often normal ASIC like architectures to completely flexible, GPPs resign in this area. The Montium architecture is coarse grain reconfigurable. The Montium tile processor is a domain specific accelerator for the Chameleon System-on-Chip template. The template consists out of several different tiles connected through a Network-on-Chip. An instantiation of such an Chameleon template looks like the one in figure C.1.

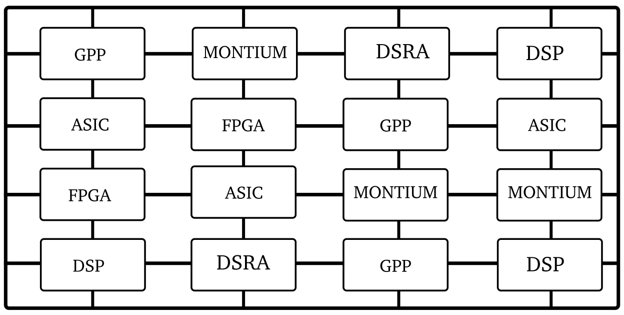


Figure C.1: An instantiation of the Chameleon template

C.3 Architecture

The Montium consists of 5 ALUs connected to 10 memories through an interconnect network, see figure C.2. Through east-west connections, the ALUs are also connected to eachother. It is characterised by its low power consumption and high efficiency. For example it is capable of doing a complex multiplication in a single clockcycle. The Montium is a 16 bit fixed pointed architecture.

Sequencer

The Montium is controlled through the sequencer. The sequencer implements a state machine that determines the instructions for the different components of the Montium. A program consists out of one or more states that are repeated. The sequencer takes care of this process. The number of sequencer of instructions are limited but their diversity is enormous. This diversity comes from the fact that all the hardware components can be configured in different ways and combined in different combinations and then used in the sequencer.

AGU

The AGU of the Montium architecture acts like a pointer in C. The AGU points towards the value in the memory from which can be read or to which can be written. The AGU, like a C pointer, can be assigned towards a single spot in the memory `agu p1m1 = 1`[1] or it can be altered by normal operators. The limiting factor is the amount of different AGU instructions. Each memory has its own AGU and each AGU can have only 8 different

[1]The AGU of memory p1m1 points to memory location 1

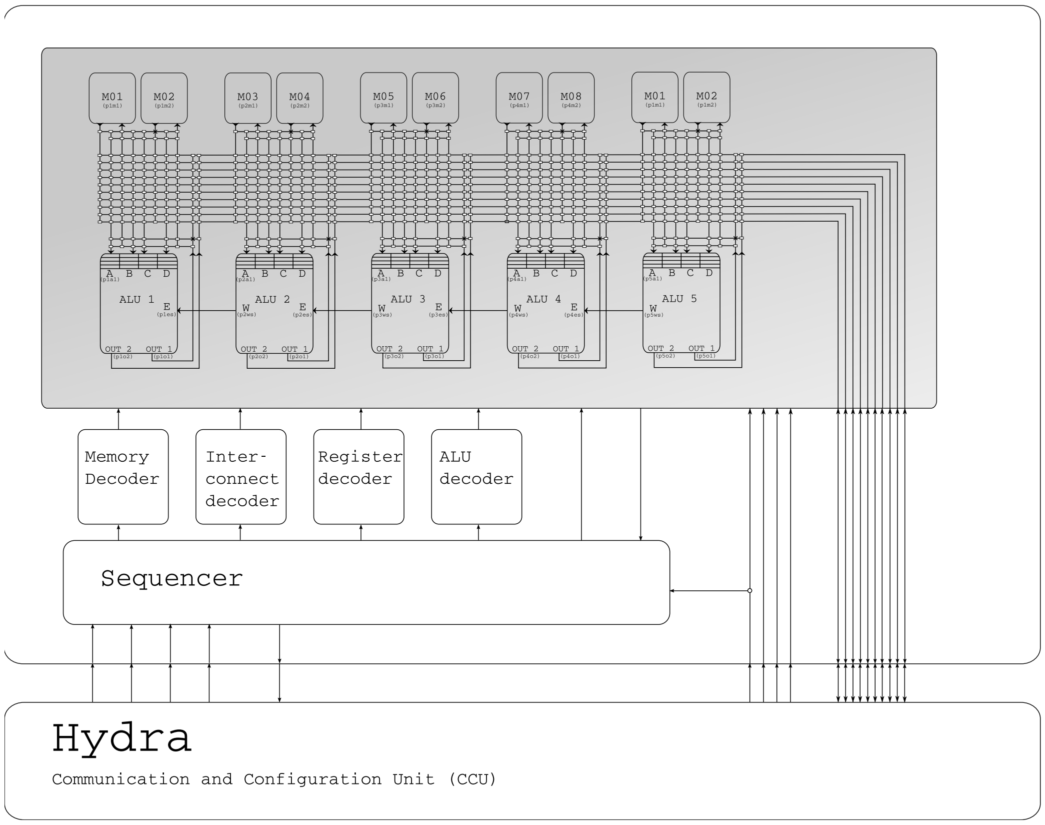


Figure C.2: The Montium tile including Hydra

instructions. This can be repeated almost indefinitely and with operators like ++ it is easy to reach all memory addresses with only 1 instruction.
An AGU does allow for more complex instructions which. For example:

```
agu p1m1 +=1 & 3 <-> 2 |= 2
```

This means that the AGU of memory p1m1 is increased by 1, it is masked with 3, it is bit reversed over 2 bits and its base address is 2. When this AGU would be initialised with 0 it would mean that the new AGU value would point to 4: the initial value is 0, add 1, leads to 1, bit reverse this over 2 bits, leads to 2. The base address was set to 2 already which with the previous outcome leads to 4.
Another feature of the AGU is that it can get the value from the output of an ALU. This means that any calculation from the ALU can be used as an input of the AGU. This is mainly used for look-up-table purposes.

Interconnect

The interconnect is the network which connects the memory to the registers of the ALUs. It is a transport network which can be configured in different ways to accommodate the source and destination that is needed.

ALU

The ALU of the Montium allows for many different calculations on the five possible input values. The ALU consists out of two parts. The first part can make different combinations of four different functional units. The functional units can do binary operations as well as boolean operations. The can then be fed into a multiplier which resides in the second part of the ALU. The adder and subtractor are also in this area. Next to the initial inputs it can take the input from the east input connection and provide an output to the west output. For a more comprehensive description of the ALU please read section 5.3 from [4].

Hydra

The Hydra provides the connection to the outside world. It is the only means by which input- and configuration-data can reach the Montium and output data can leave the Montium. Therefore it is also the only way to configure the Montium. To distinguish between data, addresses and configuration-data all the values are tagged. The tag together with the data is called a flit. For further information on these flits and their make-up, please read [20].

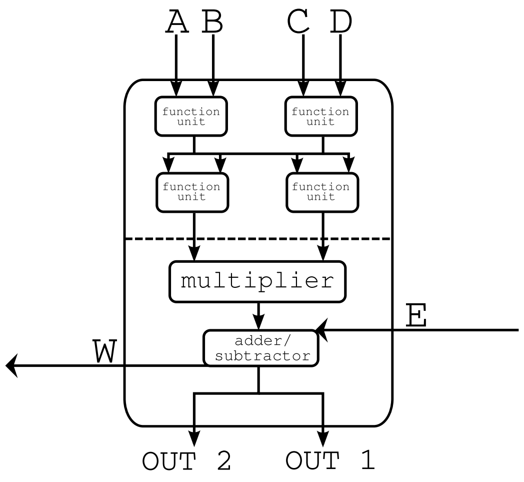


Figure C.3: An ALU of the Montium. The functional units are all capable of various logical operations. For a complete listing please refer to [4]

C.4 Application Development

Application development on the Montium is done on a normal pc from which the compiled code can be either fed into the simulator or an actual Montium. Most of the development is done using the simulator since it is the fastest. The compiled code can also be read by a tool called by the Montium Configuration tool. This tool is a graphical front end of all the configuration bits in the Montium. This allows for a bit wise configuration which is a very time consuming job and if possible needs to be avoided. It is however the only tool that makes run-time reconfiguration of the Montium possible and is therefore a necessity.

CDL Programming Language

The CDL programming language is the basic assembler like language in which the Montium is programmed. There is currently no C or higher level language available. Although assembler like, there are some major differences. For example clock cycles are explicitly declared which makes it possible to define explicit parallel calculations. The CDL language comes with a comprehensive pre-processor. It allows for basic structures like `for` loops which make it easier to generate the Montium code. For example when 100 clockcycles need to be programmed it is sufficient to program 1 in a for loop and let the pre-processor generate the other 99. For a complete overview of the CDL language and its capabilities please read [21].

Simulator

There is a simulator available for the Montium. It is capable of simulating the Montium including or excluding the Hydra. For input and output it uses files. These files contain the configuration data as well as the input data. All the current register values of the Montium can be read in a tree wise fashion like in the way files are read in a *nix prompt or old DOS prompt. This makes it difficult to have certain values side by side since quite a few directory changes could be needed. The simulator can single cycle step through the code generated but there is no way of going back or to break at a certain point. For more information on the simulator please read [22]
As of writing this thesis a second simulator is in development that does have these features. Although it does not show as many parameters of the Montium it is far more friendlier to use if only functionality of the algorithm needs to be tested. It has a graphical Java frontend and has the capability of stepping forward and backward and allows for a single breakpoint in the code.

Matlab

Matlab is a tool to format the input and read the output files. Matlab scripts either format only the input data in the correct order when there is no simulation of the Hydra. In this mode the compiled code is directly read by the simulator and Matlab's task is only to format the input data. It can also pack the configuration data together with the input data when the Hydra is part of the simulation. When the Hydra is simulated, Matlab's task is to tag all the configuration and input data according to the specification [20].