Digital Hardware Aspects of Multiantenna Algorithms

Fredrik Edman

Lund 2006

LUND UNIVERSITY

Department of Electroscience
Lund University
P.O. Box 118
SE-221 00 Lund
Sweden

Series of licentiate and doctoral theses
ISSN 1402-8662 No. 61

Abstract

The field of wireless communication is growing rapidly, with new requirements for the next generation of mobile and wireless communications technology. In order to achieve the capacities needed for future wireless systems, the design and implementation of advanced communications techniques such as multiantenna systems is required. These systems are realized by computationally complex algorithms, requiring new digital hardware architectures to be developed. The development of efficient and scalable hardware building blocks for realization of multiantenna algorithms is the focus of this thesis.

The first part of the thesis deals with the implementation of complex valued division. Two architectures implementing a numerically robust algorithm for computing complex valued division with standard arithmetic units are presented. The first architecture is based on a parallel computation scheme offering high throughput rate and low latency, while the second architecture is based on a resource conservative time-multiplexed computation scheme offering good throughput rate. The two implementations are compared to an implementation of a CORDIC based complex valued division.

The second part of the thesis discusses implementation aspects of fundamental matrix operations found in many multiantenna algorithms. Four matrix operations were implemented; triangular matrix inversion, QR-decomposition, matrix inversion, and singular value decomposition. Matrix operations are usually implemented using large arrays of processors, which are difficult to scale and consume a lot of resources. In this thesis a method based on the data flow was applied to map the algorithms to scalable linear arrays. An even more resource conservative design based on a single processing element was also derived. All the architectures are capable of handling complex valued data necessary for the implementation of communication algorithms.

In the third part of the thesis, developed building blocks are used to implement the Capon beamformer algorithm. Two architectures are presented; the first architecture is based on a linear data flow, while the second architecture utilizes the single processing element architecture. The Capon beamformer implementation is going to be used in a channel sounder to determine the direction-of-arrival of impinging signals. Therefore it was important to derive and implement flexible and scalable architectures to be able to adapt to different measuring scenarios. The linear flow architecture was implemented and tested with measured data from the channel sounder. By analyzing each block in the design, a minimum wordlength design could be derived.

The fourth part of the thesis presents a design methodology for hardware implementation on FPGA.

Contents

Part IV

Preface

This thesis summarizes the results of my academic work in the Digital ASIC group at the department of Electroscience, Lund University, for the Ph.D. degree in circuit design. The main contributions of this thesis are derived from the following publications;

Fredrik Edman and Viktor Öwall, “Implementation of a Scalable Matrix Inversion Architecture for Triangular Matrices,” *Proceedings of PIMRC’03,* Beijing, China, September 2003.

Fredrik Edman and Viktor Öwall, “Implementation of a Highly Scalable Architecture for Fast Inversion of Triangular Matrices,” *Proceedings of ICECS’03,* Sharjah, United Arab Emirates, December 2003.

Fredrik Edman and Viktor Öwall, “Implementation of a Full Matrix Inversion Architecture for Adaptive Antenna Algorithms,” *Proceedings of WPMC’04,* Abano Terme, Italy, September 2004.

Zhan Guo, Fredrik Edman, Peter Nilsson, and Viktor Öwall, "On VLSI Implementations of MIMO Detectors for Future Wireless Communications," *Proceedings of IST-MAGNET Workshop,* Shanghai, November 2004.

Fredrik Edman and Viktor Öwall, “A Scalable Pipelined Complex Valued Matrix Inversion Architecture," *Proceedings of ISCAS’05,* Kobe, Japan, May 2005.

Fredrik Edman and Viktor Öwall, “Hardware Implementation of two Complex Divider Architectures," *Proceedings of RVK’05,* Linköping, Sweden, June 2005.

Fredrik Edman and Viktor Öwall, “Fixed-point Implementation of a Robust Complex Valued Divider Architecture,”*Proceedings of ECCTD’05,* Cork, Ireland, August 2005.

Fredrik Edman and Viktor Öwall, “A Computational Platform for Real-time Channel Measurements using the Capon Beamforming Algorithm,” *Proceedings of WPMC’05,* Aalborg, Denmark, September 2005.

Fredrik Edman and Viktor Öwall, “Compact Matrix Inversion Architecture Using a Single Processing Element,” *Proceedings of ICECS’05,* Gammarth, Tunisia, December 2005.

Fredrik Edman and Viktor Öwall, ”A Compact Real-time Channel Measurement Architecture Based on the Capon Beamforming Algorithm,” *Proceedings of DSPCS'05 and WITSP'05,* Nosa Heads, Australia, December 2005.

Fredrik Edman, Fredrik Tufvesson and Viktor Öwall, ”Computational and Hardware Aspects of the Capon Beamformer Algorithm,” *To be submitted to IEEE Trans. on Consumer Electronics.*

The first part of the thesis deals with complex valued division. The main contribution in this part is the derivation and implementation of two architectures based on a numerically stable algorithm, which reduces the number of overflows and underflows during computation. One of the architectures offers high-throughput rate producing one complex valued division per clock cycle in continuous run, while the second architecture offers lower resource consumption with good throughput rate. The two

implementations are compared to a CORDIC based complex valued divider implementation.

The second part of the thesis covers the architectural development of four fundamental matrix operations; triangular matrix inversion, QR-decomposition, matrix inversion, and singular value decomposition. A novel method, mapping the algorithms to scalable linear array architectures more suitable for hardware implementation, was used. Single processing element architectures for the algorithms are also proposed. All architectures were implemented using standard arithmetic units with good throughput rate, and are capable of handling both real and complex valued matrices.

In the third part of the thesis, the building blocks from part I and part II are combined to implement the Capon beamformer algorithm. The main contribution of this part is the derivation of a scalable architecture suitable for using in a constantly changing channel sounder. A wordlength analysis was done to determine the minimum wordlength architecture for the Capon beamformer implementation.

In the last part of this thesis presents a design methodology for FPGA implementations, derived and used by the author, is presented.

In total 16 FPGA implementations are presented in this thesis.

Acknowledgements

I would like to start by thanking my supervisor, Dr. Viktor Öwall, for all the help throughout the years studying for my Ph.D. and for guiding and supporting me in my research work.

A special thanks to Dr. Mats Torkelsson for all the discussions over the years concerning everything from research to “crazy” ideas. I am also grateful to Dr. Peter Nilsson and Dr. Fredrik Tufvesson for helpful technical discussions.

I would like to thank the technical and administrative staff at the department. Erik and Leif for maintaining the computer network and the many hours of stimulating talks about computers and software, Stefan for his help with the CAD tools, Lars for helping with the day-to-day work, and not least Pia, Elsbieta, Stina, and Birgitta for all their administrative assistance.

I would also like to extend my gratitude to all colleagues and friends at the Department of Electroscience. Thanks for all the lively and highly stimulating discussions during lunch and coffee breaks.

Thanks to the Personal Computing and Communication (PCC) national research program financed by the Foundation for Strategic Research (SSF), Vinnova Competence Center for Circuit Design (CCCD), and EU’s 6th framework IST-MAGNET program for supporting this project.

Finally, I would like thank Anna for all the help with the thesis, and more importantly for being such a wonderful, supportive, and loving person.

Lund, January 2006

Fredrik Edman

Introduction

“Logic is a systematic method of coming to the wrong conclusion with confidence.”

Rathenn, B5

Chapter 1

Background

This part of the thesis gives a brief introduction to the area of wireless communication, and more precisely, the area of multiantenna systems and algorithms. The first chapter begins with the author's view of the future of wireless communication and which engineering challenges it will bring.

In chapter two, a brief introduction to the fundamentals of multiantenna systems, including smart antennas and MIMO systems, is given. The chapter continues with a discussion about smart antenna and MIMO algorithms. It is shown that the algorithms can be decomposed into a few fundamental matrix operations, referred to as building blocks. These building blocks needs to be optimized for high throughput rate and be scalable to meat the demands of future mobile communication algorithms.

Chapter three deals with the choice of implementation platform when implementing high performance matrix operations for multiantenna systems. Different platforms such as general purpose processors, digital signal processors, FPGA, and ASIC are reviewed.

1.1 The future of wireless communication

In modern usage, wireless communication refers to a method that uses radio waves to transmit data between devices without cables or cords. Historically, the transmitted data has been voice originating from people talking in phones with each other. Today more and more electronic handset devices are communicating with each other via radio waves thereby getting rid of cables.

Wireless communication is growing rapidly and we are still only at the beginning of the mobile revolution. Already the requirements for the next generation of mobile and wireless communications technology are emerging. Future mobile and wireless networks will be characterized by open, global, and ubiquitous communications, making people free from spatial and temporal constraints. Different communication technologies including global and short range wireless systems as well as wired systems will be combined in a common platform to complement each other in the optimal way for different service requirements and radio environments (figure 1.1).

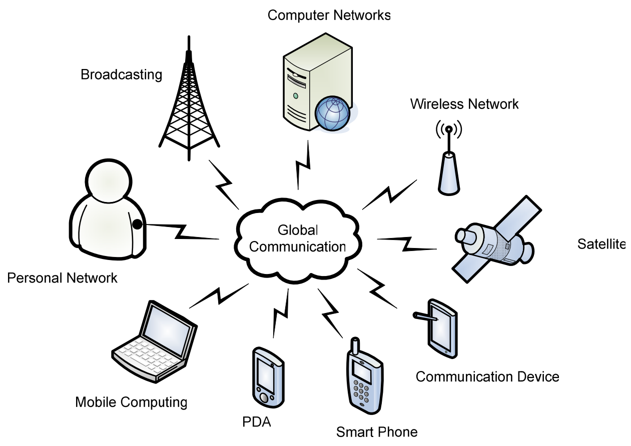


Figure 1.1 Interaction between different types of communications systems creating a true global communication system.

In the next generation of mobile communication, often referred to as 4G, different areas such as information technology, multimedia and mobile communications will be combined and integrated together. Right now, the majority of traffic is changing from speech-oriented communications to data based communications. In the near future, it is expected that the number of portable devices such as voice enabled PDAs, laptops and general hand held devices will exceed the number of PCs connected to the Internet. These portable devices or mobile terminals will use global mobile access to share and access multimedia services for voice, data, messages, video, Internet, GPS, radio, TV, software, etc. There are many fields that could greatly benefit from 4G, but the most rapidly growing is the field of entertainment, including music, written material, gambling, video, and games, which enables interaction in real-time with other users, locally or globally (figure 1.2).

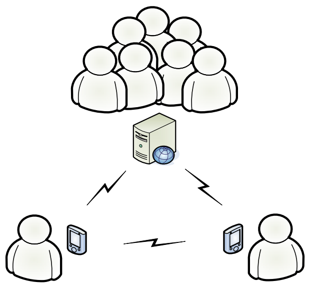


Figure 1.2 Real-time interactions and sharing of data between people, both locally and globally, over wireless links.

All these applications will be heavily bandwidth consuming, resulting in high data rate requirements for future systems. To be able to meet the demand of higher data rates, new high-speed communications systems must be developed.

Chapter 2

Multiantenna systems

The wireless spectrum is limited and during the last decade it has become a precious resource. Achieving the capacities needed for future wireless systems without increasing the required spectrum will only be accomplished by the design and implementation of advanced communications techniques such as multiantenna systems. These systems are realized by time-consuming and computationally complex algorithms, requiring new digital hardware architectures to be developed. The development of efficient hardware architectures for multiantenna algorithms is the focus of this thesis.

Multiantenna systems consist of two or more antenna elements either at the transmitter, the receiver, or both. Here the two different groups of multiantenna systems, smart antenna systems and multiple input - multiple output systems (MIMO), will be discussed.

2.1 Smart antenna systems

A smart antenna is a digital wireless communications antenna system with multiple antenna elements at the source (transmitter), the destination (receiver), or both, where signals from the different antenna elements are combined or created in an intelligent way by an algorithm. The smart antenna system can be utilized in a number of ways. It can be used to increase the capacity and the coverage (beamforming) in a mobile communication system. It can also be used for improving the link quality, user position estimation, and to decrease the delay dispersion [1].

In conventional wireless communications, a single antenna is used at the source, and another single antenna is used at the destination as shown in figure 2.1. This communication system is referred to as a single input - single output (SISO) system.

Assume that a transmitter with a single antenna element transmits omni directional, meaning that the signal or wavefront is transmitted in all directions, and that the receiver antenna listens for signals coming from all directions. Sending signals by transmitting energy in all directions is not energy efficient.

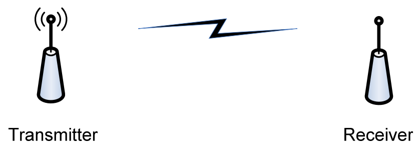


Figure 2.1 A conventional SISO communications system.

A better way is to only transmit in the direction of the receiver. In the same manner it is more efficient to only listen in the direction of the transmitter and not in all directions at the same time. This will increase energy efficiency and will also lead to a reduction in interference between different transmitters and thereby increase the efficiency in an interference limited system.

Another drawback with SISO systems is that they are vulnerable to multipath effects. When the electromagnetic wavefront travels towards the receiver, its propagation path can be obstructed by objects. In an outdoor environment this can for instance be caused by objects such as hills, buildings, trees, cars, etc., while in an indoor scenario the signal can be obstructed by doors, walls, people, furniture, etc. The wavefronts will then be reflected and scattered by these objects, thus creating multiple paths to the receiver (figure 2.2).

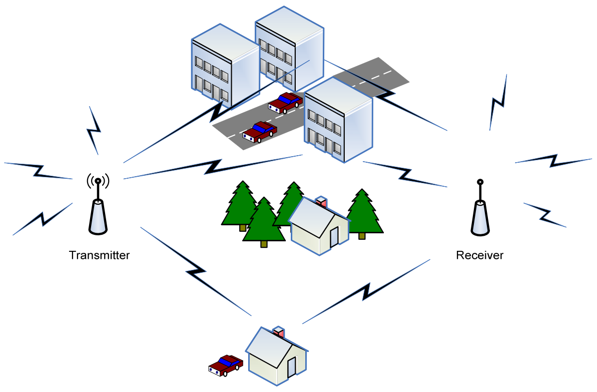


Figure 2.2 Scattered and reflected signals due to obstructions, causing multipath effects.

The wavefront, arriving in scattered portions at different time instances, can cause problems resulting in intermittent reception [2],[6]. In digital communications this can cause an increase in the number of errors resulting in a reduction in data rate. The use of smart antennas can reduce the deterioration of the transmitted wavefront caused by multipath wave propagation by automatically changing the directionality of its radiation patterns in response to its signal environment.

There are mainly two different categories of smart antenna systems [6]:

- Single input - multiple output system (SIMO). In a SIMO system, one antenna is used at the transmitter, and two or more antennas are used at the receiver as shown in figure 2.3.

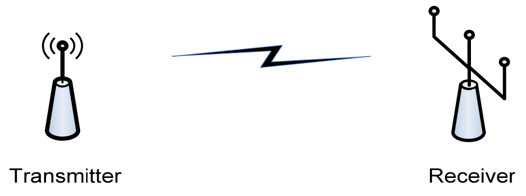


Figure 2.3 A single input - multiple output system.

- Multiple input - single output (MISO). In a MISO system, two or more antennas are used at the transmitter, and one antenna is used at the receiver as shown in figure 2.4.

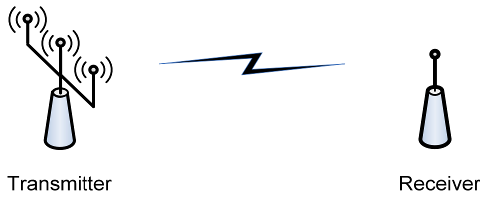


Figure 2.4 A multiple input - single output system.

By applying the techniques shown in figures 2.3-2.4 we can transmit in a specific direction or listen in a specific direction. Figure 2.5 shows the same scenario as in figure 2.2 but with a smart antenna as a receiver. The smart antenna system detects the three multipaths and creates “listening” beams for those directions. Subsequently, all other signals are suppressed [5]-[7].

In this way the signals coming from the directions of the listening beams can be combined at the receiver, thus increasing the signal-to-noise ratio and lowering the bit error rate.

The concept of using smart antennas to transmit and receive data more intelligently has existed for many years. Simple smart antenna techniques, like the switched beam technology, where the antenna systems form multiple fixed beams with heightened sensitivity in particular directions, have been used in commercial applications for some time [4]-[6]. These antenna systems detect signal strength, choose from one of several predetermined, fixed beams, and switch from one beam to another as the mobile device moves throughout the beampattern.

Smart antenna technology represents the most advanced approach to date taking advantage of its ability to effectively locate and track various types of signals to minimize interference and maximize signal reception [6].

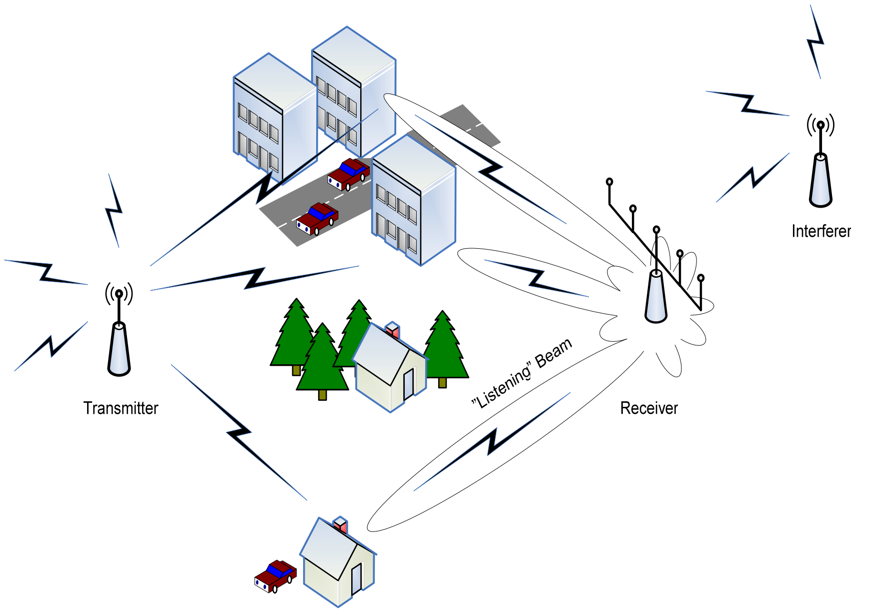


Figure 2.5 A SIMO system where the multiple antennas at the receiver create beams that listen in the directions of the multipaths.

One sophisticated utilization of smart antenna technology is spatial division multiple access (SDMA) [1],[6]. In this technique, single mobile terminals are located and tracked by adaptively steering transmission signals toward users and away from interferers (figure 2.6). In this way a high level of interference suppression is achieved, making possible more efficient reuse of the frequency spectrum.

Smart antenna technology can, with some modification, be integrated into all major access methods such as frequency division multiple access (FDMA), time division multiple access (TDMA), code division multiple access (CDMA), etc. and has widespread applications in several different areas such as digital television (DTV), body area networks (BAN), personal area networks (PAN), wireless local area networks (WLAN), metropolitan area networks (MAN), and mobile communications [7],[8]. However, the technique requires sophisticated algorithms and computationally heavy algorithms to operate in real-time.

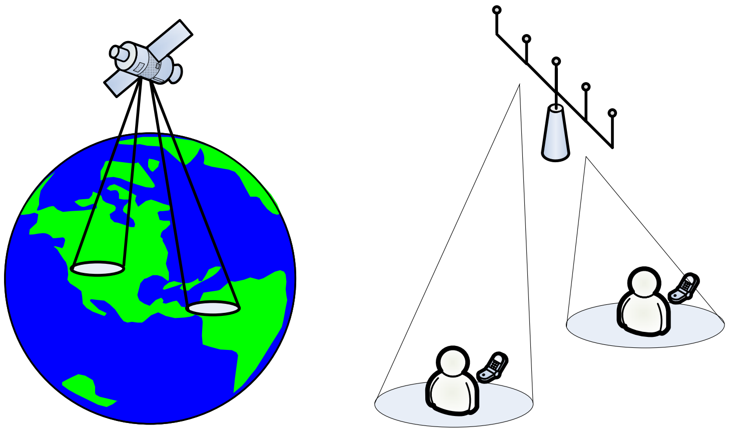


Figure 2.6 Smart antenna techniques can be used in satellite transmission to cover small hot spots, or in cellular systems to track individual mobiles.

2.2 MIMO systems

MIMO systems are characterized by having multiple antennas at both the transmitter and the receiver as shown in figure 2.7. The number of antenna elements does not have to be the same at the transmitter and the receiver.

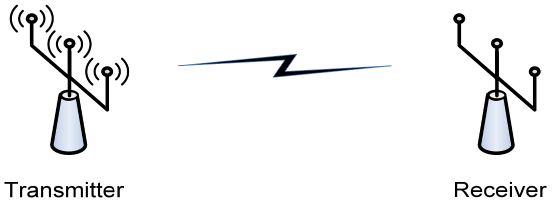


Figure 2.7 A multiple input - multiple output system.

A MIMO system is mainly used for three different purposes; beamforming, diversity, and spatial multiplexing. Both beamforming and diversity can be use in the same way as in the case of the smart antenna system [6]. By applying a MIMO beamforming system to the scenario in figure 2.2, the signal can be transmitted in one or more favorable directions. Figure 2.8 shows how the signal is transmitted in two beams from the transmitter and received via two beams formed by the receiver antenna.

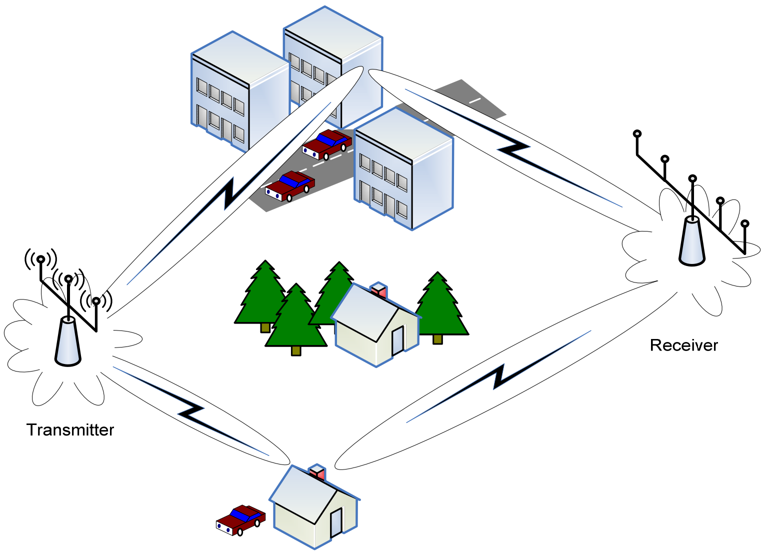


Figure 2.8 A MIMO system using beamforming to transmit the signal in specific directions and creating beams to listen for signals coming from those directions.

In this way transmission energy is saved, since less energy is transmitted in other directions than those of the receiver.

Another way of using a MIMO system that has attracted lots of interest in recent years is spatial multiplexing [7]-[9]. Spatial multiplexing offers an improvement of the capacity by simultaneously transmitting multiple datastreams. This is done by multiplexing a datastream into several parallel datastreams that are sent from separate antenna elements as shown in figure 2.9 [7].

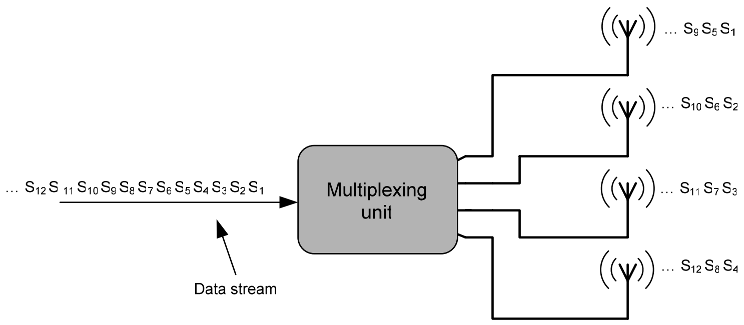


Figure 2.9 A datastream is multiplexed onto different antenna elements in a MIMO system.

Data transmitted from the multiple antenna elements will be mixed when traveling throughout the propagation channel as shown by figure 2.10. Each individual antenna element in the receiver will detect a combination of the transmitted data.

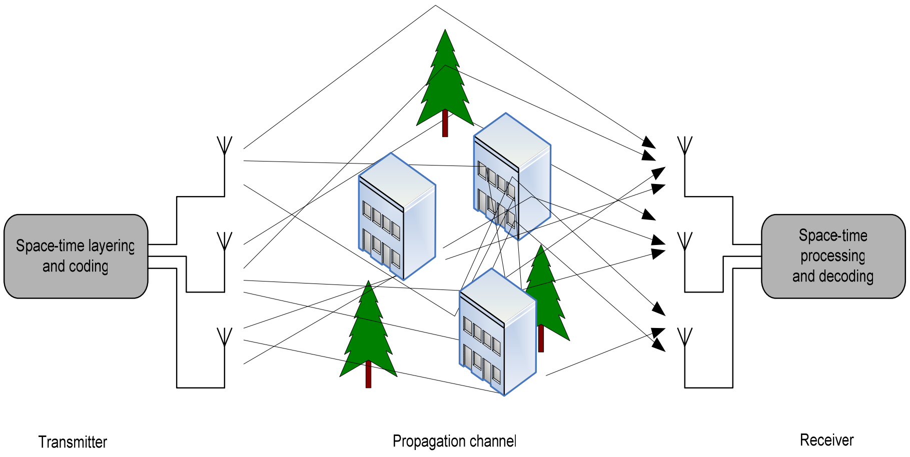

Space-time layering and coding
Space-time processing and decoding
Transmitter
Propagation channel
Receiver

Figure 2.10 The transmitted datastream is mixed when traveling through the channel.

The received data must then be resolved by signal processing algorithms before it can be combined into a single datastream again. In this way MIMO can exploit the phenomena of multipath propagation to increase throughput, or reduce bit error rates, rather than suffer from it [7].

MIMO will be incorporated into the new IEEE 802.11n standard for local-area wireless networks, which will improve the coverage and data rate significantly. The IEEE 802.11n standard is still being discussed, but data throughput is estimated to reach a theoretical 540 Mbit/s. The data rate requirement at the physical layer may be even higher, prompting for new high-speed hardware solutions. Although a few manufacturers have released consumer products with so called pre-n hardware, exploiting rudimentary diversity by using 2 to 4 antenna elements, the widespread usage of MIMO will not be a reality before the standard is set. MIMO has also been added to the latest draft version of Mobile WiMAX (IEEE 802.16e).

To be able to fully take advantage of the emerging standards, new high-throughput hardware architectures must be developed. A first step in the development process is to analyze the multiantenna algorithms and to identify common algorithmic features.

2.3 Multiantenna algorithms

It is important to point out that the antennas themselves are not “smart”, it is rather the underlying antenna systems that have the intelligence in the form of advanced signal processing algorithms. In order to be able to take full advantage of the multiantenna techniques, discussed in the previous section, advanced and computationally heavy communications algorithms must be used. There are myriads of different algorithms, which are optimized and specialized for different multiantenna systems and for different user scenarios. A brief discussion on smart antenna and MIMO methods are given below.

2.3.1 Smart antenna algorithms

Smart antennas, in their simplest form, linearly combines antenna signals into a weight vector that is used to control the beam pattern. The weights can be determined in a number of ways using different algorithms. These smart antenna algorithms can crudely be divided into three classes of algorithms, spatial reference, temporal reference, and blind algorithms [7]. The common features of the two first algorithm classes are that they both form beam patterns and they are based on linear weighting and addition of received signals at the antenna elements. The difference between the two classes is in how they calculate the antenna weights. The third class of algorithms uses neither of the features used by spatial and temporal reference algorithms. Instead they exploit the statistical properties of the transmit signal.

In *Spatial reference algorithms (SR)* the antenna weights are chosen based on knowledge of the array structure [6],[7]. These algorithms estimate the direction of arrival (DOA) of both the desired and interfering signals. The DOAs can be determined by applying different methods to the sampled data from the antenna array. The simplest way of extracting the DOAs is to use spatial Fourier transform on the signal vector. This method is limited by its resolution (size of antenna array) and has therefore limited usages. In cases where good resolution is necessary, so called high-resolution methods could be used. High-resolution methods are limited only by the modeling errors and noise and not by the size of the antenna array [6],[7] . Common high-resolution algorithms include:

- Minimum Variance Method (a.k.a. Capon’s beamforming algorithm) [6]. Capon’s algorithm is a spectral-based search method. It determines an angular spectrum for each direction by minimizing the noise and the interference from other directions. This algorithm has been implemented in this thesis and is discussed in more detail in part III.
- MUSIC algorithm [6],[7]. This algorithm determines the signal and noise subspaces and then searches the spectrum to find DOAs.

- ESPRIT algorithm [6]. This algorithm determines the signal subspace, from which the DOAs are determined in closed form.
- SAGE algorithm [6],[7]. The SAGE algorithm is based on maximum-likelihood estimation of the parameters of the impinging waves on the antenna array.

When the DOAs are determined an appropriate beampattern is created that maximizes the beam pattern in the direction of the wanted signals and places nulls in the direction of unwanted interfering signals.

Temporal reference algorithms (TR) are based on prior knowledge of the time structure of the received signals [6],[7]. Usually a training sequence is used as a temporal reference. The receiver aims to adjust or choose antenna weights in a way so that the deviation of the combined signal at the output and in the known training sequence is minimized. The calculated weights are then used to form a beam pattern.

The third class of algorithms is termed *blind algorithms (BA)* [6],[7]. These algorithms are based on prior knowledge of the signal properties of the transmitted signal. Depending on which statistical properties of the transmitted signal are exploited, we are able to apply different algorithms to determine the signal matrix from the received sample data.

2.3.2 MIMO algorithms

Many of the approaches and algorithms used in conjunction with SIMO and MISO systems can also be used in a MIMO system. Thus, the MIMO system can be used for diversity and/or beamforming both at the receiver and at the transmitter [6],[7]. However, there are more elaborate schemes available for MIMO systems, such as layered space-time structures (LST), also known as Bell Labs Layered Space-Time (BLAST) architectures [10]. The simplest BLAST architecture is the horizontal (also known as vertical) BLAST architecture, where the datastream is multiplexed into several parallel streams, which are encoded separately before transmitted. A drawback with horizontal BLAST (H-BLAST) is that it does not utilize diversity [7],[11],[12]. However, another scheme called diagonal BLAST (D-BLAST) cycles the streams through the different transmitter antennas in such way that every stream is sent on every possible antenna element, thus taking advantage of diversity. The transmitted datastreams can also be encoded using space-time coding (STC) [9],[13]. In space-time coding redundancy is introduced by encoding the datastream and sending a differently encoded version of same signal from each antenna element. Space-time codes may be split into two main types: Space–time trellis codes, which distribute a trellis code over multiple antennas and multiple time-slots and provide both coding gain and diversity gain, and space–time block codes, which act on a block of data at once and provide only diversity gain, but are much less complex in implementation terms than trellis codes [9],[13].

The optimal detection strategy for a MIMO receiver is to perform maximum likelihood estimation (MLE) over all possible transmitted symbol sets [11]-[13]. To date, such an approach has been considered too complex to implement for high data rate systems.

MIMO algorithms is a hot research topic, and there are many algorithms that combine the different features discussed above. However, a common feature among the algorithms is that they operate on matrix data and are very computationally heavy.

2.4 Building blocks for multiantenna algorithms

2.4.1 Common building blocks

The advanced communication algorithms can be described and realized under a common framework of well known methods. This framework consists for example of methods such as solving least squares problems, eigenvalue and singular valued decomposition, factorizations, or using filtering techniques such as Kalman filtering [6]-[8]. In multiantenna systems the received data is often collected and processed by these methods in either matrix or vector form. These methods can therefore be decomposed into fundamental matrix operations such as matrix-matrix multiplication, matrix-vector multiplication, matrix inversion, QR-decomposition, LU-decomposition, eigenvalue decomposition (EVD), singular value decomposition (SVD), etc. [14],[15]. For example, in the case of the Capon algorithm a matrix is to be inverted and in the ESPRIT algorithm a SVD is done. Some of the matrix operations will be discussed in detail in part II of the thesis which deals with the implementation of matrix operations such as the QR-decomposition and matrix inversion.

Figure 2.11 shows a hierarchic view of the decomposition of the multiantenna algorithms into common methods, which are decomposed into basic matrix operations. The matrix operations presented in figure 2.11 constitute the computational bottle neck in the hardware implementations of multiantenna algorithms. To be able to develop efficient multiantenna systems a bottom-up approach must be applied. This means that by developing optimized building blocks performing basic matrix operations, efficient, thereby meaning high throughput rate, multi antenna algorithms can be implemented. The main focus of this thesis is on the development of some of the most commonly used building blocks.

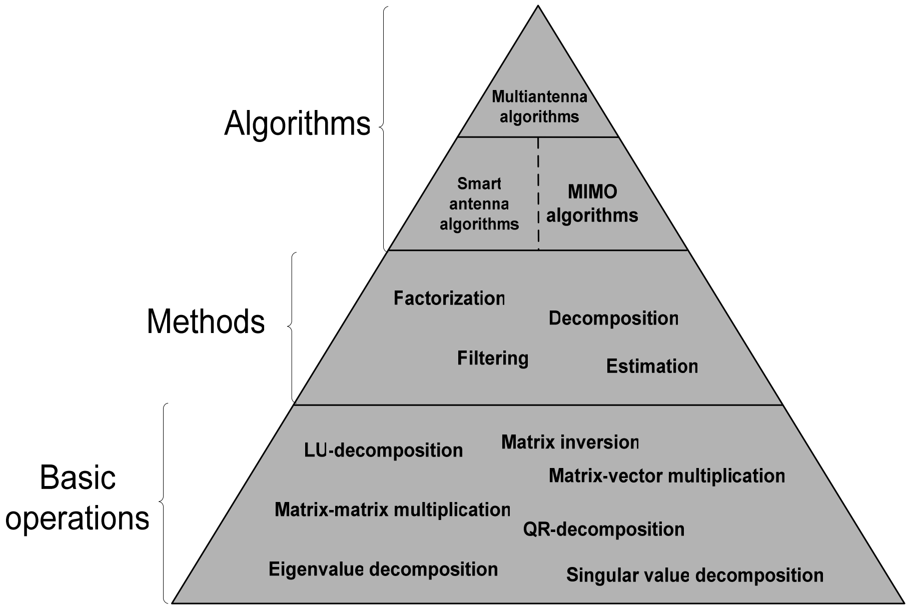


Figure 2.11 Multiantenna algorithms can be decomposed into common matrix operations which constitutes fundamental building blocks.

2.4.2 Scalable building blocks

Besides efficient implementations, scalability of the hardware design is an important factor. Previous hardware implementations of basic matrix operations have often been mapped onto large multiprocessor networks, such as array architectures or massive parallel architectures. Such architectures are seldom easy to tailor for one owns needs and they tend to grow rapidly with the matrix size, thus consuming large amounts of hardware resources. Resource conservative and scalable hardware blocks therefore highly sought for. The scalability of the architectures developed in this research project has therefore been prioritized.

Basic matrix operations are not only used in the implementation of multiantenna systems. Matrix operations occur in virtually every branch of engineering [15]. Therefore, it is important to make hardware blocks that can be used in many different types of applications.

Chapter 3

Implementation platform

3.1 Choosing platform

There are several different hardware platforms to choose from when implementing high performance algorithms for multiantenna system. Four categories of platforms can be identified; general purpose processors, digital signal processors, field programmable gate arrays (FPGAs), and application specific integrated circuits (ASICs). These four categories of platforms can be divided into general architectures and specialized architectures. A general architecture is a platform which is not limited to a specific application, such as implementation of filters. On the contrary, its purpose is to be able to adapt to many different areas of applications. General purpose processors and digital signal processors are two categories of platforms typically belonging to general architectures.

General purpose processors (a.k.a. micro processors or short µP) are software programmable circuits found in many types of consumer electronic products. Two well-known companies making µPs for personal computers are AMD [17] and Intel [18]. µPs have some drawbacks when it comes to functioning as a platform for implementing high-performance multiantenna algorithms. One drawback is that the hardware architecture and the programming instruction set is too general, resulting in a relatively low performance for signal processing applications. Another drawback is the huge power consumption required for high performance processors, which in many cases reaches over 100W.

Digital signal processors are usually based on a multiply-accumulate unit (MAC) which is a key digital signal processing unit, with the support of specialized hardware blocks and addressing modes. The specialized hardware blocks acts as accelerators and can be utilized by high-level software programming of the circuit. Today, many companies make high-performance digital signal processors targeting specific areas of applications. One of the leading companies in the market is Texas Instruments. They have a wide range of processors ranging from high performance to low power, and include both float- and fixed-point alternatives [19].

The advantage of using a general architecture can come at the price of not reaching a high enough throughput rate. If high-throughput applications, such as digital signal processing for multiantenna systems, are going to be implemented, the general architecture platforms do not reach the throughput rate needed. To be able to implement these high-throughput applications a specialized architecture must be used. Two such platforms where specialized architectures can be implemented are FPGAs and ASICs.

An FPGA is a semiconductor device containing programmable logic components and interconnections. The logic can be programmed to perform the functionality of different operations ranging from that of basic logic gates (such as AND, OR, XOR, NOT) to more complex combinatorial functions such as matrix operations. FPGAs often include optimized components, such as memory and dedicated hardware blocks, ranging from multipliers to microprocessors. FPGAs come in many different sizes, ranging from a few thousand to hundreds of thousands of logic cells, and with built-in dedicated hardware. Evaluation boards with a mounted FPGA and support circuits such as Ethernet, USB, external memories etc. are also available, which makes them suitable for experimental implementations.

An ASIC is an integrated circuit (IC) that is customized for a specific use, rather than intended for general-purpose usage. The desired functionality is designed in hardware developing tools and sent to manufacturing.

The choice of which type of platform (µPs, digital signal processors, ASIC, or FPGA) to use in implementation is sometimes hard to make. Ultimately it comes down to making trade-offs between the pros and cons in relation to the application that will be implemented, available recourses at the working place, and time/budget constraints. In our design the µPs and digital signal processors where not considered due to the drawbacks mentioned above. The choice of platform was between making an ASIC design or an FPGA design. The main advantages of designing with FPGAs are:

- The time from completing the design in a hardware description language to obtain a functional unit is limited. The design is programmed into the FPGA and can be tested immediately. This is especially important when the development time is limited.
- FPGAs are very good prototyping vehicles. If the application is going to use an FPGA the step from prototyping to final product is negligible.
- Developing prototypes with FPGAs are cost effective compared to an ASIC.
- There are plenty of implementation and optimization tools to choose from. The number of companies developing and selling tools for developing and optimizing FPGA designs has exploded in the last couple of years. Also, the number companies making optimized IP blocks are rapidly growing.

The main disadvantages of designing with FPGAs instead of making an ASIC are:

- *Area & Utilization*. Limited design area combined with low utilization of the existing gates in the FPGA.
- *Performance*. If the logic must run at high clock rates an ASIC design is preferable.
- *Power*. FPGAs still consumes significantly more power than a low power ASIC design.

However, it is no doubt that designing with FPGAs is becoming more and more popular even for implementing high performance digital communication systems. The reason for that is the rapid technical development of FPGA circuits in the last year. This has closed the gap between ASIC and FPGA, reducing the impact of the main disadvantages of using FPGAs. Today the FPGAs are mainstream devices and many manufacturers have gone from developing ASICs to using high-end FPGAs in their products instead. The mobile communication field has also started consider to use FPGAs for their new high speed communication equipment [20],[21].

The effective programmable area and the achievable clock speed of an FPGA have grown significantly in recent years. Reducing the power consumption of FPGAs has until now not been a driving force in the FPGA world since it has been mainly for prototyping and testing of systems that are not that power sensitive. But times are changing and power is now a main concern in the FPGA society and much is done to develop FPGAs with lower power consumption.

3.2 Flexible platform

Another reason for choosing the FPGA as the implementation platform for multiantenna systems is in this case the target application. The Capon beamforming algorithm has been implemented in hardware for usage in a channel sounder system. Both physical parameters, being the transmission and receiver antennas, and software parameters, such as the multiantenna algorithm, are often changed in the channel sounder equipment when investigate different scenarios. This means that the underlying hardware implementation must be easy to adapt to changes. Also, for the channel sounder system hardware and power constraints are relaxed.

As discussed in 2.4.2, one way of meeting the requirements of a flexible platform is to use scalable matrix building blocks. In this way a new algorithm could easily be implemented in hardware in a limited time. To be able to do this, a flexible platform is needed and therefore, the choice between an ASIC and an FPGA platform was easy. In this case by using an FPGA platform multiantenna algorithms can easily be investigated without long waiting times for a chip to come back from manufacturing.

3.3 Summary

When implementing algorithms in hardware, it is important to choose the right platform for one’s needs. As discussed in previous chapters, high performance algorithms for multiantenna systems requiring high throughput rate was the target for implementation in this thesis. To be able to meet the performance demands, a specialized architecture such as an ASIC or an FPGA is needed.

The target application in this thesis was a channel sounder. Since the configuration of the channel sounder often changes and many different types of algorithms will be tested, it is important to choose a flexible and platform. From these criteria an FPGA platform is the best choice.

Part I

Complex Valued Division

Chapter 1

Introduction

Complex numbers are an extension of the ordinary numbers used in everyday engineering with the unique property of representing and manipulating two variables as a single quantity. In digital signal processing, complex numbers may shorten complicated equations and enable computational techniques that are difficult or impossible with real numbers alone. Unfortunately, complex arithmetic and techniques are computationally demanding, and require a significantly greater number of computational steps than the corresponding real operations. It also takes a great deal of study and practice to use them effectively, or as Steven W. Smith said in his book, The Scientist and Engineer's Guide to Digital Signal Processing [22],

"Many scientists and engineers regard complex techniques as the dividing line between DSP as a tool, and DSP as a career."

Techniques involving complex arithmetic occur frequently in a wide range of applications, such as audio and video signal processing, digital communications, weather forecasting, economic forecasting, seismic data processing, analysis and control of industrial processes, and in mathematical operations and transforms [23]. Many of these applications include computationally heavy algorithms that need to be performed in real-time with good numerical accuracy [15]. To be able to comply with such demands, dedicated hardware implementations of these algorithms is a necessity. Therefore, it is important to derive efficient hardware architectures handling complex numbers, which can be used in realizations of applications today and in the future.

The building blocks for implementation of multiantenna algorithms implemented in this thesis consist of matrix operations involving complex valued data. To implement these matrix operations, arithmetic building blocks such as addition/subtraction, multiplication, and division are used. To be able to implement efficient and numerically accurate matrix operations, equally efficient and accurate implementations of these arithmetic building blocks must be at hand. The most computationally heavy arithmetic operation is the complex valued division, which is also the most challenging arithmetic operation to implement with good throughput rate and without too high resource consumption. In this part of the thesis, three hardware architectures for complex valued division are suggested.

In chapter 2, complex valued division is discussed, and two ways of dividing complex numbers with each other, division by fraction and by transformation, are presented. Chapter 3 investigates two hardware architectures for complex valued division by fraction based on Smith’s formula, while chapter 4 looks closer at hardware architecture for complex valued division by transformation based on CORDIC are investigated. Chapter 5 summarizes the three hardware implementations and compares them to each other in terms of throughput rate and resource consumption.

Chapter 2

Complex valued division

Dividing two complex numbers with each other is a computationally demanding operation. During algorithm development, divisions, and especially complex valued divisions, are commonly tried to be avoided or replaced by an estimation function. However, with a growing number of applications requiring high numerical precision, in many cases such avoidance or replacement will result in a severe degradation of the numerical accuracy.

In recent years, a great deal of attention has been given to the development and implementation of efficient division algorithms. However, most of the proposed implementations have targeted real valued division, and very few have dealt with complex valued division. Complex valued division using the conventional formula shown in equation (2.6) is often implemented in software. Software implementations of complex valued division often result in low calculation speed, making them unsuitable to use in modern high performance signal processing applications. Therefore, it is increasingly important to derive hardware architectures fulfilling contradictory requirements regarding numerical accuracy, calculation speed, latency, and area consumption.

2.1 Definition of complex numbers

Complex numbers are an extension of the real numbers by the inclusion of the imaginary unit *i*, satisfying $i^2=-1$. Every complex number can be written in the form $a+ib$, where *a* and *b* are real numbers, denoting the real part and the imaginary part of the complex number, respectively. Pairs of complex numbers can be added, subtracted, multiplied, and divided in a similar manner to that of real numbers. The set of all complex numbers is denoted by $\mathbb{C}$.

A complex number can be viewed as a point or a position vector in a two-dimensional Cartesian coordinate system, called the complex plane or Argand diagram. Figure 2.1 shows the complex plane with a real and an imaginary axis and a unity circle. The Cartesian coordinates of a complex number are denoted by the real part *a* and the imaginary part *b*, while the circular coordinates are denoted by

$$r = |z| = \sqrt{a^2 + b^2}\ , \tag{2.1}$$

called the absolute value or modulus, and

$$\theta = \arg(z) = \arctan(b/a) + n\pi \text{ where } n = \begin{cases} 0 \text{ if } x > 0 \\ 1 \text{ if } x < 0 \end{cases}, \tag{2.2}$$

called the complex argument of z. Together with Euler's formula

$$e^{i\theta} = \cos\theta + i\sin\theta \tag{2.3}$$

the complex number z can be expressed in exponential form as

$$z = a + ib = r(\cos\theta + i\sin\theta) = re^{i\theta} \tag{2.4}$$

where

$$\begin{cases} a = r\cos\theta \\ b = r\sin\theta \end{cases} \tag{2.5}$$

The complex conjugate of the complex number $z = a + ib$ is defined to be $\overline{z} = a - ib$. As shown in figure 2.1, $\overline{z}$ is the reflection of z in the real axis.

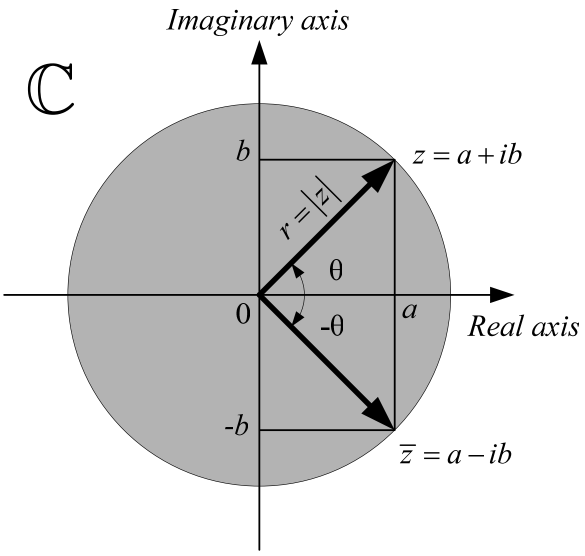


Figure 2.1 The definition of complex numbers in the complex plane $\mathbb{C}$.

There are several ways of dividing a complex number *a+ib* by another complex number *c+id,* whose magnitude is non-zero, but the most commonly used methods are division by fraction and division by transformation.

2.2 Division by fraction

In division by fraction, the division is expressed as a fraction between two numbers. Both the numerator and denominator are multiplied by the complex conjugate of the denominator. This causes the denominator to simplify into a real number as shown in equation (2.6), in this thesis referred to as the pen-and-paper algorithm.

$$\begin{aligned} z = a + ib = \frac{c+id}{e+if} = \frac{(c+id)(e-if)}{(e+if)(e-if)} = \frac{(ce+df)+i(de-cf)}{e^2+f^2} = \\ = \left(\frac{ce+df}{e^2+f^2}\right) + i\left(\frac{de-cf}{e^2+f^2}\right) \quad \text{when } e, f \neq 0 \end{aligned} \tag{2.6}$$

2.3 Division by transformation

In division by transformation, both complex numbers are converted using equation (2.1) and (2.2) into the exponential form described by equation (2.4). A real valued division can then be performed between the absolute values of the complex numbers, r_1 and r_2, and the quotient can easily be derived by subtraction of the two arguments, θ_1 and θ_2, as shown in equation (2.7).

$$z = \frac{c+id}{e+if} \xrightarrow{\text{Transform}} \frac{r_1 e^{i\theta_1}}{r_2 e^{i\theta_2}} = \frac{r_1}{r_2} e^{i(\theta_1-\theta_2)} = r_3 e^{i\theta_3} \xrightarrow{\text{Transform}} g + ih \tag{2.7}$$

The result can then, if necessary, be transformed back into a rectangular form using equation (2.5).

Chapter 3

Complex valued division by fraction

3.1 Algorithms for division by fractions

A straightforward way to implement a complex valued division is, as discussed in section 2.2, to use the conventional pen-and-paper algorithm presented in equation (2.6). As shown in figure 3.1, implementing the pen-and-paper algorithm will require 2 real divisions, 6 multiplications, and 3 additions/subtractions to perform a complex valued division.

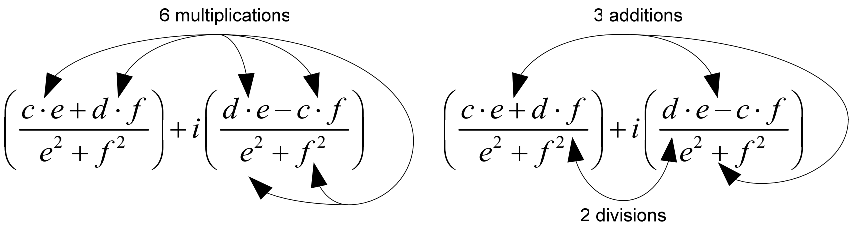

6 multiplications
$\left(\frac{c \cdot e + d \cdot f}{e^2 + f^2}\right) + i\left(\frac{d \cdot e - c \cdot f}{e^2 + f^2}\right)$
3 additions
$\left(\frac{c \cdot e + d \cdot f}{e^2 + f^2}\right) + i\left(\frac{d \cdot e - c \cdot f}{e^2 + f^2}\right)$
2 divisions

Figure 3.1 The required number of arithmetic operations needed to implement the pen-and-paper complex valued division algorithm.

The pen-and-paper algorithm is considered to have a high computational complexity and would consume lots of resources if implemented in hardware. This algorithm also suffers from the risk of overflow and/or underflow in intermediate computations of the denominator e^2+f^2. An overflow/underflow of the denominator could induce a severe numerical error, that will affect the accuracy of the result, thus rendering the calculation useless [25],[26].

Taking these drawbacks into account, the algorithm in its current state is not suitable for hardware implementation.

3.1.1 Reducing hardware complexity

Before implementing arithmetic algorithms in hardware, a careful analysis of the complexity of the algorithms should be performed. A possible reduction of the complexity could result in a number of improvements, such as an increase of the overall computation speed, lower power consumption, a reduction of area, and simplification of the control circuitry, etc. [23],[27]. Therefore, it is important to do a careful analysis and try to reduce the arithmetic complexity of an algorithm as much as possible. One way to reduce the complexity is to use a numerical strength reduction technique [23],[28]. Basic arithmetic operations can be ranked in terms of required hardware resources needed for implementation. The ranking of arithmetic operations, starting with the operation requiring the most resources (the strongest operation), is division, followed by multiplication, addition/subtraction, and finally bit-shift that corresponds to multiplications and divisions by powers of two.

When applying the strength reduction technique, a strong operation is traded for one or more weaker operations. This is done by either rewriting the algorithmic expression or by transforming a high ranked operation into several low ranked operations. Both procedures aim to improve the performance in terms of area, speed, and power consumption. Example 1 shows how strength reduction can be applied to complex multiplication through rewriting.

Example 1

Equation (3.1) shows how a multiplication of two complex values is usually calculated. The complex multiplication requires 4 real valued multiplications, and 2 additions/subtractions to be realized in hardware.

$$\left(x+iy\right)\cdot(z+iw)=xz+ixw+iyz-yw=\left(xz-yw\right)+i(xw+yz) \tag{3.1}$$

The expression can be rewritten by adding the expression *+xw-xw* to the real part of the computation and *+yw-yw* to the imaginary part of the expression as shown in equation (3.2). The term *w(x-y)* can then be extracted in both the real and the imaginary part.

$$\begin{aligned}\Re&=\left(xz-yw\right)=xz-yw+xw-xw=x(z-w)+w(x-y)\\ \Im&=\left(xw+yz\right)=xw+yz+yw-yw=y(z+w)+w(x-y)\end{aligned} \tag{3.2}$$

The term *w(x-y)* only needs to be computed once, resulting in savings of computational resources. The complex multiplication implemented as shown in equation (3.2) requires 3 real valued multiplications and 5 additions/subtractions to realize in hardware.

□

By applying a strength reduction technique and rewriting the expression, one multiplication has been traded for 3 additions, which in some cases may lead to significant savings of resources.

Caution!

Strength reduction techniques should be used with caution. Not all trades of strong operations for multiple weaker operations are as beneficial as they first seem to be. Trading a multiplication for 3 additions may look beneficial at a first glance. However, depending on how the arithmetic operations are implemented, 3 additions might actually consume more area and power than a single multiplier.

Another pitfall of strength reduction is when implementing in an FPGA. Some FPGAs have built-in dedicated hardware blocks such as optimized multipliers. If you trade a multiplier in your design, that could have been mapped onto one of these optimized multipliers, you will consume unnecessarily large amounts of hardware in the FPGA and get a slower design. When dealing with FPGA designs, all optimization and reduction strategies that are applied to your design must be done with respect to available resources in the FPGA.

3.1.2 Reducing complexity of the pen-and-paper algorithm

Strength reduction can be applied to the algorithm in equation (2.6) by adding the term *+fc-fc* to the numerator of the real part and the term *+fd-fd* to the numerator of the imaginary part. By rewriting the two expressions, as shown in equation (3.3), the term *f(c+d)* now occurs in both the real and the imaginary part of the algorithm and will only be calculated once. In this way, 1 multiplication is traded for 3 additions, which in most cases is beneficial.

$$\begin{aligned}\Re(z) &= \frac{ce+df}{e^2+f^2} = \frac{ce+df+\boldsymbol{fc-fc}}{e^2+f^2} = \frac{c(e-f)+\boldsymbol{f(c+d)}}{e^2+f^2} \\ \Im(z) &= \frac{de+cf}{e^2+f^2} = \frac{de+cf+\boldsymbol{fd-fd}}{e^2+f^2} = \frac{d(e+f)-\boldsymbol{f(c+d)}}{e^2+f^2}\end{aligned} \tag{3.3}$$

Equation (3.4) shows the strength reduced equation which would require 2 real divisions, 5 multiplications, and 6 additions/subtractions to implement in hardware.

$$z = a+ib = \frac{c+id}{e+if} = \left(\frac{c(e-f)+f(c+d)}{e^2+f^2}\right) + i\left(\frac{d(e+f)-f(c+d)}{e^2+f^2}\right) \tag{3.4}$$

By rearranging the computational order of the algorithm, as shown in equation (3.5), one division can be traded for two multiplications. In this way the division is carried out only once and is then multiplied with the numerator of the real and imaginary part. A division 1/y, where y is an arbitrary real number, is usually cheaper to implement in hardware than a division by two arbitrary real numbers, x/y. Implementing equation (3.5) in hardware would require 1 real division, 7 multiplications, and 6 additions/subtractions. In this way we trade one division for one 1/y division and two multiplications.

$$\left(\frac{1}{e^2+f^2}\cdot\left(c\cdot(e-f)+f\cdot(c+d)\right)\right)+i\left(\frac{1}{e^2+f^2}\cdot\left(d\cdot(e+f)-f\cdot(c+d)\right)\right) \tag{3.5}$$

However, all the above equations still suffer from the risk of overflow and underflow in intermediate computations due to the term e^2+f^2 in the denominator. A more numerically stable algorithm is therefore sought for.

3.1.3 Smith's algorithm

There are two ways of reducing the risk of overflow and underflow in the computation of the denominator in equation (3.5). One way is to increase the dynamic range by increasing the wordsize of the computational unit, which could be costly in hardware. Another way to reduce the risk is to rewrite the algorithm. R.L. Smith [24] showed that by multiplying the nominator and the denominator with either *1/e* or *1/f*, a more robust algorithm can be achieved. If for example *1/e* is used, the algorithm in equation (2.6) can be rewritten as shown in equation (3.6).

$$\left(\frac{ce+df}{e^2+f^2}\right)+i\left(\frac{de-cf}{e^2+f^2}\right)=\left(\frac{\frac{1}{e}ce+\frac{1}{e}df}{\frac{1}{e}e^2+\frac{1}{e}f^2}\right)+i\left(\frac{\frac{1}{e}de-\frac{1}{e}cf}{\frac{1}{e}e^2+\frac{1}{e}f^2}\right)=\left(\frac{c+d(f/e)}{e+f(f/e)}\right)+i\left(\frac{d-c(f/e)}{e+f(f/e)}\right) \tag{3.6}$$

Equation (3.6) is numerically stable only when the absolute of operand *e* is greater than or equal to the absolute of operand *f*. By multiplying the nominator and the denominator with the term *1/f* we get a similar equation that is numerically stable for the reverse condition as shown in [25]. The complete formula is shown in equation (3.7). Depending on the sizes of the operands *e* and *f*, one of the two expressions in (3.7) should be used. This algorithm is much more numerically robust than the algorithm in equation (2.6) if computed in the order indicated by the parentheses [25]. Smith's algorithm requires 3 divisions, 3 multiplications, 3 additions/subtractions, and a comparison unit to determine the sizes of the operands *e* and *f*.

$$z=a+ib=\frac{c+id}{e+if}=\begin{cases}\left(\frac{c+d(f/e)}{e+f(f/e)}\right)+i\left(\frac{d-c(f/e)}{e+f(f/e)}\right) & \text{(when } |e|\geq|f|\text{)}\\ \left(\frac{d+c(e/f)}{f+e(e/f)}\right)+i\left(\frac{c-d(e/f)}{f+e(e/f)}\right) & \text{(when } |e|\leq|f|\text{)}\end{cases} \tag{3.7}$$

The computational order of the algorithm in equation (3.7) can be rearranged in the same manner as was done with equation (3.4). The resulting equation is shown in equation (3.8).

$$z = a + ib = \frac{c+id}{e+if} = \begin{cases} \left(\frac{1}{e+f(f/e)} \cdot (c+d(f/e))\right) + i\left(\frac{1}{e+f(f/e)} \cdot (d-c(f/e))\right) & (when\ |e| \geq |f|) \\ \left(\frac{1}{f+e(e/f)} \cdot (d+c(e/f))\right) + i\left(\frac{1}{f+e(e/f)} \cdot (c-d(e/f))\right) & (when\ |e| \leq |f|) \end{cases} \tag{3.8}$$

In this way, 1 division is traded for 2 multiplications. The algorithm in (3.8) requires 2 divisions, 5 multiplications, 3 additions/subtractions, and a comparison unit to determine the sizes of the operands *e* and *f*.

3.1.4 Resource comparison

Table 3.1 summarizes the hardware resources needed for each complex valued division algorithm presented in the previous sections.

Table 3.1 Comparison of resources for the complex valued division algorithms.

Algorithm	# div.	# mult.	# add.	Other resources
Equation (2.6)	2	6	3	---
Equation (3.4)	2	5	6	---
Equation (3.5)	1	7	6	---
Equation (3.7)	3	3	3	operand comparison*
Equation (3.8)	2	5	3	operand comparison*

* A comparison unit can typically be implemented using a subtractor. The two operands are subtracted from each other and the sign bit will indicate which operand is the largest one.

As shown in table 3.1, Smith's algorithm (equation 3.8) is less expensive in terms of hardware resources (one multiplication) compared to the original pen-and-paper algorithm (equation 2.6) and to one of the strength reduced versions (equation 3.4). However, it is not wise to base the decision of which algorithm to use only on the resource estimation presented in table 3.1. There are several other parameters that must also be taken into account before making the decision, for example:

1. The individual sizes of the arithmetic units. If a division consumes about two multiplications worth of resources, equation (3.5) may be more expensive to implement than equation (3.8). If it consumes about 4 multiplications worth of resources, equation (3.5) will be less expensive than equation (3.8).

2. Available resources in the FPGA. Limited amounts of embedded multipliers, adders, MACs, etc. in the FPGA may rule out some of the algorithms for implementation.
3. Various hardware constraints such as latency, output production rate, numerical aspects, etc. may play an important role in the choice of algorithm to implement. For instance, scheduling an algorithm with limited hardware resources may result in an insufficient output sample rate. However, a more resource expensive algorithm may reach the targeted output production rate.

3.1.5 Numerical comparison

The numerical properties of the algorithms in equations (2.6), (3.4), (3.5), (3.7), and (3.8) were compared using a fixed-point MATLAB Simulink simulation. The equations were implemented, with the computational order indicated by its parentheses, using identical hardware components and rounding technique. A test vector with all combinations of operand input values was constructed. During simulations overflows and underflows were registered and the result was compared. The first column in table 3.2 indicates the total number of overflows and underflows compared to the number of divisions performed in the test, expressed in percentage.

Table 3.2 Comparison of numerical accuracy between the complex valued division algorithms.

Algorithm	# overflow/underflow	Accuracy error
Equation (2.6)	14%	≤ 2 LSB
Equation (3.4)	10%	≤ 2 LSB
Equation (3.5)	9%	≤ 2 LSB
Equation (3.7)	1%	< 1 LSB
Equation (3.8)	4%	≤ 1 LSB

The result shows that equation (3.7) has the best numerical accuracy and the least amount of overflows/underflows compared to the other equations. If the information from table 3.1 and table 3.2 is combined, equation (3.7) seems to be the best algorithm to use when computing a complex valued division. One may argue that equation (3.8) is a better choice since it consumes less hardware resources and only has slightly poorer numerical accuracy. However, as will be shown in section 3.3, a division is comparable to two multiplications in terms of hardware resources, which strengthens the arguments for choosing equation (3.7).

Caution!

The results in table 3.2 are strongly dependent on how the hardware blocks are implemented. However, they give an indication of the relative order in terms of numerical accuracy.

3.2 Architectures for complex valued division

3.2.1 Architectures and schedules for Smith's algorithm

Smith's algorithm presented in equation (3.7) can be implemented in several different ways depending on the available hardware arithmetic units and their individual constraints such as latency. A scheduling analysis comparing different computational orders must be done before a hardware implementation can be initiated [29]. Depending on the order in which the arithmetic operations are performed, and their individual computation times, several hardware architectures are possible. Assuming that all arithmetic operations require one cycle to produce a result, two fundamental architectures can be identified [30]:

1. Parallel architecture: The parallel architecture is optimized for low latency and high throughput rate. This architecture will generate a result already in 4 cycles but it will consume a large amount of hardware resources. An as-soon-as-possible (ASAP) scheduling of the parallel architecture is shown in figure 3.2.

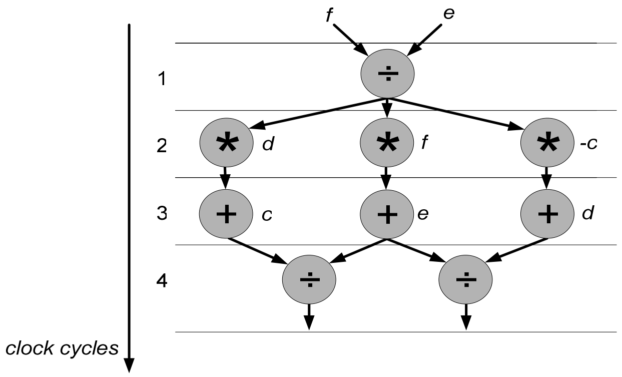


Figure 3.2 An ASAP scheduling of the parallel architecture.

2. Multiplexed design: The suggested multiplexed architecture is based on a limited amount of hardware resources (one arithmetic unit of each kind) and it uses feedback loops and time multiplexing strategies to reuse idle arithmetic blocks units. This approach results in an area conservative architecture but with a larger start up latency and lower throughput rate than the parallel architecture. An as-soon-as-possible scheduling of the multiplexed architecture is shown in figure 3.3.

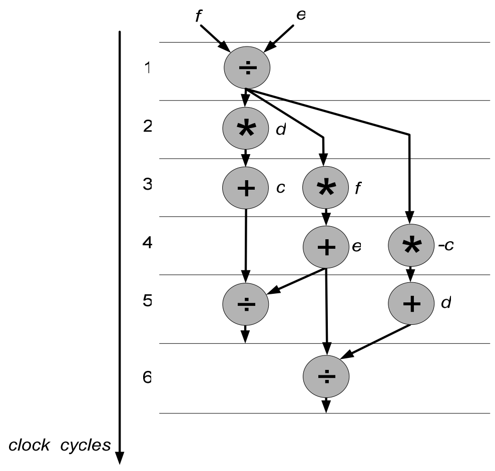


Figure 3.3 An ASAP scheduling of the multiplexed architecture.

The final scheduling that is going to be used in the FPGA hardware implementation can only be done after the arithmetic units have been chosen. The implementation of the arithmetic units will have a huge impact on the total architecture of the complex valued divider in terms of hardware size, numerical accuracy, and execution speed [27]. The largest and most computationally heavy unit in both architectures is the real divider. In this thesis, the real divider was implemented using two redundant modified-Booth multipliers, and a small lookup table (LUT) with normalized and optimized table entries for high numerical accuracy (appendix A). All other building blocks were implemented using Xilinx optimized building blocks. The arithmetic building blocks used in the FPGA design require the following number of cycles to produce an output value:

- The real divider: 2 cycles.
- The multiplier unit: 1 cycle.
- The adder/subtractor unit: 1 cycle.

3.2.2 Scheduling of the parallel architecture

When the computation times are set the ASAP schedule can be revised. Figure 3.4 (a) shows the revised ASAP schedule of the parallel design, describing in which cycle each arithmetic unit is in use, and which arithmetic computation is performed. In cycle 1 and 2 the real division of operand *f* with operand *e* is performed. The real division is pipelined into two steps, thus taking two cycles to produce the result. In cycle 3 and 4 the multiplications and additions of the operands are performed in parallel and in cycle 5 and 6 the real and the imaginary parts are calculated by the two real divisions. The resulting real and imaginary parts are delivered at the same time at the end of cycle 6. Figure 3.4 (b) shows the corresponding activity schedule of the parallel architecture which is used in later scheduling.

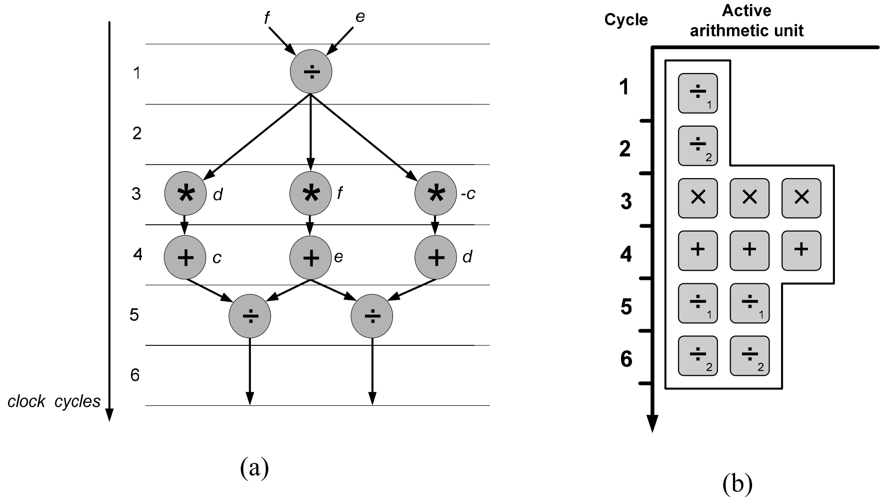


Figure 3.4 Scheduling of a parallel design of Smith's algorithm.

In many high performance signal processing systems, multiple divisions are performed after one another in a sequence. The parallel architecture can be scheduled so that several overlapping division operations can be performed simultaneously as shown by figure 3.5. Each set of black or grey squares represents a single division in progress. When the scheduled design is fully utilized, through software pipelining [23],[29], it produces one output value every cycle as indicated by the black dots in the output column of figure 3.5.

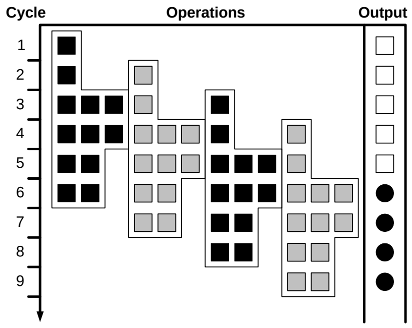


Figure 3.5 Scheduling of four simultaneous divisions in the parallel design of Smith's algorithm.

The start-up latency of the parallel design is 6 cycles and a 100% utilization of the arithmetic units is maintained at all times. The hardware cost to implement the design is 3 real valued dividers, 3 multipliers, and 3 adders.

3.2.3 Scheduling of the time multiplexed design

Figure 3.6 (a) shows the revised schedule of the multiplexed architecture, describing in which cycle each arithmetic unit is in use, and which arithmetic computation is performed.

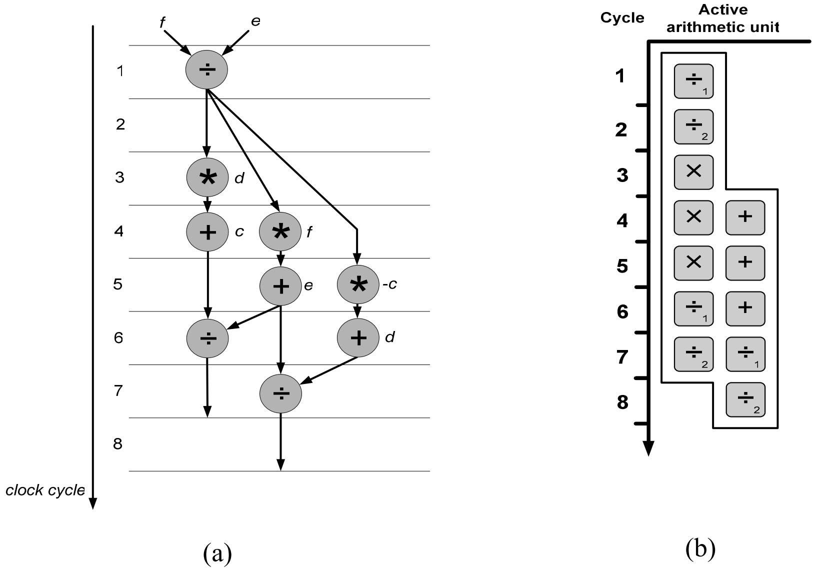


Figure 3.6 Scheduling of a time multiplexed design of Smith's algorithm.

In cycle 1 the first part of the real divider unit is in use, calculating the term *f/e* in equation (3.7). The computation of the term *f/e* is finished in cycle 2, and in cycle 3 operand *d* is multiplied with *(f/e)*. In the next cycle, two operations, *c+d(f/e)* and *f(f/e)*, are performed simultaneously, occupying the adder and the multiplier unit, and in cycle 5, *e+f(f/e)* and *-c(f/e)* are calculated simultaneously. The first part of the real division, *c+d(f/e)* with *e+f(f/e),* is computed in cycle 6. The real part of the complex valued division is produced at the end of cycle 7, while the imaginary part is produced at the end of cycle 8. Hence, a complete complex valued division, using only three arithmetic units, is performed in only 8 cycles. Figure 3.6 (b) shows an activity schedule of the multiplexed architecture which is used in later scheduling.

As shown in figure 3.6 not all hardware block are utilized in every cycle. This means that there are hardware resources available that can be used to perform another complex division at the same time. Figure 3.7 shows two alternatives of how the time-multiplexed design can be scheduled to perform two simultaneous computations at once.

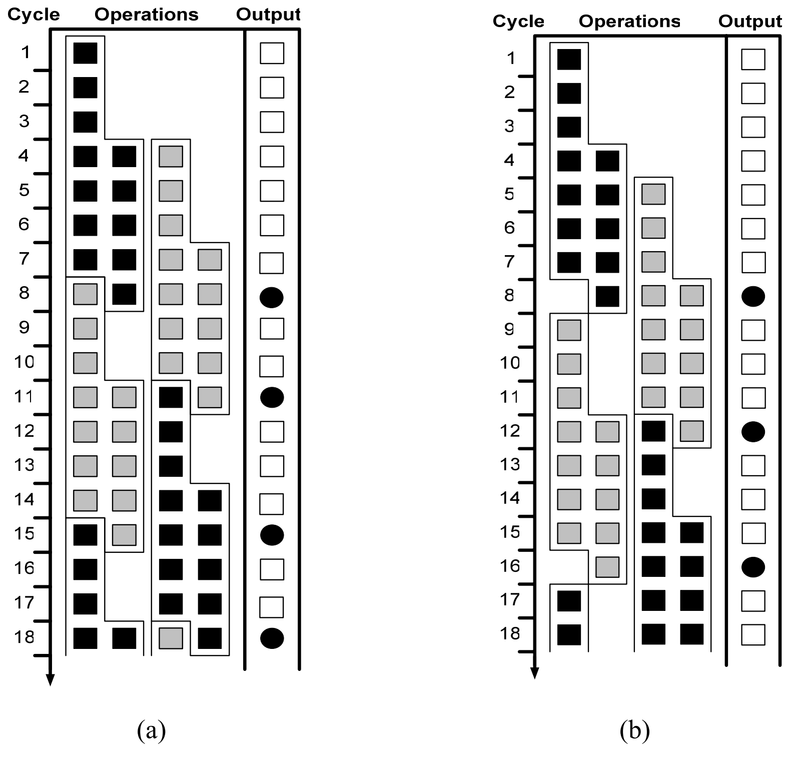


Figure 3.7 Two different ways of scheduling the time multiplexed design of the complex valued division algorithm.

The black and grey boxes in 3.7 (a) and (b) indicate the two ongoing computations. The black dots in the column marked output indicates when a complex output value is produced.

Both schedules have a start-up latency of 8 cycles before the first complex value is produced. In continuous operation, the schedule in (a) has a varying output production rate, producing on average an output value every 3.5 cycles (3 – 4 – 3 – 4 – …) while (b) has a constant production rate (4 – 4 – 4 – 4 – …) of an output value every 4 cycles.

The hardware utilization is not 100% in every cycle for either schedule. Looking at the schedule in (a), a 100% utilization is for instance reached in cycle 7 and 8 (a full row), but only a 75% utilization in cycle 9 and 10. On average, the degree of hardware utilization in schedule (a) is 87.5%. In schedule (b) the hardware utilization degree is lower than in (a), reaching only 81.25%. The schedule in (a) was chosen for implementation due to the higher degree of hardware utilization and the marginally better production rate. If an irregular output production rate results in a more complicated control and data flow, the schedule in (b) should be chosen instead.

If implemented in an ASIC, the unused hardware blocks could be turned off when not used to save some power. However, as discussed in the design methodology in part IV of this thesis, when implementing in FPGA this power saving strategy is not applicable in the same manner.

3.3 FPGA implementation

Both the parallel and the time-multiplexed [31] architecture have been implemented in a Xilinx Virtex II XC2V1000 FPGA (speed grade 4) using fixed-point representation. The wordlength of the input operands was set to 8 bits. To keep rounding errors to a minimum, convergent rounding was used throughout the designs in this chapter [28]. This bias-free rounding scheme avoids any statistical bias since it will round upward about 50% of the time and downward about 50% of the time. Saturation arithmetic ensures that rounded results that overflow (or underflow) the dynamic range will be clamped to the maximum positive (or negative) value. The maximum accuracy error of the implementation is below 1 unit in the last position (ulp).

3.3.1 Parallel architecture

Figure 3.8 shows a block diagram of the hardware architecture of the parallel design. The hardware architecture is directly derived from the schedule of the parallel design in figure 3.4. The architecture has a linear data flow enabling several computations to be performed simultaneously as shown in figure 3.5. Pipeline registers are placed between each arithmetic unit. In an ASIC implementation these pipeline registers must be implemented, but in an FPGA each logic cell has build-in registers, which can be used without any cost.

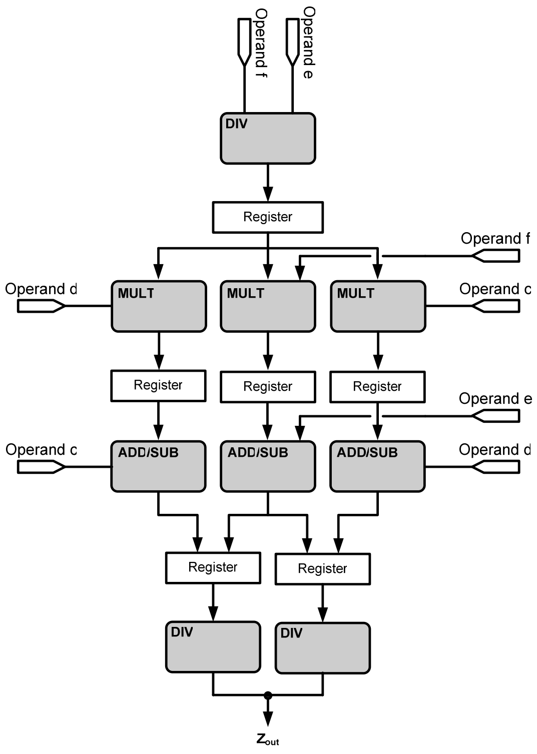


Figure 3.8 Block scheme of the hardware architecture of the parallel design.

Table 3.3 lists the amount of resources occupied by the design in the FPGA, and indicates the percentage of the total resources of the FPGA that was consumed. In average the implementation consumes 8% of the total resources in the FPGA. The implementation can be run at a maximum clock frequency of 147 MHz. New FPGAs have built-in high performance hardware units, such as dedicated multipliers. It is recommended that multipliers in a design are mapped onto these dedicated multipliers if possible, since they offer higher performance than multipliers constructed from slices. Furthermore, it frees up slices that can be used to implement other functions. A slice in an FPGA normally consists of two flip-flops, two lookup tables (LUTs) and some associated multiplexers, carry and control logic. BRAMs are dual-ported block RAMs, which in this design are acting as a lookup table for the real valued division. Three BRAMs were used in this design.

Table 3.3 Consumed resources of the parallel architecture implementation.

	Eq. (3.7) Parallel
Number of Slices	436 (8%)
Number of Slice Flip Flops	820 (8%)
Number of 4 input LUTs	799 (7%)
Number of BRAMs	3 (7%)
Maximum frequency (MHz)	147

Figure 3.9 (a) shows the routed floorplan of the FPGA implementation and (b) shows the placement of the arithmetic building blocks and the lookup tables in the design. The hardware building blocks have been placed by the place and route tool fairly clustered in one part of the FPGA. The placement and routing of the design can be modified to one's liking by either manually placing individual parts or by defining placement constraints in the place and route tool. If no placement constrains are defined, and the design occupies only a few cells, the place and route tool has a tendency to spread out the design in the FPGA. When the number of free slices decreases or defined timing constraints must be met, the tool will compact the design. The elongated part in the upper left corner is the BRAMs and the nine clusters correspond to the nine arithmetic blocks in the architecture.

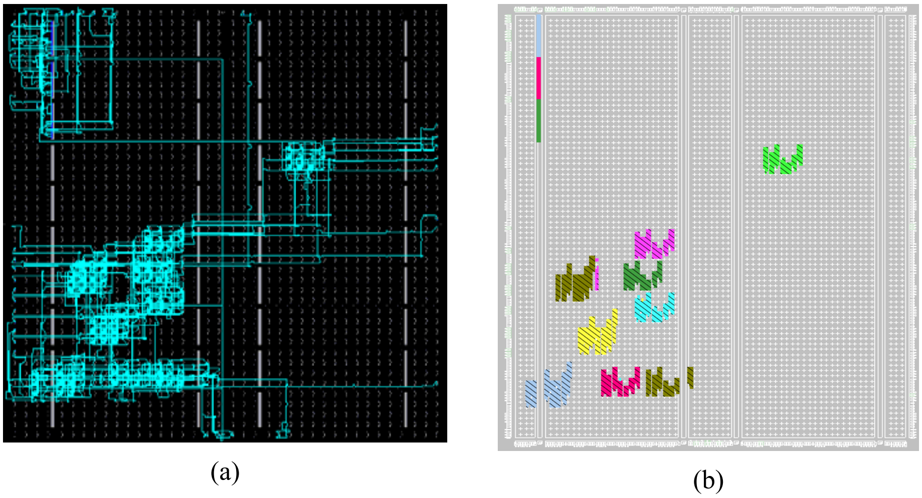


Figure 3.9 (a) Routed floorplan of the FPGA implementation of the parallel architecture. (b) Placement of the arithmetic building blocks.

3.3.2 Multiplexed architecture

Figure 3.10 shows a block diagram of the hardware architecture of the multiplexed design. The hardware architecture is derived from the schedule of the time multiplexed design in figure 3.6 and the scheduling in 3.7 (b) to allow two simultaneous complex valued division operations. The multiplexers and the delays regulate the data flow between the arithmetic units to utilize the hardware. As with the previous design, registers have been inserted in the block diagram for clarification purposes only. In an ASIC implementation these pipeline registers must be implemented, but in an FPGA each logic cell has build-in registers, which can be used without any cost.

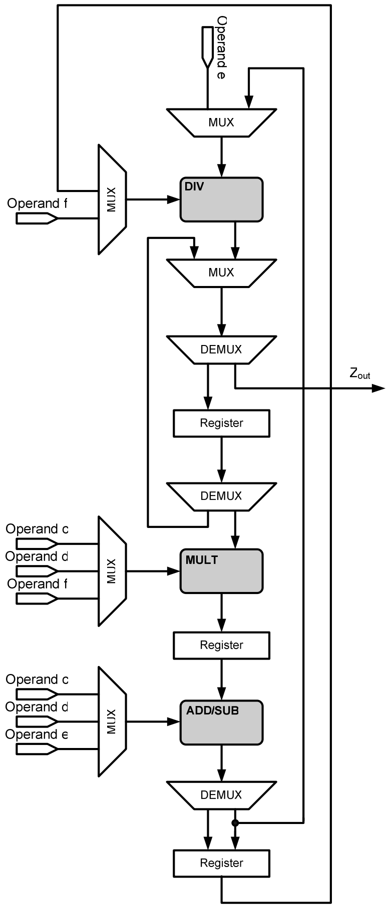


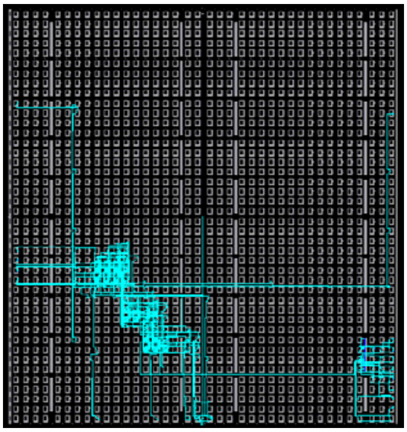
Figure 3.10 Block scheme of the hardware architecture of the time multiplexed design.

Table 3.4 lists the amount of resources occupied by the design in the FPGA and indicates the percentage of the total resources of the FPGA that was consumed. In average the implementation only consumes 2% of the total resources in the FPGA. The implementation can be run at a maximum clock frequency of 156 MHz.

Figure 3.11 (a) shows the routed floorplan of the FPGA implementation of the multiplexed complex valued division architecture, and in (b) the cells occupied by the design are shown. The elongated part in the lower right of figure 3.11 (b) is the BRAM and the other three parts in the lower left are the arithmetic building blocks and the flow control circuitry.

Table 3.4 Consumed resources of the multiplexed architecture in the FPGA.

	Eq. (3.7) Parallel
Number of Slices	146 (2%)
Number of Slice Flip Flops	276 (2%)
Number of 4 input LUTs	270 (2%)
Number of BRAMs	1 (2%)
Maximum frequency (MHz)	156


(a)

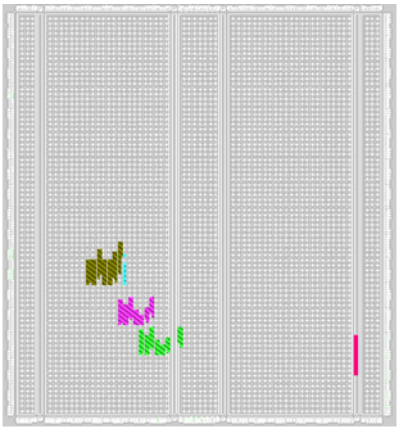

(b)

Figure 3.11 (a) Routed floorplan of the FPGA implementation of the multiplexed architecture. (b) Placement of the arithmetic building blocks.

3.3.3 Comparison of FPGA implementations

Table 3.5 shows a side by side comparison of the hardware implementation and architectural parameters of the two FPGA implementations. The parallel design has shorter startup latency and can accommodate 6 simultaneous complex valued divisions as compared to only 3 simultaneous operations in the multiplexed architecture. In continuous operation the parallel architecture produces one complex valued output every cycle compared to one complex value every 4 cycles in the multiplexed architecture.

The parallel architecture pays for the greater throughput rate and the number of simultaneous operations by consuming more hardware resources. In average the parallel architecture consumes about 4 times as many resources as the multiplexed architecture. The maximum achievable frequency of the two designs is virtually the same.

Table 3.5 Comparison between the parallel and the multiplexed implementation of a complex valued divider.

	Eq. (3.7) Parallel	**Eq. (3.7) Multiplexed**
Startup Latency (cycles)	6	8
Throughput rate (output/cycle)	1/1	1/4
Max simultaneous operations	6	3
Number of Slices	436 (8%)	146 (2%)
Number of Slice Flip Flops	820 (8%)	276 (2%)
Number of 4 input LUTs	799 (7%)	270 (2%)
Number of BRAMs	3 (7%)	1 (2%)
Maximum frequency (MHz)	147	156

When choosing which of the designs to use in an FPGA implementation, it all comes down to a trade-off between throughput rate and resource consumption. If resources are of importance then choose the multiplexed architecture, and if speed is the major concern then choose the parallel architecture. In an ASIC implementation, power is the third parameter that has to be taken into account. However, in FPGAs the power consumption is of less concern, since the FPGA in itself consumes the largest part and the added power consumption by the design is minimal in many cases.

Another way of implementing a complex valued division, which has grown in popularity with the introduction of special hardware architectures, is to transform the complex values to the polar coordinate system and to perform the computation in polar form instead. By popular belief, it is also considered to be much cheaper in terms of hardware resources. For comparison reasons, a complex valued division by transformation architecture was also implemented.

Chapter 4

Complex valued division by transformation

As discussed in chapter 2, complex valued division implemented with arithmetic building blocks as shown in chapter 3 is usually tried to be avoided since it consumes too much resources. Another way of implementing a complex division is by transformation. As shown in equation (2.7) in section 2.3 the complex valued division can be simplified if executed in the polar coordinate system instead of in the Cartesian coordinate system. Transforming the complex numbers from Cartesian coordinates into polar coordinates is done by using equation (2.1) and (2.2). When the division has been carried out by using equation (2.7), and the result must be presented in Cartesian coordinates, a transformation back into Cartesian coordinates is done by using equation (2.5). Implementing the complex valued division in this way would require in total 1 square root, 4 multiplications, 2 real divisions, 1 subtraction, 1 arctan operation, 1 sine operation, and 1 cosine operation. Using this strategy to divide two complex numbers would be very expensive in terms of hardware resources and is therefore not recommended.

Another way of doing the transformations and the operations in the polar coordinate system is to use the Coordinate Rotation Digital Computer (CORDIC) method. The CORDIC method has gained much popularity in the last ten years and has become the standard way of implementing trigonometric, hyperbolic, logarithmic, and some linear functions including complex valued division. The CORDIC method was first presented by Volder in 1956 [32]. The idea is based on rotating the phase of a complex number by multiplying it by a succession of constant values. By letting all multiplications be of powers of 2, they can be implemented using just shifts and additions and no actual multiplier is needed [28].

Compared to other approaches, CORDIC is a clear winner when it is important to save the number of gates required to implement trigonometric and linear functions (e.g. in an ASIC) [33]. On the other hand, when hardware multipliers are available (e.g. in an FPGA or microprocessor), methods such as the one presented in chapter 3 are generally faster than CORDIC but consume more hardware resources.

4.1 The CORDIC algorithms

If a vector $[x, y]^T$ is rotated by an angle θ in a Cartesian coordinate system, it results in the vector $[x', y']^T$ as shown by figure 4.1.

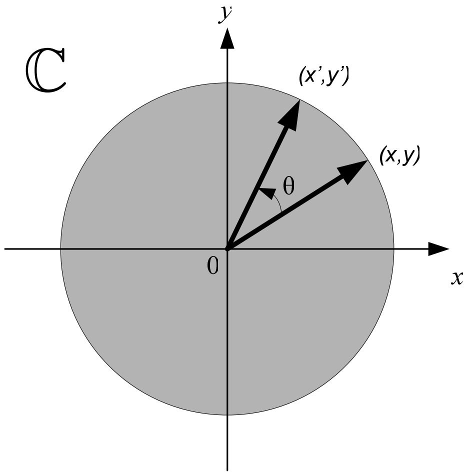


Figure 4.1 A rotation of θ degrees in a Cartesian coordinate system.

A general rotation can be expressed by the rotation transformation equations:

$$\begin{cases} x' = x\cos\theta - y\sin\theta \\ y' = y\cos\theta + x\sin\theta \end{cases} \tag{4.1}$$

By rearranging equation (4.1) we get:

$$\begin{cases} x' = \cos\theta \cdot [x - y\tan\theta] \\ y' = \cos\theta \cdot [y + x\tan\theta] \end{cases} \tag{4.2}$$

The rotation by an angle θ is implemented in the CORDIC algorithm as an iterative process, consisting of several micro-rotations during which the initial vector is rotated by pre-determined angles α_i. Any arbitrary angle θ can be represented by a set of n partial angles α_i. The direction of rotations is specified by the parameter d_i. An arbitrary angle θ can then be determined by the sum of all angle steps as follows:

$$\theta = \sum_{i=0}^{n-1} d_i \alpha_i \quad \text{where} \quad d_i \in \{-1, 1\} \tag{4.3}$$

The process to determine an angle is depicted in figure 4.2.

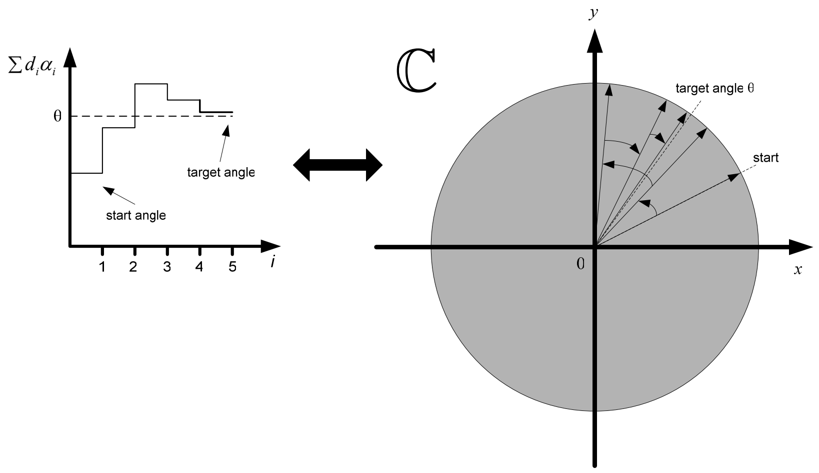


Figure 4.2 Successive micro-rotations for determination of the angle *θ*.

If the rotation angles are restricted to

$$\tan\theta = \pm 2^{-i} \quad \text{where } i = 0,1,\ldots,n-1 \tag{4.4}$$

the multiplication of the tangent term is reduced to a shift operation, which is very beneficial in hardware implementation. A variable z_i shown in equation (4.5), containing the accumulated partial sum of step angles, can be used to determine the sign of the next micro-rotation. For $z_0=\theta$ we have

$$z_{i+1} = z_i - d_i \cdot \tan^{-1}(2^{-i}) \tag{4.5}$$

where

$$d_i = \begin{cases} +1 & \text{if } z_i \geq 0 \\ -1 & \text{if } z_i < 0 \end{cases} \tag{4.6}$$

The CORDIC usually operates in one of two different modes, the "rotation" mode ($z_n \to 0$) or the "vectoring" mode ($y_n \to 0$). In the rotation mode the input vector is rotated by a specified angle, which is given as an argument. In the vectoring mode the input vector is rotated towards the x-axis, while storing the angle value required to make the rotation. The iteration equations of the CORDIC in rotation mode can be described as:

$$\begin{cases} x_{i+1} = x_i - y_i d_i 2^{-i} \\ y_{i+1} = y_i - x_i d_i 2^{-i} \\ z_{i+1} = z_i - d_i \cdot \tan^{-1}(2^{-i}) \end{cases} \quad \text{where } d_i = \begin{cases} +1 & \text{if } z_i \geq 0 \\ -1 & \text{if } z_i < 0 \end{cases} \quad \text{for } i = 0,1,...,n-1 \tag{4.7}$$

which provides the following output result:

$$\begin{cases} x_n = K_n(x_0 \cos z_0 - y_0 \sin z_0) \\ y_n = K_n(y_0 \cos z_0 + x_0 \sin z_0) \text{ where } K_n = \prod_{i=0}^{n-1} \sqrt{1+2^{-2i}} \\ z_n = 0 \end{cases} \tag{4.8}$$

However, a CORDIC rotation is not a pure rotation of the vector but a rotation and an extension (a.k.a. pseudo-rotation), where the scale factor K_n, given by (4.8), represents the increase in magnitude of the vector during the rotation process [34]. The resulting extension of a pseudo-rotation is shown in figure 4.3. When the numbers of rotations are fixed, the scale factor is a constant and when $n \to \infty$ the scale factor approaches the value 1.647. In most cases this extension by the scale factor K_n must be compensated for to achieve an accurate result.

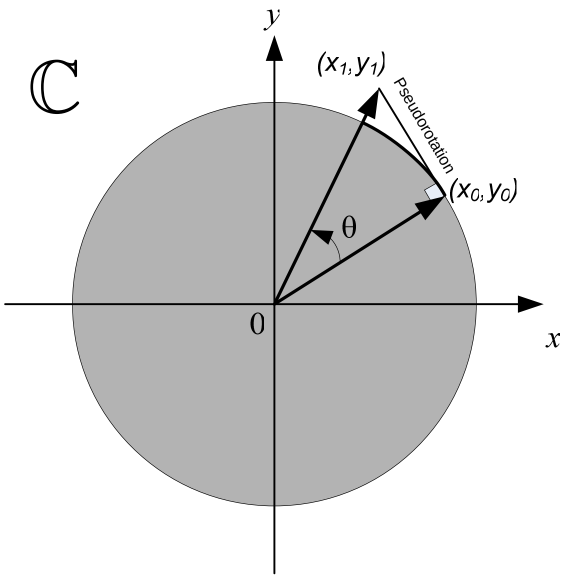


Figure 4.3 A pseudo-rotation in CORDIC.

The CORDIC operations can be performed in three different coordinate systems set by the coordinate system parameter *m*. The coordinate systems are the circular (*m=1*), the linear (*m=0*), and the hyperbolic (*m=-1*), as shown by figure 4.4. Depending on the coordinate system in which the rotations are done they correspond to different functions. A rotation in the circular coordinate system (*m=1*) corresponds to a cosine and sinus operation, while a similar rotation in the linear coordinate system corresponds to an addition.

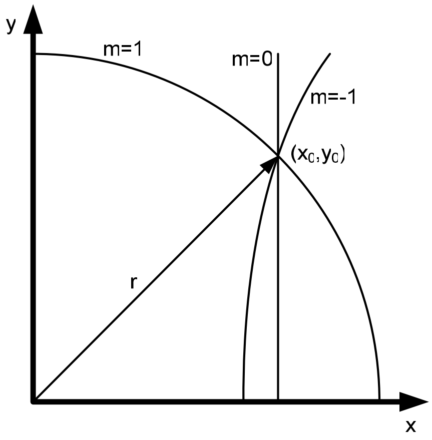


Figure 4.4 Plot of a vector *r* for different coordinate systems.

The rotation equations in equation (4.8) can be generalized into:

$$\begin{cases} x_{i+1} = x_i - m\delta_{m,i} y_i d_i \\ y_{i+1} = y_i - x_i \delta_{m,i} d_i \\ z_{i+1} = z_i - d_i \alpha_{m,i} \end{cases} \tag{4.9}$$

where the individual parameters are defined as:

$m =$ coordinate system parameter $\{1,0,\text{-}1\}$

$d_i =$ direction of rotation of the *i*th iteration $\{\text{-}1,1\}$

$\delta_{m,i} =$ incremental change of the *i*th iteration $\{0<\delta_{m,i} \leq 1\}$

$\alpha_{m,i} =$ partial angel of the *i*th iteration

$i =$ iteration index $\{0, 1,\ldots, n\text{-}1\}$

The resulting generalized partial scaling factors and partial angles for each rotation are shown in table 4.1, while table 4.2 summarizes the results after *n* iterations for the different modes of operation.

Table 4.1 The partial scaling factor and angle relative to coordinate system (*m*).

m	$\alpha_{m,i}$	$K_{m,i}^{-1}$
1	$\tan^{-1}\delta_{1,i}$	$\sqrt{1+\delta_{1,i}^2}$
0	$\delta_{0,i}$	1
-1	$\tan^{-1}\delta_{-1,i}$	$\sqrt{1-\delta_{-1,i}^2}$

Table 4.2 The CORDIC functions for different operation modes.

Mode	m	CORDIC function
Rotation $z_n \to 0$	1	$\begin{cases} x_n = x_0 \cos z_0 - y_0 \sin z_0 \\ y_n = y_0 \cos z_0 + x_0 \sin z_0 \end{cases}$
	0	$\begin{cases} x_n = x_0 \\ y_n = y_0 + z_0 x_0 \end{cases}$
	-1	$\begin{cases} x_n = x_0 \cosh z_0 - y_0 \sinh z_0 \\ y_n = y_0 \cosh z_0 + x_0 \sinh z_0 \end{cases}$
Vectoring $y_n \to 0$	1	$\begin{cases} x_n = \sqrt{x_0^2 + y_0^2} \\ z_n = z_0 + \tan^{-1}(y_0/x_0) \end{cases}$
	0	$\begin{cases} x_n = x_0 \\ z_n = z_0 + y_0/x_0 \end{cases}$
	-1	$\begin{cases} x_n = \sqrt{x_0^2 + y_0^2} \\ z_n = z_0 + \tanh^{-1}(y_0/x_0) \end{cases}$

4.2 Complex division by CORDIC

As mentioned in section 4.1, a complex valued division can be computed by transforming the complex numbers from the rectangular coordinate system into the polar coordinate system, performing the operations indicated by equation (2.7), and then retransforming the result back to the rectangular coordinate system. To accomplish these operations, CORDIC rotators operating in different modes can be used. A computational scheme of a complex valued division by CORDIC is shown in figure 4.5.

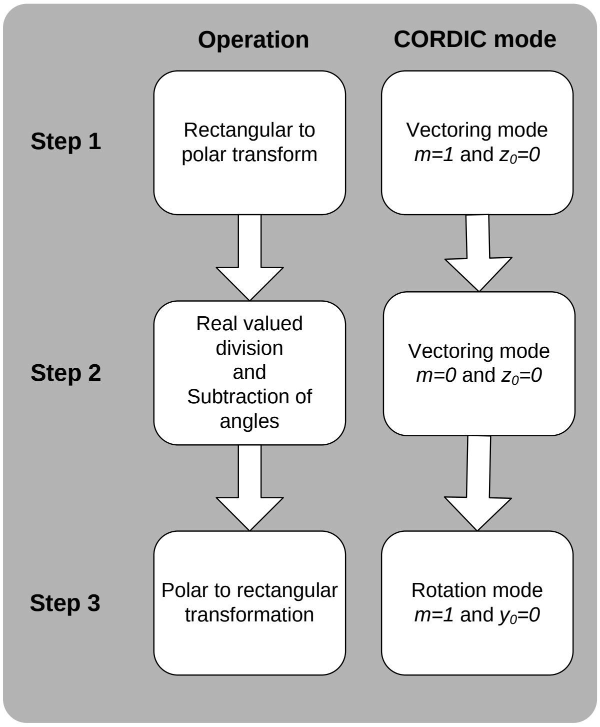

Operation
CORDIC mode
Step 1
Rectangular to polar transform
Vectoring mode $m=1$ and $z_0=0$
Step 2
Real valued division and Subtraction of angles
Vectoring mode $m=0$ and $z_0=0$
Step 3
Polar to rectangular transformation
Rotation mode $m=1$ and $y_0=0$

Figure 4.5 Computation scheme of complex valued division by CORDIC.

Step 1: Rectangular to polar transformation

Transforming from Cartesian or rectangular coordinates to polar coordinates with a CORDIC consists of finding the magnitude, equation (2.1), and the phase angle, equation (2.2), of a given input vector. As shown by the vectoring mode in table 4.2, both functions are provided simultaneously when $m=1$ and $z_0=0$. The magnitude of the result will be scaled by the CORDIC rotator gain. Unwanted gain can be corrected by multiplying the resulting magnitude by the reciprocal of the gain constant.

Step 2: Division and subtraction

Using the CORDIC in vectoring mode with $m=0$ and $z_0=0$, a division between two input values can be performed. The rotations in the linear coordinate system ($m=0$) have a unity gain that eliminates the need for any correction of the scaling constant.

The subtraction between the two angles can be performed by using a separate CORDIC processor, an available subtractor in the CORDIC or an external subtractor unit.

Step 3: Polar to rectangular transformation

Transformation from polar to rectangular coordinates is done using the equations in (2.5). The transformation is accomplished by using the CORDIC in the rotation mode, with the polar magnitude and the polar phase as inputs, and $m=1$ and $y_0=0$. As in the case with transforming from rectangular to polar coordinates, the magnitude of the result will be scaled by the CORDIC rotator gain. Unwanted gain can either be corrected by multiplying the resulting magnitude by the reciprocal of the gain constant, or by multiplying the polar magnitude with the reciprocal of the rotator gain before inputting it into the CORDIC rotator.

4.3 Architecture for a CORDIC based complex valued division

4.3.1 CORDIC element architectures

A standard single iteration CORDIC stage is shown in figure 4.6. The architecture is a straightforward derivative of the equations discussed in table 4.2.

The CORDIC element architecture in figure 4.6 perform one iteration per clock cycle and consists of three b-bit adders/subtractors, two shifters, a ROM for the angle constants, and control circuitry. In the initial step, operand values are loaded into the registers. In the next n steps, the values from the registers are passed through the adders/subtractors and shifters, and the results are fed back to the registers. The shifters are modified during each iteration to cause the desired shift and the ROM address is incremented so that the appropriate angle value is fed into the adder/subtractor of z. The iteration process is controlled by the control circuitry, which takes the signs of the three input values, mode m, and the iteration level as inputs.

Care must be taken when implementing this architecture in an FPGA. Problems with the mapping of the architecture due to its bit-parallel design can arise. The bit-parallel design requires high fan-in bit-parallel variable shifters, which in FPGA (and in ASIC) implementations could result in several layers of logic being needed. This will result in a slow and resource consuming design.

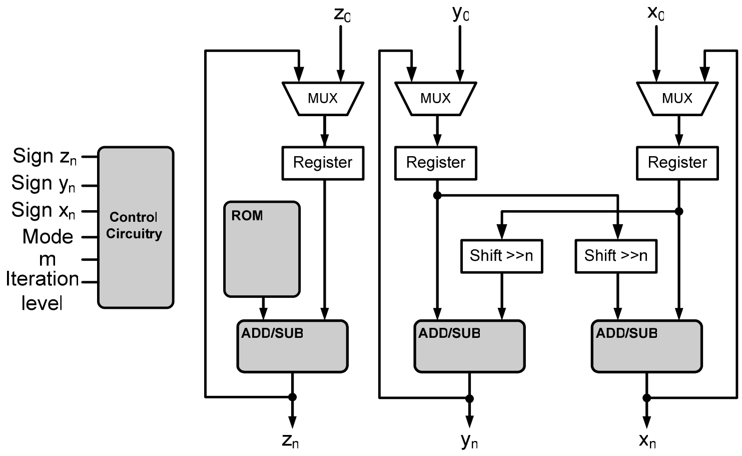


Figure 4.6 Hardware Hardware architecture of a standard single iteration CORDIC element.

A more compact architecture is achievable by using bit-serial arithmetic. The bit-serial CORDIC element architecture shown in figure 4.7 consists of three bit-serial adder/subtractors, three shift registers, and a serial ROM [33],[34].

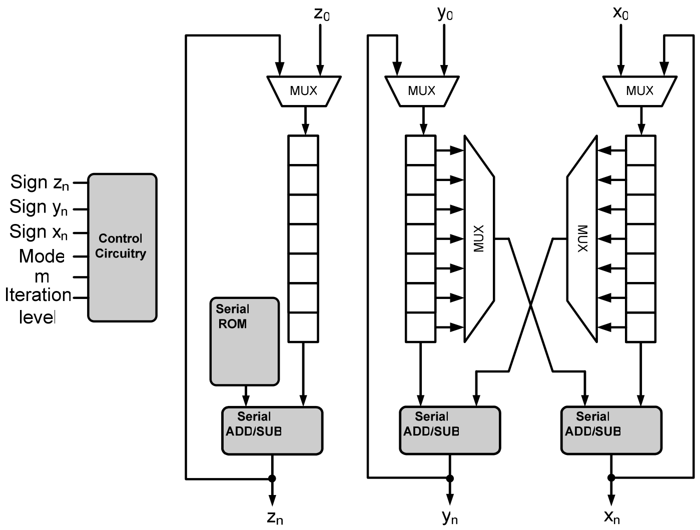


Figure 4.7 Hardware architecture of a bit-serial single iteration CORDIC element.

When initialized, the load multiplexers are open for *w* cycles to fill the registers. Once loaded, the data is fed through the adder/subtractor and returned to the shift registers, requiring *w* cycles to complete the operation. Due to the minimal interconnections and the simple logic of the bit-serial architecture, it can be run at very high clock rate on the FPGA. The architecture requires *w* cycles per iteration, where *w* is the precision of the adder/subtractor. This architecture is considerably slower than the iterative architecture, when operated at the same clock frequency. However, a higher clock rate can often make up for the large amount of cycles needed to complete each rotation [33],[34]. If the system in which the bit-serial CORDIC element is going to be used is not bit-serial in its nature, then conversion logic must be used at the inputs and the outputs.

4.3.2 CORDIC processor architectures

A CORDIC processor (CP) architecture consists of one or more CORDIC elements. There are mainly three types of architectural approaches [35]:

1. The iterative or sequential approach. This architecture is unfolded in time and often consists of one CORDIC element connected recursively as shown in figure 4.8.

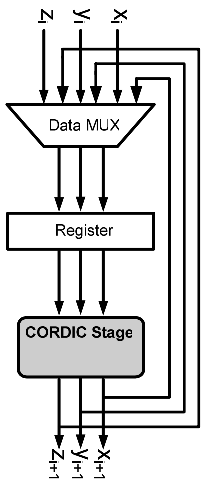


Figure 4.8 Iterative CORDIC implementation.

2. The cascaded or parallel approach. This architecture is unfolded in space and is implemented as a cascade of *n* CORDIC elements as shown in figure 4.9.
3. A combination of 1 and 2. This architecture is often referred to as the cascaded fusion approach, and is based on a sequential structure where the logic for several successive iterations is cascaded and executed within one clock cycle [33],[34].

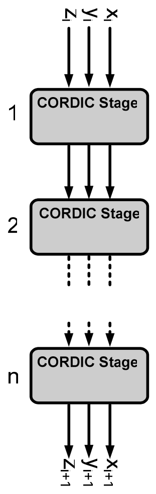


Figure 4.9 Cascade CORDIC implementation.

Both the CORDIC element architectures presented in section 4.3.1 can be used in the three processor architectures above. If the cascade architecture is used, the CORDIC elements are unrolled so that each of the *n* elements always performs the same iteration. This will simplify both architectures considerably and decrease the computation time, but it will also consume lots of resources in the FPGA. Figure 4.10 shows an example of a cascaded CORDIC processor with *n* CORDIC elements.

Each element always performs the same rotation, and so the ROMs can be replaced with constants and the shifters can be set to shift the same number of steps every time.

The array of interconnected adders/subtractors may need to be pipelined to keep down the propagation delay through the structure. Choosing which of the CORDIC processor architectures to use in a given application comes down, as always, to a trade-off between speed and area.

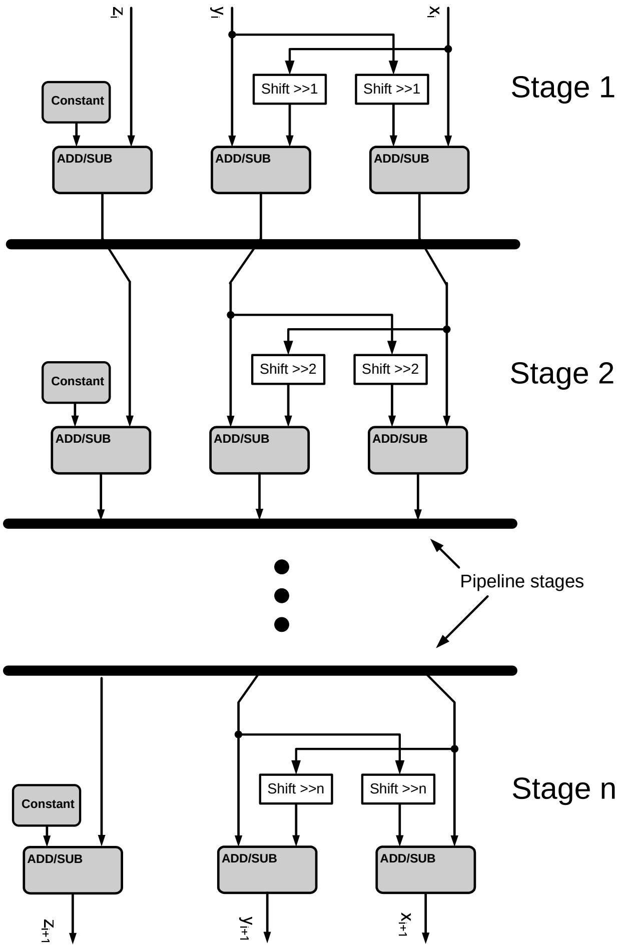
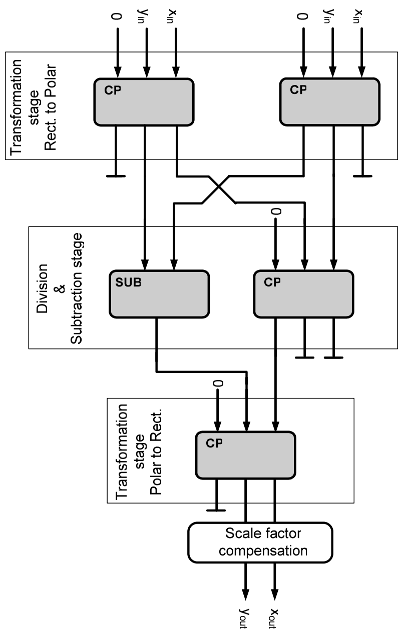
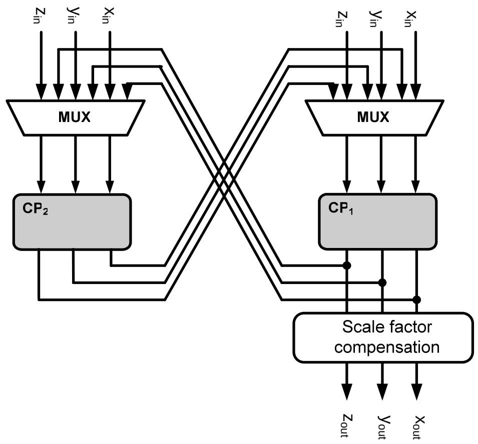


Figure 4.10 A CORDIC processor consisting of *n* cascaded CORDIC elements.

4.3.3 CORDIC based complex valued division architecture

The computational scheme in figure 4.5 can be directly translated into a CORDIC based hardware architecture. The resulting CORDIC architecture is shown in figure 4.11 (a). The architecture consists of four CORDIC processors (CP), one subtractor, and a scale factor compensation unit performing the complex valued division. This architecture will consume lots of resources, especially if the CORDIC processors are implemented using a number of cascaded iterative CORDIC elements. To minimize the resource consumption the three steps of the architecture were mapped onto a single step


Transformation stage Rect. to Polar
Division & Subtraction stage
Transformation stage Polar to Rect.
CP
SUB
Scale factor compensation
0
y_{in}
x_{in}
y_{out}
x_{out}



(a) (b)

Figure 4.11 (a) Complex valued division using three CORDIC processors in cascade. (b) A compact iterativede versionof the complex valued division.

consisting of a pair of CORDIC processors. The new and more compact architecture is shown in figure 4.11 (b) which will discussed in the following section.

The magnitude of the result will during transformation from rectangular to polar coordinates be scaled by the CORDIC rotator gain, K_n. The same will happen during the transformation from polar to rectangular coordinates. Linear operations have a CORDIC rotator gain equal to unity. This means that the scale factor must compensate for $2K_n$ gain by multiplying the result with the factor $1/2K_n$.

4.3.4 Scheduling of the complex valued division architecture

A scheduling diagram of the architecture in 4.11 (b) is shown in figure 4.12. Both CORDIC processors CP_1 and CP_2 are realized with eight cascaded iterative CORDIC elements requiring $b=8$ cycles to produce a result.

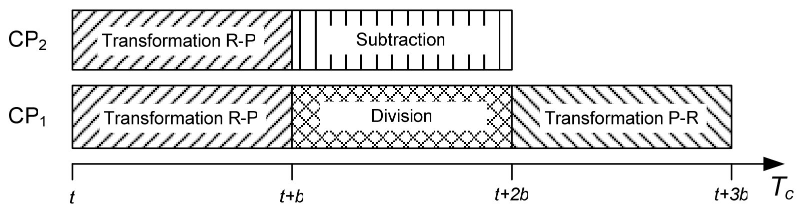


Figure 4.12 Scheduling diagram of the CORDIC based complex valued divider architecture where *n* is the number of cascaded elements or iterations needed to produce a result.

At time instance *t* and *t+b* the transformation of the two complex numbers from the rectangular coordinate system to the polar coordinate system is performed. Both CORDIC processors are set to operate in vector mode with $m=1$ and $z_0=0$. The resulting angles are then subtracted from each other in time instance *t+b* to *t+2b* where CP_2 is set in rotation mode with $m=0$ and $x_0=1$. In the same time instance CP_1 performs the division between the two absolute values. CP_1 is operating in vectoring mode with $m=0$ and $z_0=0$. The resulting polar number is then transformed by CP_1 working in rotation mode with $m=1$ and $y_0=0$ in time instance *t+2b* to *t+3b*. The whole process is performed in 3*b* cycles, which in this case translate into 24 cycles since 8 CORDIC elements are used in both CPs.

Instead of doing a division and a subtraction in time instance *t+b* to *t+2b*, the two CPs can be set to do a multiplication (CP_1 in rotation mode $m=0$ and $z_0=1$) and an addition (CP_2 in rotation mode $m=0$ and $y_0=0$). In this way the architecture is also able to do a complex valued multiplication.

4.4 FPGA implementation of a CORDIC based complex valued division

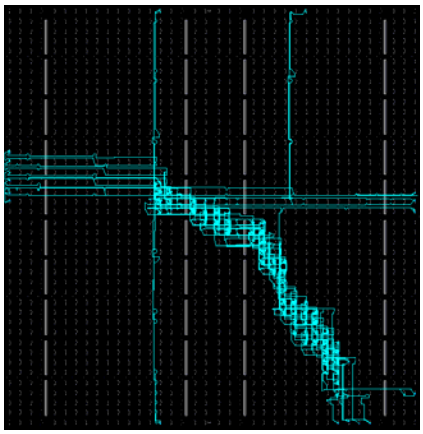
The CORDIC based complex valued division architecture has been implemented in a Xilinx Virtex II XC2V1000 FPGA with a speed grade of 4. Fixed-point representation was used and the bit length of the input operands was 8 bits. Table 4.3 shows a summary of the architectural and implementation parameters. The CORDIC design has a startup latency of 24 cycles and it can accommodate 1 simultaneous complex valued division. In continuous operation the parallel architecture produces one complex valued output every 24 cycles. The unrolled architecture consumes 4% of the FPGAs resources and the maximum clocking frequency is 168 MHz.

Table 4.3 Sumary of the architectural and implementation parameters.

	CORDIC divider
Startup Latency (cycles)	24 (*3b**)
Throughput rate (output/cycle)	1/24 (1/*3b**)
Max simultaneous operations	1
Number of Slices	243 (4%)
Number of Slice Flip Flops	-
Number of 4 input LUTs	454 (4%)
Number of BRAMs	-
Maximum frequency (MHz)	168

* Where *b* is the number of bits in the operand.

Figure 4.13 (a) shows the routed floorplan of the FPGA implementation of the complex valued division CORDIC architecture. In 4.13 (b) the cells occupied by the design are shown. It is easy to see the chain of cascaded CORDIC elements bending downwards into the right corner on the floorplan. Unlike the placement and routing of the two other complex valued division designs in chapter 3, the logic is placed in a chain and not clustered. This means that the delay time due to long wires is avoided and that a higher clock rate can be achieved.


(a)

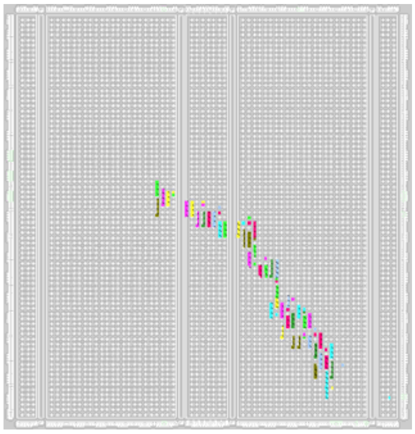

(b)

Figure 4.13 (a) Routed floorplan of the FPGA implementation of the CORDIC based architecture. (b) Placement of the arithmetic building blocks.

Chapter 5

Comparison of complex valued division implementations

In previous chapters three different implementations of complex valued division have been presented. To put the different FPGA designs into perspective, a summary of the design parameters are presented in table 5.1.

Table 5.1 Comparison of the three complex valued division architectures.

	Eq. (14) Parallel	**Eq. (14) Multiplexed**	**CORDIC divider**
Startup Latency (cycles)	6	8	24 (*3b**)
Throughput rate (output/cycle)	1/1	1/4	1/24 (1/*3b**)
Max simultaneous operations	6	3	1
Number of Slices	436 (8%)	146 (2%)	243 (4%)
Number of Slice Flip Flops	820 (8%)	276 (2%)	-
Number of 4 input LUTs	799 (7%)	270 (2%)	454 (4%)
Number of BRAMs	3 (7%)	1 (2%)	-
Maximum frequency (MHz)	147	156	168

* Where *b* is the number of bits in the operand.

The three designs will be judged from two perspectives: throughput rate and resource consumption. These two parameters are important in most FPGA hardware implementations including the ones presented in part II of the thesis.

5.1 Throughput

The parallel architecture is clearly the winner in terms of throughput. In continuous operation it produces 1 complex valued division every cycle with a start-up latency of 6 cycles. The multiplexed design also has good performance, producing one complex valued every fourth cycle. The throughput for both the parallel and the multiplexed design is independent of the wordlength. However, the throughput of the CORDIC based complex valued division is dependant on the bit-length of the operands. In the case presented in previous sections the CORDIC based complex valued division requires $3b$ cycles to produce a result, where b is the bit-length of the operator. For 8 bit operands (b=8) the CORDIC has six times lower performance than the multiplexed design and 24 times lower than the parallel design.

5.2 Resources

All three designs where implemented using the same wordlength and types of arithmetic building blocks and can therefore be compared to each other in terms of resource consumption in an FPGA. The parallel design clearly consumes the largest amount of resources of the three designs. It uses nearly three times as many resources as the multiplexed design and 1.8 times the amount of resources that the CORDIC based divider uses. In this comparison the multiplexed design consumes the least amount of resources.

It may be argued that the resource comparison is not fair since only one design each, with a short wordlength for common practical use, was implemented. For instance when the wordlength grows, the LUTs in the parallel and multiplexed designs will grow rapidly compared to the CORDIC based design. To investigate how the resource consumption changes with the wordlength, several designs where constructed and compared. The result is shown in figure 5.1.

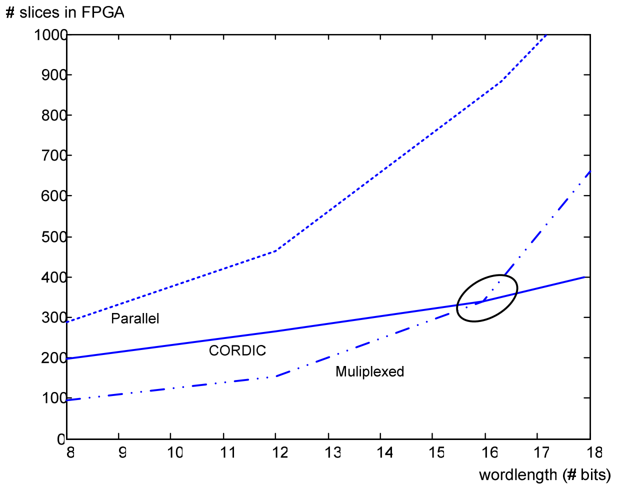


Figure 5.1 Consumed resources vs. wordlength of the implementations.

The figure clearly shows that the multiplexed design consumes the least amount of resources up to the breakeven point, marked in the diagram by an ellipse, around a 16 bit wordlength. If the input operands exceed 16 bits each for the real and the imaginary part, and resources is of great importance, the CORDIC based approached should be chosen.

5.3 Summary

In this chapter three different implementation of complex valued division have been presented. The choice of which of the three implementations to use depend on the requirements of the target application and available hardware resources. If the complex valued division is going to be used in a system requiring high data rates, the parallel design based on Smith's formula scheduled as in figure 3.4 should be used. This architecture consumes a lot of hardware resources, especially when the wordlength is long, but produces a complex division every cycle.

If hardware resources are scarce and the operand wordlength exceeds 16 bits, the CORDIC base complex valued division should be used. However, the architecture requires many cycles to produce an output value. Many times, the low production rate can be compensated by a higher clocking rate of the architecture.

If both high throughput rate and resource consumption is of importance, and the wordlength of the operands is not greater than 16 bits, the multiplexed design should be chosen for implementation. It has relatively small resource consumption and a good throughput rate especially compared to the CORDIC architecture.

Part II

Architectures for Matrix Computations

Chapter 1

Introduction

Matrix computations are, as discussed in the introduction of this thesis, fundamental operations used in a wide variety of algorithms in many different fields. This part of the thesis is devoted to matrix operations which are used in enumerable applications: inversion of a triangular matrix, QR-decomposition, matrix inversion, and singular value decomposition (SVD). The matrix inversion is formed by combining the QR-decomposition together with the triangular matrix inversion. These four matrix operations can be viewed as basic building blocks in many modern algorithms.

In chapter 2, an investigation of the inversion of real and complex valued triangular matrices. Three different architectures implementing the inversion are derived handling both real and complex input matrices. A new mapping method which facilitates the scheduling of the architectures is presented. The chapter ends with a look at several different hardware implementations of the architectures that has been derived.

Chapter 3 concerns the QR-decomposition of a matrix. The chapter starts of by investigating an algorithm that is suitable for hardware implementation. The derived algorithm is then mapped onto the same architectures derived in chapter 2 for the inversion of triangular matrices. The scheduling from chapter 2 is also used with a slight modification. The chapter ends with a look at two complex valued QR-decomposition architectures implemented in hardware.

In chapter 4 we take a closer look at the real and complex valued matrix inversion. The matrix inversion is solved by combining the QR-decomposition and the triangular array architecture architectures. This chapter ends with a look at two implementations capable of doing complex valued matrix inversion.

In chapter 5 we look at singular value decomposition of complex valued matrices. A commonly used architecture for implementing singular values decomposition is scrutinized. A new more area conservative and scalable architecture is proposed. The chapter ends with a look at an implementation of the singular values decomposition for complex matrices. Finally a summary is made and conclusions are drawn.

Chapter 2

Inversion of a triangular matrix

Inversion of triangular matrixes is a key operation in a number of important decomposition techniques used in numerical and signal processing algorithms [15],[36],[37].

It is therefore vital to derive efficient hardware architecture for inverting a triangular matrix. An alternative to conventional mapping strategy is proposed along with three different architectures offering different benefits as computation speed, low resource consumption, and ease of scalability. Hardware implementations of the architectures, for complex and real valued matrices, are derived.

2.1 Matrix definitions

A vector $\boldsymbol{x}$ of dimension n is an array of n scalars of the form

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \quad \text{or} \quad x^T = (x_1, x_2, \cdots, x_n) \qquad x \in \mathbb{R}, \mathbb{C} \tag{2.1}$$

The scalar x_i is called the component of x. The set of n-vectors with real components are written $\mathbb{R}^n$ and with real or complex components are written $\mathbb{C}^n$. The left vector in (2.1) is commonly called a column vector (column matrix) and the right vector in (2.1) is commonly called a row vector (row matrix).

An $m \times n$ matrix A is an array of scalars written on the form

$$A = (a_{ij}) = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \qquad a_{ij} \in \mathbb{R}, \mathbb{C} \tag{2.2}$$

where the scalar a_{ij} is called the entry or element of the matrix $\boldsymbol{A}$. The set of $m \times n$ matrices with real elements is written $\mathbb{R}^{m \times n}$ and with real or complex elements is written $\mathbb{C}^{m\times n}$. The indices i and j of the scalar a_{ij} are called respectively the row index and the column index. An $n\times n$ matrix is called a square matrix and a $m\times n$ matrix is called a rectangular matrix.

A triangular matrix is a square matrix, $n\times n$, where the entries below or above the main diagonal are zero. A matrix with real or complex valued entries above and including the diagonal as shown to the left in equation (2.3) is called an upper triangular matrix or right triangular matrix. A matrix with entries below the diagonal as shown to the right in equation (2.3) is called a lower triangular matrix or left triangular matrix. The letter ***L*** is commonly used for a lower triangular matrix, while the letters ***U*** or ***R*** are used for an upper triangular matrix.

$$R = \begin{pmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,n} \\ 0 & r_{2,2} & \cdots & r_{2,n} \\ 0 & 0 & \ddots & \vdots \\ 0 & 0 & 0 & r_{n,n} \end{pmatrix} \qquad L = \begin{pmatrix} l_{1,1} & 0 & 0 & 0 \\ l_{2,1} & l_{2,2} & 0 & 0 \\ \vdots & \ddots & \ddots & 0 \\ l_{n,1} & l_{n,2} & \cdots & l_{n,n} \end{pmatrix} \tag{2.3}$$

There are other types of matrices that also belong to the family of triangular matrices. A triangular matrix with zeros on the main diagonal is termed a strictly upper or lower triangular matrix. A matrix which is both upper and lower triangular is diagonal. If the main diagonal consists of only ones, the matrix is termed unit upper (lower) or normed upper (lower) triangular matrix. The identity matrix, ***I***, is the only matrix which is both normed upper and lower triangular. The matrix is shown in equation (2.4).

$$I = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.4}$$

2.2 Triangular matrix inversion algorithm

The algorithm for inversion of a triangular matrix is derived from the relations shown in equation (2.5). Let ***R*** be a triangular matrix and ***I*** be the identity matrix. Then

$$\left(R^T\right)^{-1} = \left(R^{-1}\right)^T \;\rightarrow\; R^T\left(R^T\right)^{-1} = I \;\rightarrow\; R^T W^T = I \tag{2.5}$$

where ***W*** is the inverse of the upper triangular array. The relations in (2.5) can be translated into a recurrence algorithm that can be used for computing the inverse of an upper or lower triangular matrix [38]. The pseudo code of the recurrence algorithm is shown in (2.6).

```
if $i > j$
    $W_{ij} = 0$
else if $i = j$
    $W_{ij} = r_{jj}^{-1}$                                        (2.6)
else if $i < j$
    $W_{ij} = -\left(\sum_{m=1}^{j-1} w_{im} r_{mj}\right) r_{jj}^{-1}$
end
```

The two first conditions in the recurrence algorithm are trivial. All elements under the diagonal in the inverted matrix will be zero and the new diagonal elements, w_{jj}, are computed by inverting the diagonal elements, r_{jj}. All other elements are computed using a back-substitution scheme which is a well known method for solving equation systems [14]. Fortunately, the recurrence algorithm has a lot of inherent parallelism. That means that the summation operations in the recurrence algorithm can be performed in parallel, which speeds up the computation and makes it suitable for hardware implementation.

Since the inversion of the triangular matrix is solved by back-substitution, it has been proven to be a numerical stable process with a low relative error even for ill-conditioned matrixes [14]. However, when implementing in hardware using arithmetic building blocks with finite precision, it is important to keep track of rounding errors during intermediate computations. If this is not considered during implementation, errors can be induced and decreased the precision of the result.

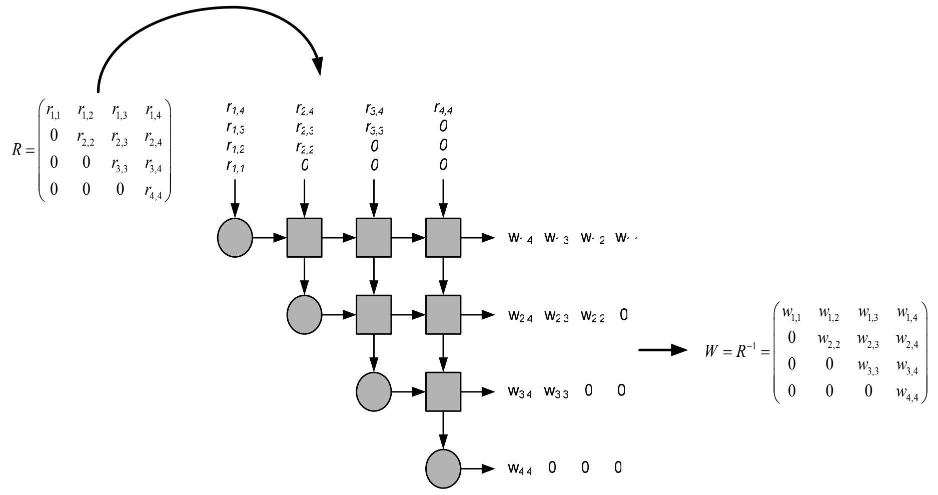
The algorithm for inverting a triangular matrix presented in equation (2.6) is also applicable to triangular matrices consisting of complex numbers. All real operations in the recurrence algorithm are then replaced by the operations handling complex numbers.

2.3 Architectures for triangular matrix inversion

Architectural designs proposed in literature for systems involving triangular matrix inversion are often based on array structures with parallel communicating processor elements [16]. These structures are also known as systolic or wave front architectures, and are characterized by their simplicity, uniformity, and local communication between processor elements. As discussed in the previous section the inherent parallelism that exists in matrix operation algorithms makes them easy to map onto such array architectures. A triangular shaped array architecture is the most used architecture in literature, but there are also other possible architectures that could be used, including linear array architecture and single processing element architecture, which will be discussed in the following sections.

2.3.1 Triangular array architecture

Two versions of the triangular array architecture usually appear in literature. The first one is in this thesis called triangular wave array architecture while the second one is called triangular pipelined array architecture [23]. Figure 2.1 shows a triangular wave array architecture with 10 communicating processor elements for inverting a 4-by-4 triangular matrix. There are two types of processing elements, boundary processing elements and internal processing elements. The functionality of the two types of processing elements will be discussed later in this chapter.



$R = \begin{pmatrix} r_{1,1} & r_{1,2} & r_{1,3} & r_{1,4} \\ 0 & r_{2,2} & r_{2,3} & r_{2,4} \\ 0 & 0 & r_{3,3} & r_{3,4} \\ 0 & 0 & 0 & r_{4,4} \end{pmatrix}$
$r_{1,4}$ $r_{1,3}$ $r_{1,2}$ $r_{1,1}$
$r_{2,4}$ $r_{2,3}$ $r_{2,2}$ 0
$r_{3,4}$ $r_{3,3}$ 0 0
$r_{4,4}$ 0 0 0
w·4 w·3 w·2 w··
w24 w23 w22 0
w34 w33 0 0
w44 0 0 0
$W = R^{-1} = \begin{pmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} \\ 0 & w_{2,2} & w_{2,3} & w_{2,4} \\ 0 & 0 & w_{3,3} & w_{3,4} \\ 0 & 0 & 0 & w_{4,4} \end{pmatrix}$

Figure 2.1 Triangular wave array architecture for inverting a 4-by-4 triangular matrix.

The matrix elements, $r_{1,1}$ to $r_{4,4}$, of the triangular matrix $\boldsymbol{R}$, is fed into the top of the array architecture and propagate through the architecture like a wave. Every clock cycle, 4 data samples, representing one row of the matrix, are processed through the architecture, thus producing one output row in the inverted matrix. The computation is triggered by the arrival of data on the inputs of the processing elements. An inversion of an n-by-n triangular matrix can thereby be completed in n cycles, and in this particular case a 4-by-4 triangular matrix is inverted in 4 cycles.

One problem with the wave architecture shown in figure 2.1 is that the wave must propagate through the whole architecture before the next wave can be initiated. When large triangular matrices are going to be inverted there will be long delay times between waves. This will affect the maximum achievable clock speed of the architecture.

One way to solve this problem in large arrays is to process several waves of data at the same time through the array architecture as shown in figure 2.2. The first wave, W_1, starts to propagate through the architecture at cycle time T_1. When the next wave of data starts to propagate through the architecture at time instance T_2 the first wave has reached halfway through the architecture. In this way idle processing elements are utilized in a more efficient way, cutting the delay time between waves in half. An inversion of an *n*-by-*n* triangular matrix can in this architecture be completed in *n/x* cycles, where *x* is the number of possible waves processed at the same time in the architecture. The timing of the waves propagating through the architecture is very important. If a collision would occur the output values will be unusable. Different handshaking strategies can be employed to reduce the risk [39].

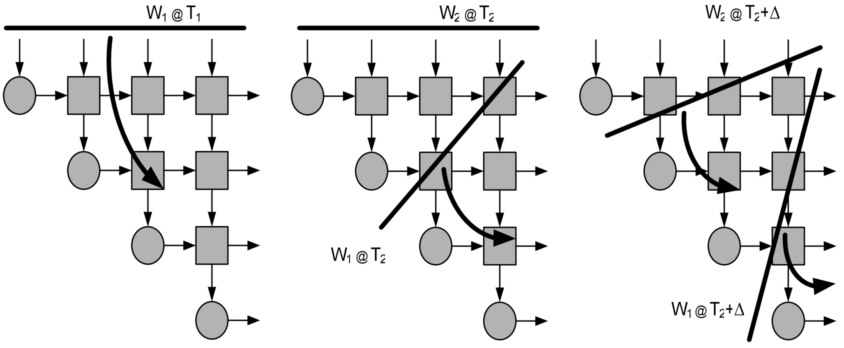


Figure 2.2 Several computational wave fronts passing through a triangular wave array architecture at different time instances.

Another way of reducing the delay time is to pipeline the architecture by placing registers between each processing element as shown in figure 2.3. Pipelining could in this way be exploited to either increase the clock speed or to reduce the power consumption at the same speed. The upper limit of the clock speed is now limited only by the propagation time through an individual processing element instead of through the whole architecture. Figure 2.3 shows a triangular pipelined array architecture with 10 communicating processor elements, separated by registers, for inverting a 4-by-4 triangular matrix.

In the triangular pipelined array architecture the matrix elements are fed into the top of the architecture in a skew manner to compensate for the delay caused by the registers. If this is not compensated for, the matrix elements will be fed into the architecture at the wrong time instance, thus corrupting the computation. The matrix data will propagate through the architecture in the direction of the arrows. Suppose that a processing element can be executed in a single cycle (a unity cycle); an *n*-by-*n* triangular matrix can then be processed in $2(n+1)$ cycles. An inversion of a 4-by-4 matrix will be completed in 10 cycles.

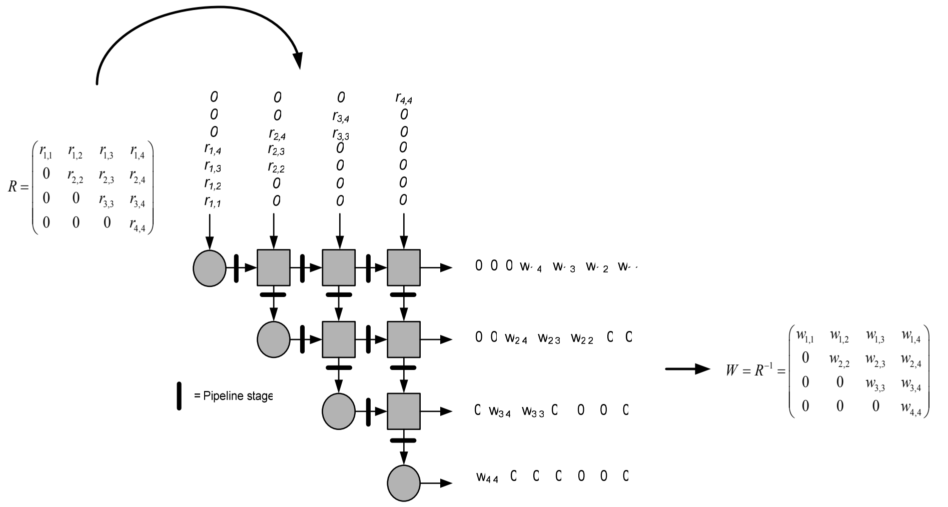

$R = \begin{pmatrix} r_{1,1} & r_{1,2} & r_{1,3} & r_{1,4} \\ 0 & r_{2,2} & r_{2,3} & r_{2,4} \\ 0 & 0 & r_{3,3} & r_{3,4} \\ 0 & 0 & 0 & r_{4,4} \end{pmatrix}$
= Pipeline stage
$W = R^{-1} = \begin{pmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} \\ 0 & w_{2,2} & w_{2,3} & w_{2,4} \\ 0 & 0 & w_{3,3} & w_{3,4} \\ 0 & 0 & 0 & w_{4,4} \end{pmatrix}$

Figure 2.3 Triangular pipelined array architecture for inverting a 4-by-4 triangular matrix ***R***.

The two triangular array architectures presented in this section have some drawbacks that make them unsuitable for hardware implementation. These drawbacks include:

- Long delay time (critical path) through array structure of figure 2.1. The propagation delay of a wave (or multiple waves) traveling throughout the architecture grows with the size (number of inputs) of the array. This will affect the maximum achievable clock frequency. However, as shown in the pipelined array architecture of figure 2.1, this drawback can easily be solved by pipelining.
- The number of processing elements will increase rapidly with the number of inputs. To be able to process *n* input samples, *n(n+1)/2* processing elements are required. The increasing number of processing elements will result in a large consumption of hardware resources (chip area in ASIC and slices in FPGA).
- Lack of easy scalability. In order to add one more input a new design must be derived.

As discussed in the introduction of the thesis there is a growing need for easily scalable, high throughput rate, and area conservative architectures in modern system-on-chip (SoC) design. The two triangular array architectures are far from optimal in this respect. Therefore, a compact and scalable design with good throughput rate is highly sought for. Presented here is an alternative architecture, a linear structure with one processing element for each input.

2.3.2 Linear array architecture

Several authors have presented mapping methods of the triangular array architecture in [23],[40]. In [41] Rader suggested spatial folding, which produces a square array architecture which according to Rader is more convenient for hardware implementation [41]. Rader’s mapping applied to the triangular array architecture shown in figure 2.4 (1) to (3). The first folding cut is made in the middle of the array architecture. The bottom part is folded in such way that it produces two rows of processing elements. The next folding cut is placed in the middle of the top part produced by the first folding cut. The resulting architecture is a square architecture with local communication between processors. However, this mapping method does not reduce the number of processing elements needed and does not make it easier to scale the architecture.

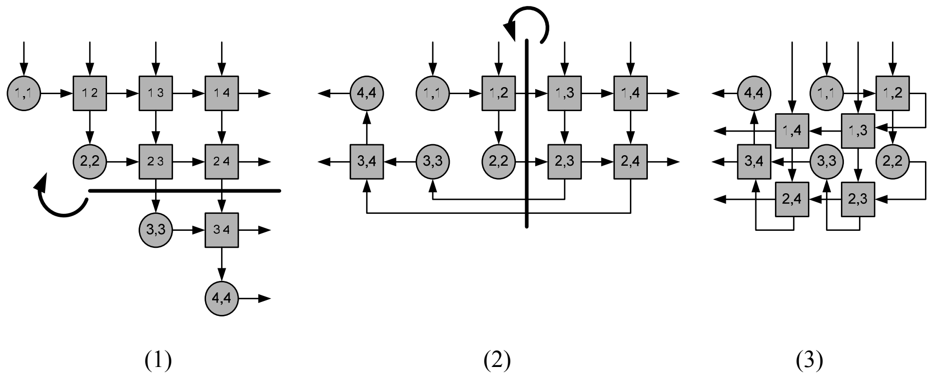


Figure 2.4 Rader's folding of a triangular array architecture onto a square array architecture.

In Walke [40] presented another mapping method where the architecture is folded in several steps into a linear array architecture. This mapping method is also spatial in its nature. Unlike in Rader’s method, Walke divides the architecture diagonally instead of horizontally as shown in figure 2.5 (1) to (4). The lower part is wrapped around and placed on top of the upper part forming a square. A new folding cut is done down the middle of the square of processing elements. Three rows of processor elements are formed and mapped onto a single linear architecture consisting of three processing elements. This mapping method reduces the number of needed processing elements significantly but it also comes with a drawback. The scheduling of the linear array architecture must be performed in the order indicated by the dotted lines in the figure. This complex scheduling requires a sophisticated control unit, pre-storage of the matrix and several registers storing intermediate computations. Moreover, the architecture is not easy to scale without modifying the schedule of the architecture. This will result in a redesign of the control circuit.

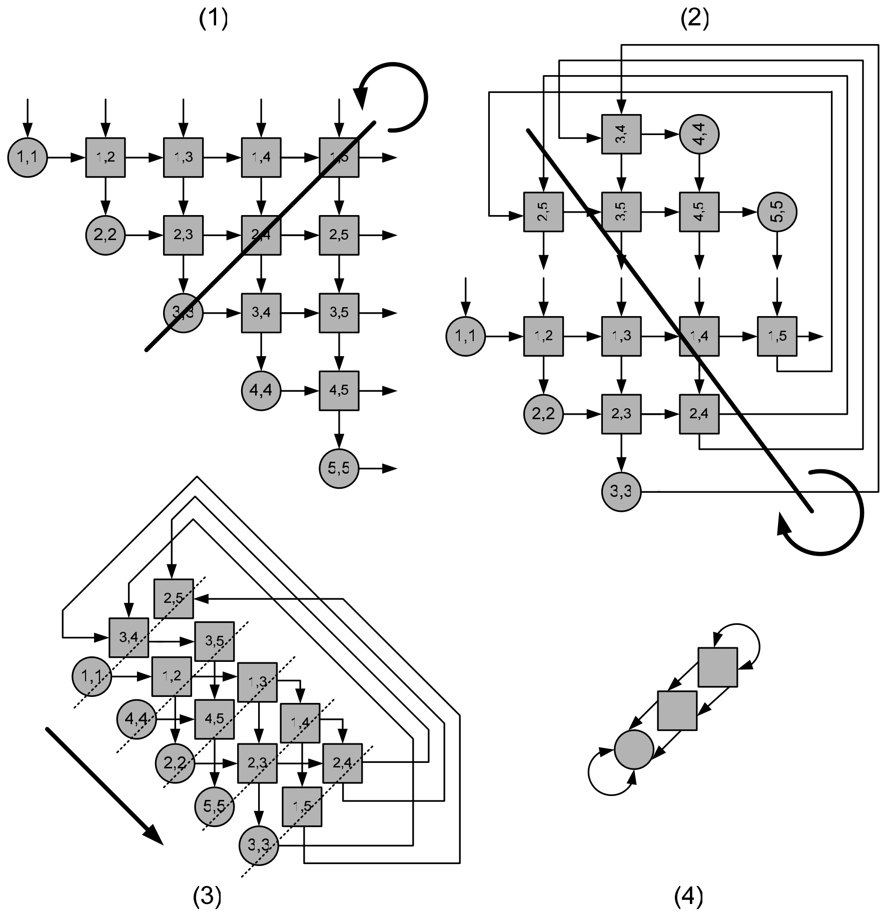


Figure 2.5 Walke's mapping of a triangular array architecture onto a linear array architecture.

The mapping method suggested in this thesis is based on mapping with respect to the data flow in the architecture. An individual input data will travel through the array architecture along a certain path interacting with other data along the way. The four data paths extracted from the triangular array architecture is shown in figure 2.6. The extracted data paths (1-4) are placed along each other as shown in the middle of figure 2.6. The behavior of the data elements are then mapped onto each other top-down, as shown by the arrows, resulting in a linear array architecture with one processing element for each input as shown in the right of figure 2.6. Unlike the processing elements in Rader and Walke, the one in this architecture are capable of performing both the function of a round processing element and a square processing element without any extra cost in hardware (section 2.6). The data flow in the linear array architecture mimics the data flow of the original triangular array architecture but eliminates complex flow control circuitry and extra registers for storing intermediate computations.

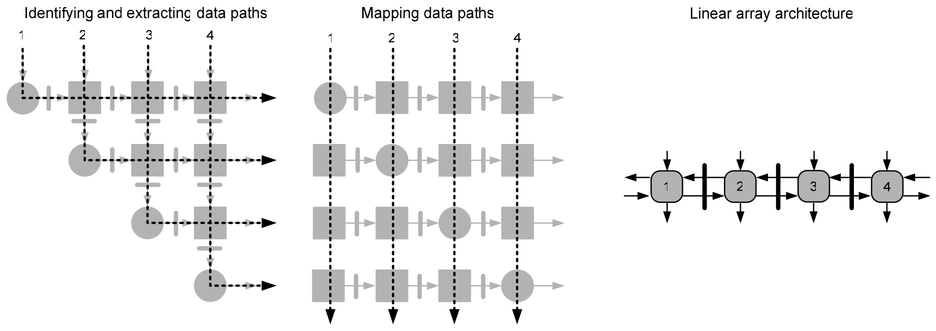

Identifying and extracting data paths
1
2
3
4
Mapping data paths
1
2
3
4
Linear array architecture
1
2
3
4

Figure 2.6 Mapping of a triangular array architecture onto a linear array architecture.

The linear array architecture in figure 2.6 reduces the required number of processing elements considerably. Instead of *n(n+1)/2* processing elements to process *n* input samples, only *n* is needed.

Unlike Rader's and Walke's architectures [40], the linear array architecture is very easy to scale if more inputs are needed to process larger matrices. As shown in figure 2.7, scaling is done by adding the number of processing elements (one for each new input) required for a specific application at the end of the linear structure. There is no upper limit to how many processing elements can be added to the structure. If large matrices were to be inverted, several chips could be connected together.

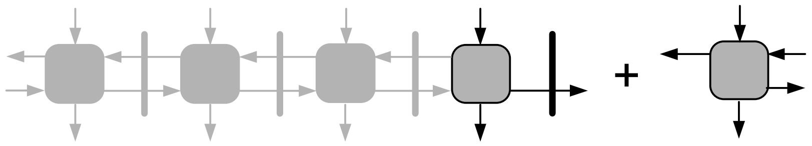


Figure 2.7 Scaling of the linear array architecture is easily done by adding a processing element at either end of the array.

2.3.3 Single element architecture

Although the size of the architecture has been significantly reduced in the linear array compared to the triangular array architecture, it still consumes lots of hardware when implementing very large matrix sizes. This can be troublesome in cases where large systems-on-chip are to be constructed. Therefore, smaller architecture can be preferable in some cases at the expense of reduced throughput.

A natural step in the process of reducing the size of the architecture is to map the linear array architecture onto one single processing element (SPE) as shown in figure 2.8.

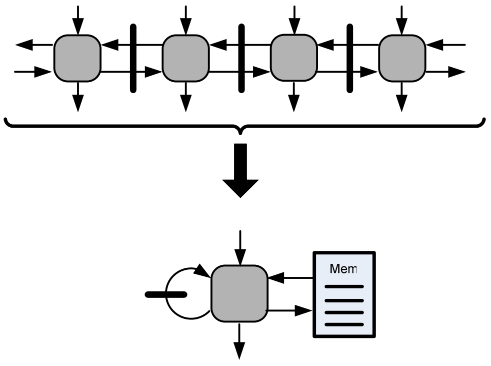


Figure 2.8 Mapping of a linear array architecture onto a single element architecture.

Compared to the triangular and the linear array architectures the single element architecture only consists of a single processing element, a delay, and a memory regardless of the number of inputs. The data flow in the single processing element architecture mimics the data flow of the original triangular array architecture. Since only one processing element is used no parallel computing is possible. Therefore, intermediate computations are stored in the memory during computations. The delay element is used to temporary postpone a value until the next cycle. The scheduling will be discussed in chapter 3. Scaling of this architecture to handle larger matrix sizes is done by increasing the memory sizes for storage of intermediate computations and increasing the computation time.

2.3.4 Architectural comparison

Table 2.1 shows a comparison between the architectures in terms of the number of processing elements needed to realize the architecture for an *n*-by-*n* matrix size, the level of scalability of the architecture, and the propagation delay through the architecture in terms of number of processing elements to pass through. From the table it is clear that the linear array structure has very attractive features for hardware implementation. It only uses *n* processing elements and has a propagation delay of only one processing element. Due to the modulized design of the processing elements in the linear array architecture it is very easy to scale (high scalability). To scale the SPE architecture the memory must be expanded and in the triangular array architecture new architecture and control circuitry must be derived. The pipelined triangular array architecture is not an alternative to the linear array architecture, even if area consumption is of less important. The single processing element architecture should only be used when resource consumption is of great concern.

Table 2.1 Comparison between the four different architectures for inverting a triangular matrix.

Architecture	# of PEs	Scalability	Propagation delay
Triangular (wave)	*n(n+1)/2*	low	*n* PEs
Triangular (pipelined)	*n(n+1)/2*	low	1 PE
Linear	*n*	high	1 PE
Single element	1*	medium	1 PE

* The number of PEs will not grow with the size of the input matrix but the memory size will. This must be taken into account when comparing the resource consumption of the different architectures.

In this section resource consumption and scalability of the different architectures have been investigated. Another important factor that must be investigated is the throughput rate of the architectures. A scheduling of the architectures must be done to be able to compare the architectures in terms of throughput rate.

2.4 Scheduling of the architectures

Scheduling the data flow in an architecture is an important step in hardware development [42]. In this work the scheduling is done in two steps:

1. In the first step, referred to as a basic scheduling, all processing elements are assumed to generate an output in one cycle. This is often not the case in reality but the basic scheduling lets us explore and evaluate different scheduling schemes.
2. When a processing element has been designed, step two in the scheduling can be initiated. In this step a more detailed scheduling is done by adjusting the basic scheduling depending on the amount of cycles needed to produce an output value. Since all processing elements are constructed in the same manner it is easy to expand the basic scheduling.

By investigating the data flow of the triangular array architecture, a schedule for the linear array and single element architecture can easily be derived.

2.4.1 Scheduling of the pipelined triangular array architecture

A basic schedule showing the inversion of a triangular matrix is shown in figure 2.9. The matrix data enters the architecture in a skewed manner (shown in figure 2.3) and propagates throughout the structure. The black boxes show which processing elements are active in each cycle. A complete matrix inversion is done in 10 cycles, assuming that a processing element takes one cycle to process the data. As seen in figure 2.9 not all processing elements are utilized during the inversion processes. The idle processing elements could either be turned off when inactive or they could be used to process a second inversion. The sequential data flow of the architecture simplifies the control circuitry, which can be implemented with a state machine.

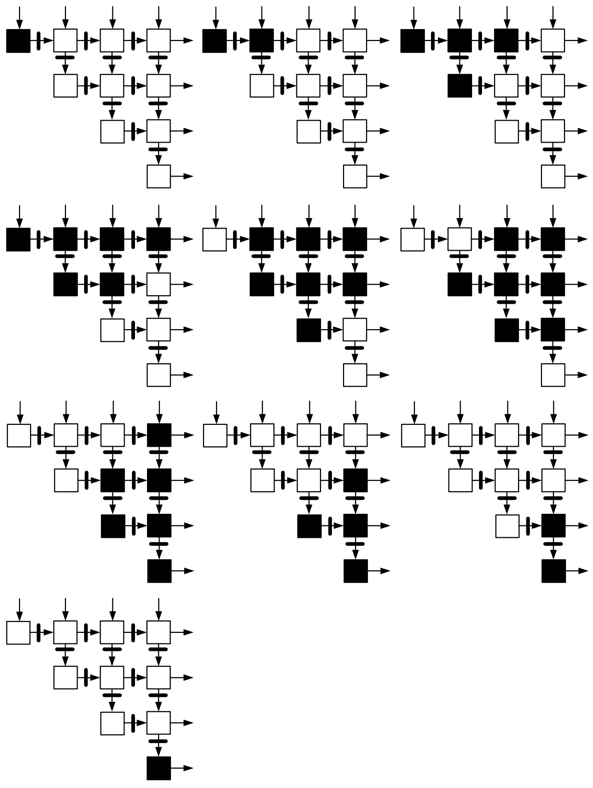
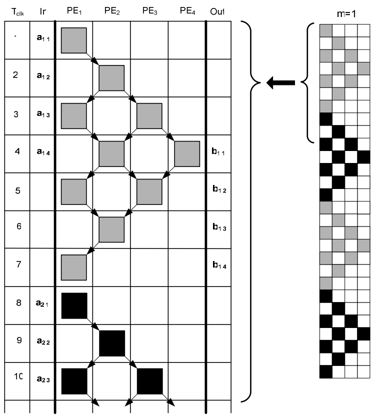


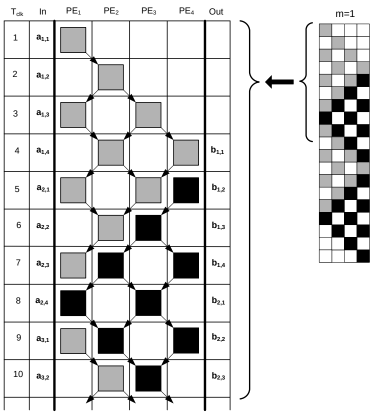
Figure 2.9 Data flow in the pipelined triangular array architecture during inversion of a triangular matrix.

2.4.2 Scheduling of the linear array architecture

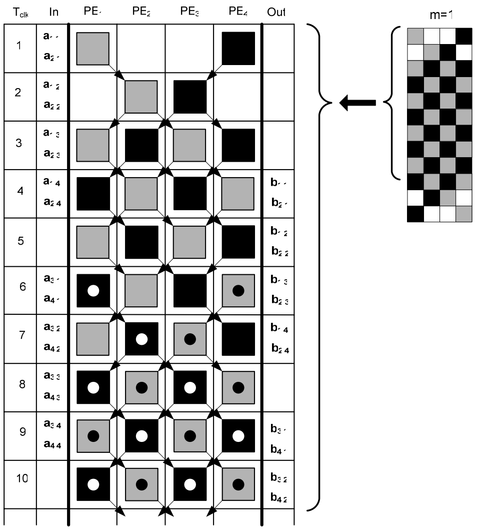
As shown in section 2.3.2 the triangular array architecture can be mapped onto a linear array architecture. Three basic scheduling schemes of the linear array architecture for inverting a 4-by-4 triangular matrix are shown in figure 2.10.



(a)



(b)



(c)

Figure 2.10 Three different basic schedules of the linear array architecture. Each processing element is scheduled to produce an output value in one cycle (*m*=1).

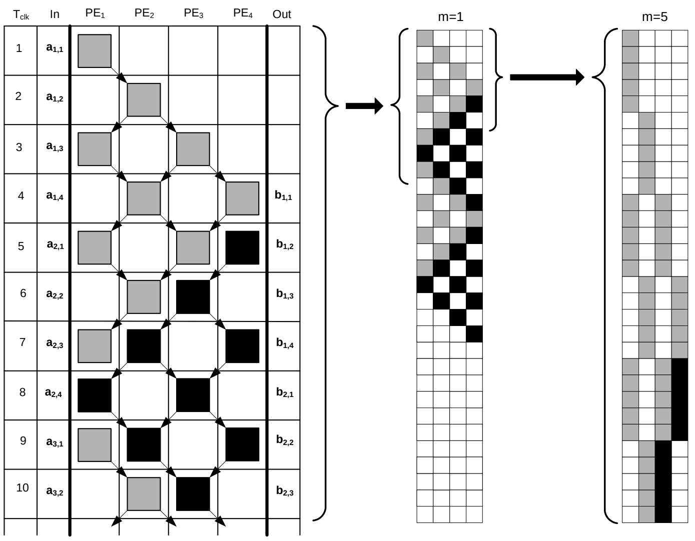
The scheduling in figure 2.10 (a) is a straightforward mapping of the data flow of the triangular wave architecture in figure 2.1. The boxes in the figure show which processing element is active in that particular cycle. The boxes form a larger triangular shaped structure, where different shades indicate the processing of different rows in the input matrix. The scheduling diagram is an enlarged part of the complete scheduling to the right in figure 2.10 (a). The directions of the arrows show where the results from a processing element are used in the next cycle. Drawbacks of this schedule are the low utilization of the processors and the irregular production rate, since the output values are not produced continuously. Maximum utilization is only 50%, e.g. cycle 3, and in most cycles the utilization is only 25%, e.g. cycle 5. The degree of utilization will vary with the matrix size. However it will never be higher than 50% at any given time.

To reduce power consumption the low utilization of the PEs could be utilized by turning off a particular PE when it is not used. This schedule inverts a triangular matrix in $2n^2-n$ cycles.

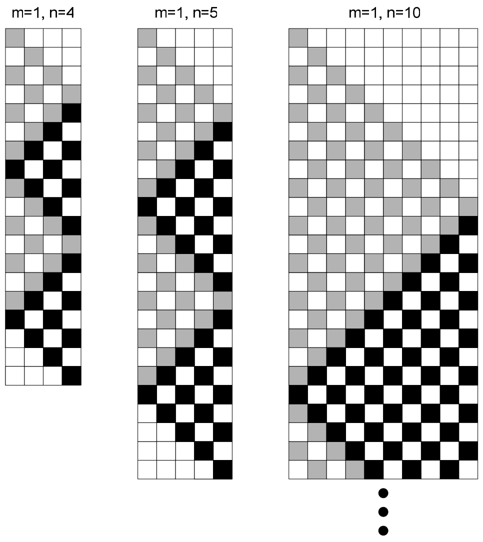
By interleaving processing of rows, the schedule in figure 2.10 (b) takes advantage of some of the idle processors to process the next row in parallel. The utilization in (b) is 50% or 75% at any given time (compare clock cycle 4 and 5 in figure 2.10 (b)). Also in this schedule the degree of utilization will vary with the matrix size, however it will never be lower than 50% at any given time. This schedule inverts a triangular matrix in n^2+n-1 cycles.

There are other possible schedules with a higher utilization degree. The schedule in figure 2.10 (c) will result in a 100% utilization. In this schedule two rows are processed simultaneously through the architecture. A drawback of this schedule is that it produces output samples in an irregular manner, which may not be desirable, and it can only be used for even matrix sizes (n=2,4,6,…). This schedule inverts a triangular matrix in $(2n-1)+((n/2)-1)(n+1)$ cycles. There are several other ways of scheduling the linear array but they are not treated in this thesis.

As discussed before, the scheduling will be affected by the implementation of the processing elements. The number of cycles needed to execute a processing element will change the scheduling diagrams shown in 2.10. As an example, let us assume that a linear array architecture is used with the schedule presented in 2.10 (b) and that a processing element can produce an output value in a singe cycle ($m=1$). If the processing element is pipelined so that it takes 5 cycles to produce an output value ($m=5$), the scheduling will change accordingly as shown in figure 2.11(a). The pipelining of a processing element will not change the fundamental data flow between the processing elements or the degree of hardware utilization, if the same schedule is kept. However, the pipelined architecture may give rise to new and more efficient schedules. The benefit of pipelining a processing element is that the clock frequency could be increased. Figure 2.11 (b) shows how the basic schedule changes with the input matrix size n. As shown, the schedule is expanding since the triangular shapes of processing elements are growing with the matrix size.



(a)



(b)

Figure 2.11 The effect on the scheduling when pipelining the processing elements. (b) shows how the schedule in (a) grows with the matrix size *n*.

The final schedule of the linear array architecture can not be done until the processing element has been developed. Table 2.2 summarizes the computation times of the three different schedules for inverting an *n*-by-*n* triangular matrix on a linear array architecture.

Table 2.2 Comparison between the three different schedules in terms of cycle needed to invert an *n*-by-*n* triangular matrix.

Schedule	# of cycles needed for inversion of an *n*-by-*n* matrix
Figure 2.10 (a)	$2n^2-n$
Figure 2.10 (b)	n^2+n-1
Figure 2.10 (c)	$(2n-1)+\left(\frac{n}{2}-1\right)(n+1)$

The scheduling in 2.10 (b) was chosen for usage with the linear array architecture for a number of reasons. It has a fair PE utilization, but foremost it has a regular input/output production rate. This helps when the architecture is going to be used together with other components in a larger system, especially if that system exhibits a regular data flow from the beginning. Thus complex buffering and control circuitry are not required.

2.4.3 Scheduling of the single element architecture

The scheduling of the single processing element architecture (SPE) is based on the schedule of the linear array architecture shown in figure 2.10 (a). Since only one processing element is used in the single processing architecture, parallel computations as in cycle 3, 4, and 5 of figure 2.10 (a) are not possible. The schedule is therefore mapped onto the single PE by delaying all parallel operations until a later cycle as shown in figure 2.12.

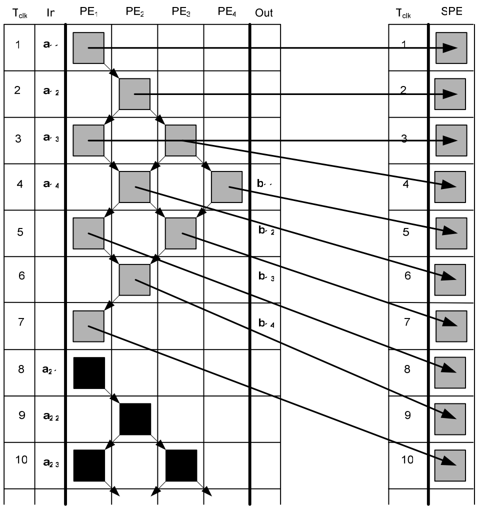


Figure 2.12 Mapping of a schedule for a multiprocessor architecture onto a schedule for a single processor architecture.

To facilitate the scheduling a memory unit was added [42]. Instead of circulating the data in delay chains, the delayed data is written to and read from a memory when needed or delayed one cycle using a delay element. Figure 2.13 shows a schedule for the SPE including the cycles in which the memory and the delay are active. This schedule inverts a triangular matrix in $n^2(n+1)/2$ cycles.

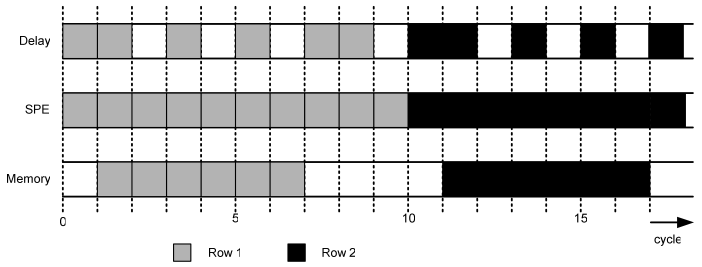


Figure 2.13 Scheduling of the single element architecture.

The scaling of the SPE architecture is done by reconfiguring the control circuit handling the order in which the operations are going to be performed. The memory size must also be increased, since more delayed data must be stored while waiting for their turn to be processed. Hence, the scalability is not that good compared to the linear architecture. However, the SPE architecture can be implemented with a large fixed size memory so that it can handle a wide range of matrix input sizes.

2.5 FPGA implementation of architectures for triangular matrix inversion

Both the linear array architecture and the single processing element architecture have been implemented on a Xilinx Virtex II XC2V1000 FPGA with speed grade 4. Each architecture has been implemented in two versions; one handling real valued matrices and one handling complex valued matrices. A combined hardware design handling both complex and real valued matrices is possible but not covered in this thesis. The only thing that would change compared to the complex valued implementation is the control circuitry regulating the data flow.

2.5.1 Hardware architecture of the linear array architecture

As explained in the previous sections, the triangular array architecture consists of two different processing elements, boundary processing elements and internal processing elements. The boundary processing elements include a subset of the same operations as the internal processing elements, and the hardware of the two elements can therefore be combined into a single processing element. A hardware block diagram of the combined processing element is shown in figure 2.14.

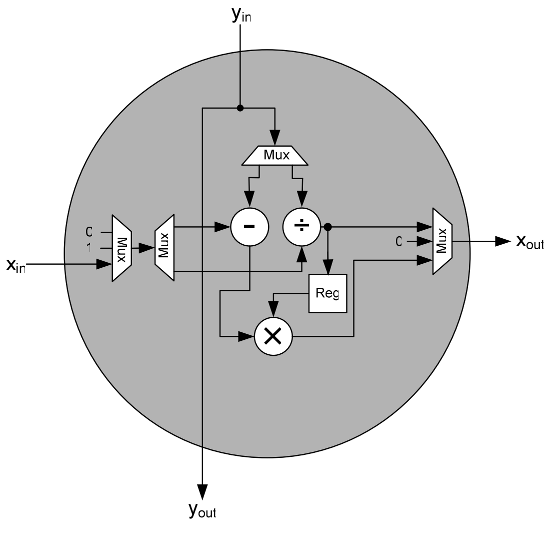


Figure 2.14 Hardware block diagram of a combined processing element.

The processing elements are capable of functioning in two modes with two sub-modes. Which arithmetic units that are active in the different modes are shown in table 2.3. A state machine within the control circuitry is keeping track of which modes and sub-modes to use, and when to use them. Mode 1 with sub-mode 1 is triggered by the arrival of a diagonal matrix element to the processing element, while mode 2 is used for every other case. The sub-modes are used when the processing element is performing the functions of the boundary element in the original triangular array architecture. When mode 1 is triggered the input values are divided by each other and stored in the register while input value Y_{in} is sent to the next PE. Sub-mode 1 is triggered when a diagonal element arrives. The diagonal element is inverted and stored. Mode 2 together with sub-mode 2 forms the sum in equation 2.6. The functionality of the different modes is summarized in table 2.3.

The data flow control system interconnecting the processing elements is shown in figure 2.15. The control system consists of data buses, registers, and multiplexers which are controlled by a state machine within the control circuitry. Since a mapping strategy based on the data flow in the triangular architecture was used, the flow control between the processing elements is very simple and regular. The input/output connections of processing elements on either end of the array are simply connected together forming a feedback loop as shown by the dashed line in figure 2.15.

Table 2.3 Summary of the different operation modes of the processing element.

Mode	Operations
Mode 1	$X_{in}/Y_{in} \rightarrow Register$ $Y_{in} \rightarrow Y_{out}$ $0 \rightarrow X_{out}$
Sub-mode 1	$1/Y_{in} \rightarrow Register$ $0 \rightarrow X_{out}$
Mode 2	$X_{in}-Y_{in} \cdot Register \rightarrow X_{out}$ $Y_{in} \rightarrow Y_{out}$
Sub-mode 2	$-Y_{in} \cdot Register \rightarrow X_{out}$

The linear array architecture and the processing elements have been implemented in two versions. The first version is an architecture capable of handling real valued input matrices, while the second version is capable of handling complex valued input matrices.

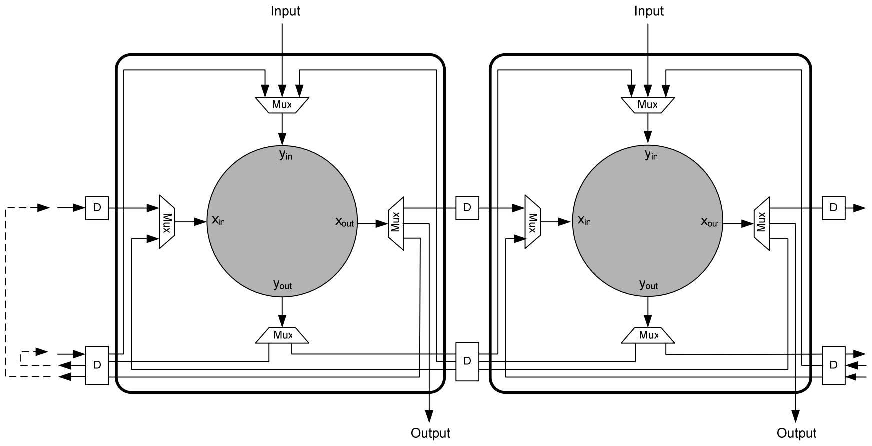


Figure 2.15 Data flow control system between two neighboring processing elements in the linear array architecture.

To keep rounding errors to a minimum, convergent rounding was used throughout the designs in this chapter. This bias-free rounding scheme avoids any statistical bias since it will round upward about 50% of the time and downward about 50% of the time. Saturation arithmetic ensures that rounded results that overflow (or underflow) the dynamic range will be clamped to the maximum positive (or negative) value. The maximum accuracy error of the implementation is below 1 unit in the last position (ulp).

2.5.2 Implementation of a real valued linear array architecture

The implementation of the linear triangular array architecture handling real valued input matrices was implemented in fixed-point with an operand wordlength of 8 bits. The real valued divider used in the implementation consists of two multipliers and a lookup table (appendix A). The same divider was used in the implementation of the complex divider in part I chapter 2 of this thesis. The divider needs two cycles to produce an output value. The multiplier and the adder/subtractor unit were implemented using Xilinx optimized building blocks needing 1 cycle each producing an output value. Since the division takes two cycles to produce an output value and the longest computation path (marked in figure 2.16) via the subtractor and the multiplier takes 1+1=2 cycles, there is no need to add additional delays except on the y_{in} to y_{out} line. The processing element shown in figure 2.16 now have a delay of 2 cycles (*m=2*).

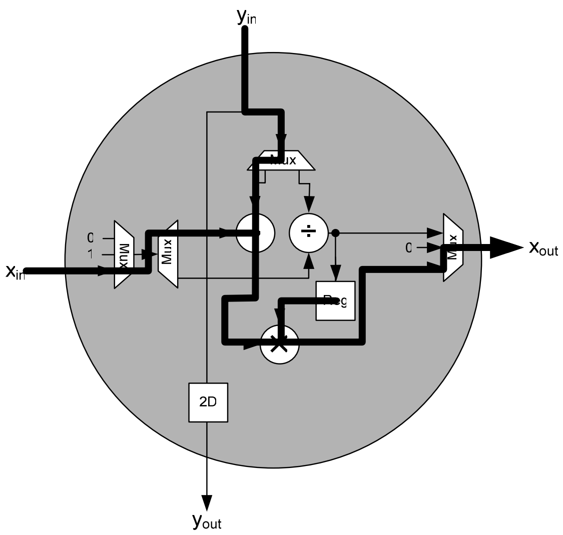


Figure 2.16 Hardware block diagram of the pipelined (*m=2*) processing element.

Table 2.4 shows the amount of resources occupied in the FPGA by a linear array architecture and by a single real processing element implementation with four real valued processing elements, it also indicates the percentage of the total resources of the FPGA that was consumed.

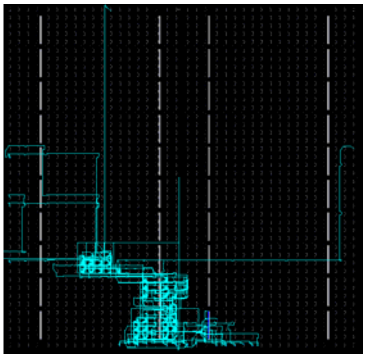
Table 2.4 Summary of the consumed resources of the real valued PE and the real valued linear array architecture.

	Real valued PE	**Real valued Array**
Number of Slices	162 (3%)	648 (12%)
Number of Slice Flip Flops	292 (2%)	1168(11%)
Number of 4 input LUTs	282 (2%)	1128 (11%)
Number of BRAMs	1 (2%)	4 (10%)
Maximum frequency (MHz)	107	103

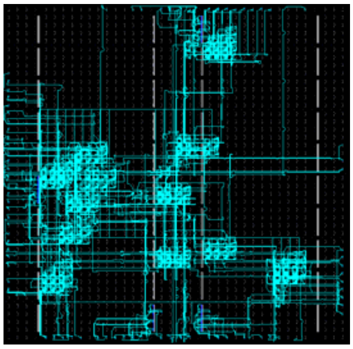
Table 2.4 shows that the design scales linearly from a single processing element using 162 slices to four processing element using 648 slices. A slice in an FPGA normally consists of two flip-flops, two lookup tables (LUTs), and some associated multiplexers, carry, and control logic. BRAMs is the number of block RAMs utilized in the design. The block RAMs implements the lookup tables in the design. It may also be note that

the maximum clocking frequency does not change significantly between the designs. The implementation can be run at a maximum clock frequency of 107 MHz.

Figure 2.17 shows the routed floorplan of the single real valued processing element in (a) and the linear array consisting of four processing elements in (b). The single real valued processing element is clustered in the lower part of the FPGA while the placing of the processing elements and the control circuitry for the linear array is scattered all over the floorplan. It is possible to manually place the design closer together or force the place and route tool to compact the design avoiding long interconnecting wires.



(a)



(b)

Figure 2.17 (a) Routed floorplan of the single real valued processing element. (b) Routed floorplan of the real valued linear array architecture.

2.5.3 Implementation of a complex valued linear array architecture

The linear array architecture handling complex valued input matrices was implemented in fixed-point with an operand wordlength of 8 bits for the real and imaginary parts. The hardware operations of the processing element do not change for a complex valued design. However, the individual arithmetic building blocks must be able to perform complex valued operations. The complex valued divider was implemented using the multiplexed complex valued divider presented in part I, chapter 2 of the thesis while the complex multiplier was implemented using the strength reduced computation scheme presented in part I, equation (3.2) and the complex valued adder/subtracter was implemented with using two real valued adder/subtractor units.

The complex valued divider produce an output value in 8 cycles and the complex multiplication and the complex subtractor is internally pipelined so that they produce an output value in 4 cycles. A delay element delaying 8 cycles are inserted on the y_{in} to y_{out} wire. A final scheduling was implemented with a delay of 8 cycles ($m=8$) using the processing element shown in figure 2.18.

Table 2.5 shows the amount of resources needed to implement a single complex valued processing element and the complex valued linear array architecture in an FPGA.

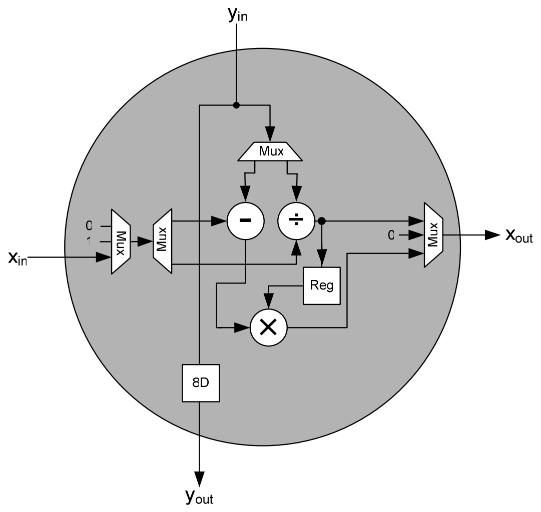


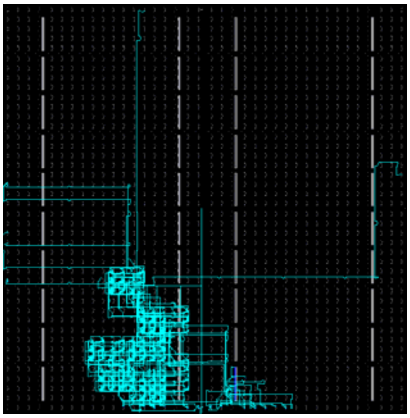
Figure 2.18 Hardware block diagram of the pipelined processing element handling complex values.

Table 2.5 Summary of the consumed resources of the complex valued PE and the complex valued linear array architecture.

	Complex valued PE	**Complex valued Array**
Number of Slices	320 (6%)	1278 (24%)
Number of Slice Flip Flops	553 (5%)	2212 (21%)
Number of 4 input LUTs	556 (5%)	2224 (21%)
Number of BRAMs	1 (2%)	4 (10%)
Maximum frequency (MHz)	103	101

Table 2.5 shows that the complex valued design also scales linearly in terms of consumed hardware resources. Both the complex valued processing element and the complex valued linear array implementations can be executed at a maximum clocking frequency of around 100 MHz. The implementation of the linear array architecture

consumes in total 1278 slices, which translates to roughly 24% of the total resources in the FPGA, without using any embedded hardware units. This is nearly twice the size of the real valued implementation. Figure 2.19 shows the routed floorplan of a single complex valued processing element in (a) and the complex valued linear array architecture with four processing elements in (b). The floorplan of the complex valued architecture suffers from the same scattering as the real valued architecture did.


(a)

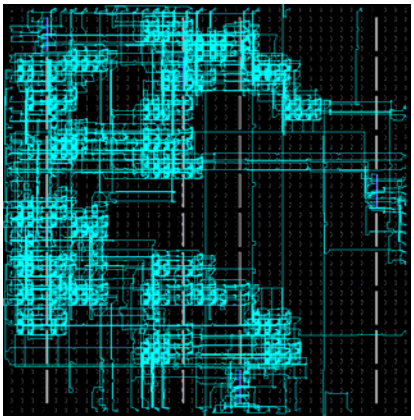

(b)

Figure 2.19 (a) Routed floorplan of the single complex valued processing element. (b) Routed floorplan of the complex valued linear array architecture.

2.5.4 Hardware architecture of single element architecture

The hardware block diagram of the single processing element architecture is shown in figure 2.20. The hardware architecture consists of three arithmetic units, a memory, and control circuitry (not shown in the figure).

The single element architecture has also been implemented in one version handling real valued input matrixes and one handling complex valued matrixes. The arithmetic units are implemented in the same way and with the same wordlength of the input operands as the arithmetic units in the linear array architecture. The multiplier and the adder/subtractor have been pipelined internally to match the computation time of the real valued (or complex valued) divider. In the case of the real valued implementation the divider uses two cycles to produce an output value while the multiplier and the subtractor produces an output value in one cycle and in the case of the complex valued implementation the division 8 cycles to produce an output value while the multiplier and the subtractor uses 4 cycles each to produce an output value. The scheduling scheme of the SPE has been changed accordingly (m=2 for real valued and m=8 for complex valued) to match the pipelining of the arithmetic hardware units. Table 2.6 shows the amount of resources occupied in the FPGA by the real and by the complex valued SPE implementation for inverting a 4-by-4 triangular matrix. It also indicates the percentage of the total resources of the FPGA that was consumed.

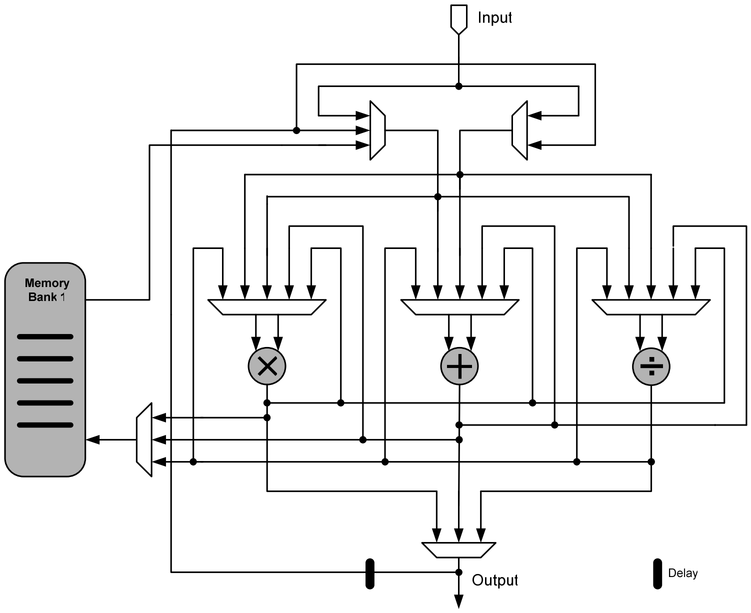


Figure 2.20 Hardware block diagram of the single processing element architecture.

Table 2.6 Summary of the consumed resources of the complex valued PE and the complex valued linear array architecture.

	SPE Real valued design	SPE Complex valued design
Number of Slices	158 (3%)	283 (5%)
Number of Slice Flip Flops	287 (2%)	524 (5%)
Number of 4 input LUTs	275 (2%)	491 (4%)
Number of BRAMs	2 (5%)	2 (5%)
Maximum frequency (MHz)	127	126

Table 2.6 shows that the real valued implementation is nearly 56% smaller than the complex valued implementation. This is due to an increase in the sizes of the arithmetic units and the memory in the complex valued implementation. The maximum frequency of the implementations is the same as in the real valued design. Figure 2.21 shows the routed floorplan of a real valued single processing element in (a) and the complex valued single processing element in (b). The floorplan of the real valued SPE is divided into two clusters. The upper cluster is the arithmetic units and the control circuitry while the lower cluster is the memory. The floorplan of the complex valued SPE architecture in figure 2.21 (b) is more compacted together.

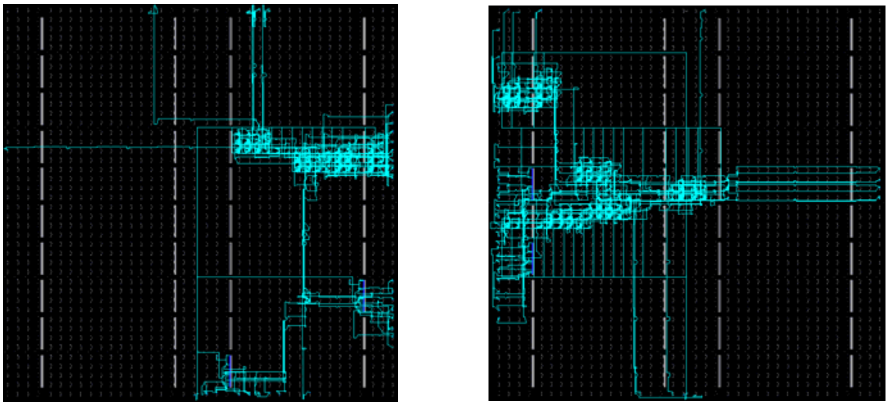

(a) (b)

Figure 2.21 (a) Routed floorplan of the single processing element architecture handling real values. (b) Routed floorplan of the complex valued single processing element architecture.

2.6 Summary and comparison

Table 2.7 shows a comparison of the three architectures from several different aspects. The table shows that the triangular array architecture consumes the highest amount of processing elements implementing an n-by-n matrix. The scalability of the architectures is compared in a relative scale using low, medium, and high as a ranking system. The linear array is modulized and the control circuitry does not need to be redesigned if extra modules are added. If a larger matrix is to be processed by the SPE architecture the memory size must be adjusted. If more inputs are to be added to the triangular array a new architecture and control circuitry must in most cases be derived. For the reasons mentioned, the linear architecture is deemed highly scalable while the SPE have a medium scalability and the triangular array architecture has a low scalability. All three designs have roughly the same maximum achievable clocking frequency. If hardware resources are scarce, either the linear or the SPE architecture should be chosen for implementation. If high throughput is needed, the triangular architecture is the best choice. The perfect compromise is the linear array architecture that needs only *O(n)* amount of resources and has a good throughput rate and very good scalability.

Both linear array implementations scale nearly linear with the number of processing elements. This makes it easy to predict the number of slices needed for a certain matrix size. The scale factor between a real valued design and a complex valued design is slightly below 2.

Table 2.7 Comparison between the three architectures for inversion of a triangular matrix.

	Triangular array	**Linear array**	**Single element**
# of PEs	$n(n+1)/2$	n	1
Scalability	low	high	medium
Cycles needed for inversion	$2(n+1)$	n^2+n-1	$n^2(n+1)/2$
# of slices in PE (real/complex)	162/320*	162/320	158/238
Size of control circuitry	small	medium	large
Maximum frequency (MHz)	99*	101	126

* Estimated values produced by the Xilinx ISE tool.

Chapter 3

QR-decomposition

3.1 Introduction

A very useful approach to solving matrix problems is to adopt a transformation that preserves the solution but simplifies the problem. A problem with applications in many fields is the least squares problem, which essentially states that, given a matrix X and a vector y, find a vector b that minimizes the expression $\|y - Xb\|_2^2$ [14],[37]. Least squares is a mathematical optimization technique that attempts to find a "best fit" to a set of data by attempting to minimize the sum of the squares of the ordinate differences (called residuals) between the fitted function and the data. This problem can be effectively solved by using QR-decomposition [14],[36]. QR-decomposition is not only used in solving the least squares problem, but also used in many other transforms and methods [14] for a wide variety of applications. It is safe to say that QR-decomposition (a.k.a. QR-decomposition) is a key technique in matrix computation, and therefore it is important to derive a useful hardware implementation of the decomposition technique.

3.2 QR-decomposition algorithms

A QR-decomposition of a matrix $\boldsymbol{A}$ is a decomposition of the matrix into

$$A = QR \tag{3.1}$$

where the matrix $\boldsymbol{Q}$ is orthogonal (meaning that $\boldsymbol{Q^T Q = I}$) and the matrix $\boldsymbol{R}$ is upper triangular. QR-decompositions can be computed in several different ways [14],[40], the most commonly used being:

- Gram-Schmidt decomposition.
- Householder reflections, also known as Householder transformations.
- Givens rotations.

All three methods have their strengths and weaknesses [14]. When it comes to hardware implementation, all three methods have been considered for implementation however, in later years the Givens rotations method, or short Givens, has been given lots of attention from hardware engineers around the world. Givens has proven to be easy to parallelize, has a good algorithm-to-hardware feasibility, and has good numerical properties which is important parameters in hardware implementation [14],[40]. Therefore, the Givens method was chosen for implementation in this thesis.

3.2.1 Standard Givens rotations algorithm

The standard, or generalized, Givens rotations is a well-known technique, which was first proposed by Givens [43]. A Givens rotation is a matrix on the form

$$G = \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \text{ where } c^2 + s^2 = 1 . \tag{3.2}$$

Each rotation performed by this matrix is orthogonal. There is a unique angle θ such that *$c=\cos(\theta)$* and *$s=\sin(\theta)$*. By rotating a vector $(a\ b)^T$ clockwise through the angle θ, the expression

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix}\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} ca + sb \\ cb - sa \end{pmatrix} \tag{3.3}$$

is obtained. In this way the rotations can be used to force a matrix element to be zero by setting

$$c = \frac{a}{\sqrt{a^2 + b^2}} \text{ and } s = \frac{b}{\sqrt{a^2 + b^2}} \text{ when } (a\ b)^T \neq 0 \tag{3.4}$$

If inserted into equation (3.3) we get

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix}\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \sqrt{a^2 + b^2} \\ 0 \end{pmatrix}. \tag{3.5}$$

Applied to a larger matrix, a rotation *$G(p,q,\theta)$* in the *(p,q)* plane is defined as a matrix of the form

$$G(p,q,\theta) = \begin{array}{cc} \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} & \begin{array}{c} \\ \\ p \\ \\ q \\ \\ \\ \end{array} \\ \begin{array}{ccccccc} & & p & & q & & \end{array} & \end{array} \tag{3.6}$$

where *c=cos(θ)* and *s=sin(θ)* for an angle *θ*. The rotation in the *(p,q)*-plane is clearly an identity matrix, in which a plane rotation has been embedded in a submatrix corresponding to the rows *p* and the columns *q* in the matrix. If

$$y = G(p,q,\theta)x \tag{3.7}$$

then

$$y_r = \begin{cases} sx_p + cx_q, & r = q \\ cx_p - sx_q, & r = p \\ x_r, \text{ when } & r \neq p,q \end{cases} \tag{3.8}$$

The rotation combines the rows *p* and *q* and leaves the others undisturbed. By choosing *c* and *s* carefully we can introduce a zero anywhere we want in the p^{th} and q^{th} row. In a similar manner a multiplication with $G(p,q,\theta)^T$ we can introduce a zero anywhere we want in the p^{th} and q^{th} column.

Example 3.1: QR-decomposition by Givens rotations

Givens rotations can be used to compute the QR-decomposition of a matrix. Consider a matrix $A \in \mathbb{R}^{4\times3}$. The Wilkinson diagrams show how Givens rotations transform the matrix into an upper triangular form.

$$A = \begin{pmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{pmatrix} \xrightarrow{G_1^T(3,4)} \begin{pmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ 0 & \times & \times \end{pmatrix} \xrightarrow{G_2^T(2,3)} \begin{pmatrix} \times & \times & \times \\ \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \end{pmatrix} \xrightarrow{G_3^T(1,2)} \begin{pmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \end{pmatrix} \xrightarrow{G_4^T(3,4)}$$
$$\begin{pmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \end{pmatrix} \xrightarrow{G_5^T(2,3)} \begin{pmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ 0 & 0 & \times \end{pmatrix} \xrightarrow{G_6^T(3,4)} \begin{pmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ 0 & 0 & 0 \end{pmatrix} = R \tag{3.9}$$

If G_i denotes the *i*th Givens rotation in the reduction of ***A*** into ***R*** then the orthogonal matrix ***Q*** takes the form of all plane rotations $\boldsymbol{Q=G_1 \cdot G_2 \cdot \ldots \cdot G_6}$.

□

The computation of *c* and *s* shown in equation (3.5) is not practical in a hardware implementation. The expressions contain two of the most computationally heavy operations namely square rot and division. There are several variations of the Givens rotation algorithm that are more suitable for implementation [40],[45]. In short, the majority of variations of the Givens rotation algorithms can be grouped into the following classes:

- The standard or conventional algorithm.
- Normalised algorithms.
- Square root free algorithms.
- Divide and square root free algorithms.

The last two classes of algorithms are the most interesting for hardware implementation, since one or both of the computationally heavy operations is avoided. The divide and square root free algorithms come with a price of reduced numerical accuracy, which must be considered when choosing an algorithm [45]. The divide operation is traded at the cost of several multiplications, more complex internal cell structure, and more extensive communication between the cells. For these reasons a divide and square root free algorithm was not chosen to for hardware implementation in this thesis. The algorithm used in this thesis was first proposed by Döhler and is a modification of Hammarlings scaled Givens rotation algorithm [44]. Döhlers approach only requires about half the number of multiplications needed by the standard Givens to derive ***R***, and is also square-root-free.

Döhler's algorithm is called Squared Givens Rotation (SGR) and has good algorithm-to-hardware feasibility and numerical stability, which makes it suitable for implementation and was therefore chosen in this thesis.

3.2.2 Squared Givens rotations algorithm

The standard Givens rotation algorithm described in equations (3.2)-(3.8) can easily be rewritten for complex numbers [14]. The rotation equation can then be expressed as

$$\begin{pmatrix} c & s^* \\ -s & c \end{pmatrix}\begin{pmatrix} r_b & r_i \\ x_b & x_i \end{pmatrix}=\begin{pmatrix} r_b' & r_i' \\ 0 & x_i' \end{pmatrix} \tag{3.10}$$

where r, x, s, and c are complex values and s^* is the complex conjugate of s. The Givens rotation can be generalized by introducing two scaling terms, $d^{1/2}$ and $\delta^{1/2}$, using the following matrix substitution

$$\begin{pmatrix} r_b & r_i \\ x_b & x_i \end{pmatrix}=\begin{pmatrix} d^{1/2} & 0 \\ 0 & \delta^{1/2} \end{pmatrix}\begin{pmatrix} \overline{r}_b & \overline{r}_i \\ \overline{x}_b & \overline{x}_i \end{pmatrix} \tag{3.11}$$

When making the substitution into equation (3.10) we get the generalized rotation matrix

$$\begin{pmatrix} c & s^* \\ -s & c \end{pmatrix}\begin{pmatrix} d^{1/2}\overline{r}_b & d^{1/2}\overline{r}_i \\ \delta^{1/2}\overline{x}_b & \delta^{1/2}\overline{x}_i \end{pmatrix}=\begin{pmatrix} d^{1/2}\overline{r}_b\,' & d^{1/2}\overline{r}_i\,' \\ 0 & \delta^{1/2}\overline{x}_i\,' \end{pmatrix} \tag{3.12}$$

which can be rearranged into

$$\begin{pmatrix} \dfrac{cd^{1/2}}{d'^{1/2}} & \dfrac{s^*\delta^{1/2}}{d'^{1/2}} \\ \dfrac{-sd^{1/2}}{\delta^{1/2}} & \dfrac{c\delta^{1/2}}{\delta^{1/2}} \end{pmatrix} \begin{pmatrix} \overline{r}_b & \overline{r}_i \\ \overline{x}_b & \overline{x}_i \end{pmatrix} = \begin{pmatrix} \overline{r}_b' & \overline{r}_i' \\ 0 & \overline{x}_i' \end{pmatrix} \tag{3.13}$$

The update values of the generalized Givens can then be written as

$$\begin{cases} \overline{r}_i' = \left(\dfrac{cd^{1/2}}{d'^{1/2}}\right)\overline{r}_i + \left(\dfrac{s^*\delta^{1/2}}{d'^{1/2}}\right)\overline{x}_i \\ \overline{x}_i' = \left(\dfrac{c\delta^{1/2}}{\delta^{1/2}}\right)\overline{x}_i - \left(\dfrac{sd^{1/2}}{\delta^{1/2}}\right)\overline{r}_i \\ \overline{r}_b'^2 = \left(\dfrac{d\overline{r}_b^2 + \delta|\overline{x}_b|^2}{d'}\right) \end{cases} \tag{3.14}$$

The square-root-free Givens rotation algorithm developed by Döhler can be obtained by making the substitution

$$d = \frac{1}{\overline{r}_b},\ d' = \frac{1}{\overline{r}_b'},\ \text{and } \delta' = c^2\delta \tag{3.15}$$

in equation into the generalized equations. The update values can then be expressed as

$$\begin{cases} \overline{r}_i' = \overline{r}_i + \delta'\overline{x}_b^*\overline{x}_i \\ \overline{x}_i' = \overline{x}_i - \dfrac{\overline{x}_b}{\overline{r}_b}\overline{r}_i \\ \overline{r}_b'^2 = \overline{r}_b + \delta'|\overline{x}_b|^2 \end{cases} \tag{3.16}$$

The equations are known as the SGR algorithm and are often implemented using a triangular array architecture where x_b is the input to the boundary cell, r_b represent the stored parameter of the boundary cell, and x_i and r_i are, respectively, the input and stored parameter of the internal cell.

3.3 Architectures for QR-decomposition

The SGR algorithm can be mapped onto a triangular array architecture in the same manner as the recurrence algorithm in chapter 2 [46],[48]. Figure 3.1 shows a triangular array architecture for QR-decomposition of a 4-by-4 matrix.

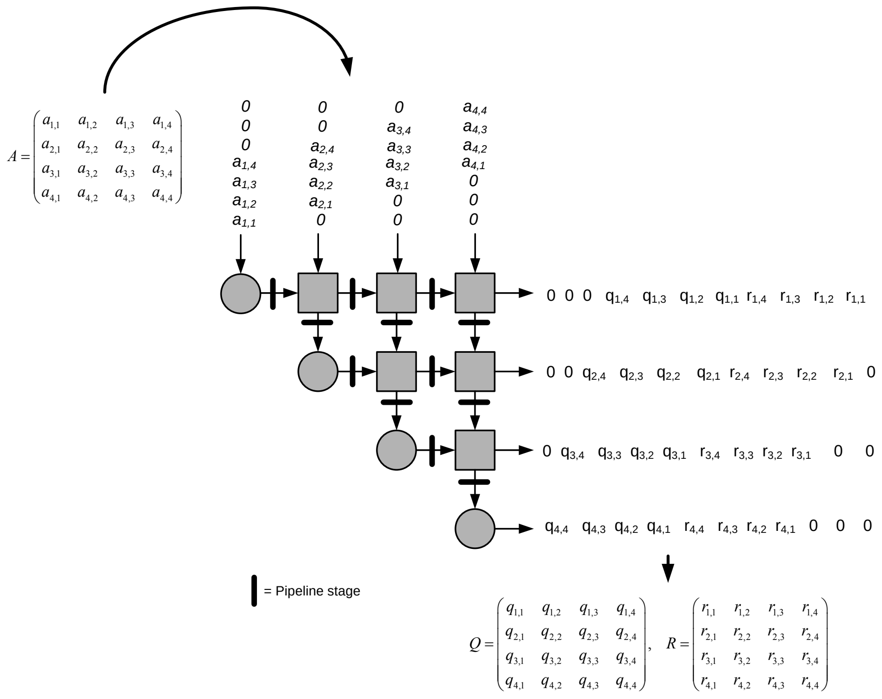


Figure 3.1 Triangular array architecture for QR-decomposition of a 4-by-4 matrix ***A***.

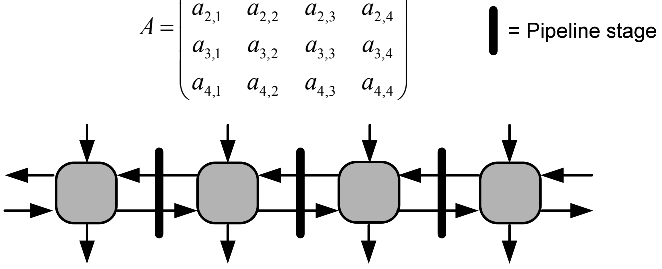
Since the architectures used in QR-decomposition of a matrix are of the same type as the architectures presented in chapter 2, only the differences separating the architectures are going to be explained. For a more detailed description about the architectures please consult chapter 2.

The processing of data through the architecture differs slightly from the case of the architectures for triangular matrix inversion. Every cycle, one row of the input matrix A is fed into the structure in a skewed manner and propagated through the structure. The factorization of ***A*** into the ***R*** and ***Q*** matrix is done by the SGR algorithm on the fly with the two operations fully overlapped in time. The ***Q*** matrix follows immediately after the ***R*** matrix resulting in the production of two matrixes for each input matrix as shown in figure 3.1. The ***R*** matrix is produced on the fly in the architecture while the ***Q*** matrix is stored in the processing elements. Architectures presented in literature often input an identity matrix ***I*** after the ***A*** matrix to purge the matrix of the stored values making up the matrix ***Q*** [38]. However, in this implementation there is no need for the identity matrix to be used since the purge function has been built-in. This will be discussed in section 3.6.

The triangular array architecture shown in figure 3.1 suffers from the same problems as the architecture described in section 2.5, and consequently a more suitable linear architecture was derived.

The triangular array architecture was mapped onto a linear array architecture shown in figure 3.2 [46]. The linear array architecture mimics the data flow of the triangular array architecture.

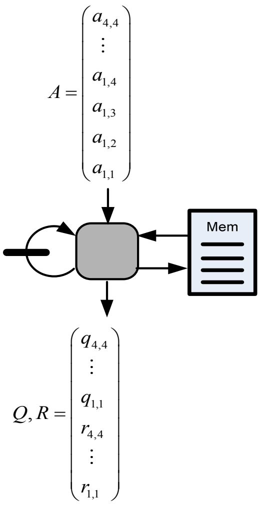
$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{pmatrix}$$



$$Q = \begin{pmatrix} q_{1,1} & q_{1,2} & q_{1,3} & q_{1,4} \\ q_{2,1} & q_{2,2} & q_{2,3} & q_{2,4} \\ q_{3,1} & q_{3,2} & q_{3,3} & q_{3,4} \\ q_{4,1} & q_{4,2} & q_{4,3} & q_{4,4} \end{pmatrix}, \quad R = \begin{pmatrix} r_{1,1} & r_{1,2} & r_{1,3} & r_{1,4} \\ r_{2,1} & r_{2,2} & r_{2,3} & r_{2,4} \\ r_{3,1} & r_{3,2} & r_{3,3} & r_{3,4} \\ r_{4,1} & r_{4,2} & r_{4,3} & r_{4,4} \end{pmatrix}$$

Figure 3.2 Linear array architecture for QR-decomposition of a 4-by-4 matrix $\boldsymbol{A}$.

A SPE architecture for the QR-decomposition was also derived in the same manner as shown in chapter 2. Figure 3.3 shows the SPE architecture for QR-decomposition of a 4-by-4 matrix $\boldsymbol{A}$.



$A = \begin{pmatrix} a_{4,4} \\ \vdots \\ a_{1,4} \\ a_{1,3} \\ a_{1,2} \\ a_{1,1} \end{pmatrix}$
Mem
$Q, R = \begin{pmatrix} q_{4,4} \\ \vdots \\ q_{1,1} \\ r_{4,4} \\ \vdots \\ r_{1,1} \end{pmatrix}$

Figure 3.3 Single element architecture for QR-decomposition of a 4-by-4 matrix ***A***.

3.4 Scheduling of the QR-decomposition architectures

As shown in the previous section the QR-decomposition algorithm can be mapped onto the same type of architecture as the triangular matrix inversion architecture in chapter 2. Since the basic data flow in the architecture is exactly the same, the same scheduling can be used. However, there are one major difference between the architectures in chapter 2 and the QR-decomposition architecture and that is that two matrices are produced instead of one. This will not influence the basic data flow. From a data flow point of view it will be the same as computing two matrix operations after each other.

The main difference between the architectures is what operations that are executed inside the processing elements.

3.4.1 Scheduling of the linear QR-decomposition array architecture

Assuming that a processing element only takes 1 cycle to produce an output value, a basic scheduling can be derived for the QR-decomposition. The schedule for QR-decomposition of a 4-by-4 triangular matrix is shown in figure 3.4.

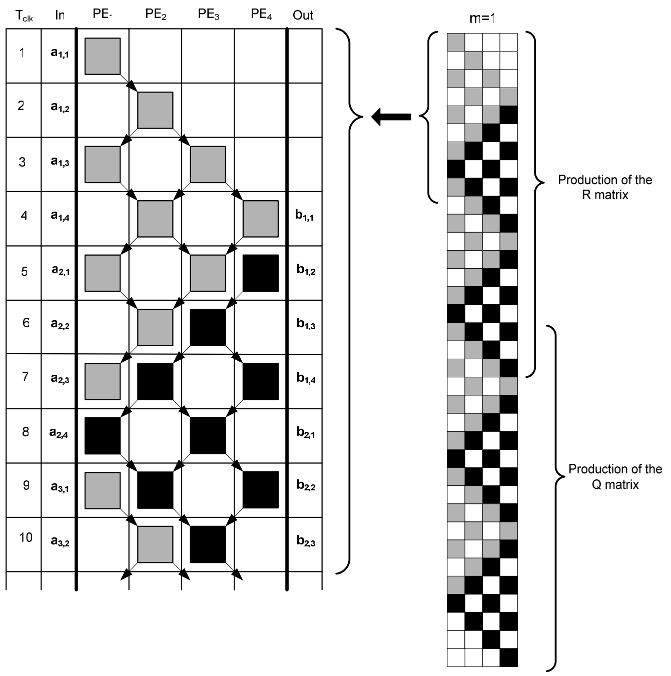


Figure 3.4 Scheduling of the linear array architecture for QR-decomposition.

The schedule in figure 3.4 is a straightforward mapping of the data flow in the triangular array architecture. The computation of ***Q*** and ***R*** is overlapped, and the schedule on the right in the figure shows how the PEs first process the matrix data and then work internally as they would if an identity matrix was the input into the system. The processor utilization is 50% or 75% at any given time in this schedule. The degree of utilization will vary with the matrix size, but it will never be lower than 50% at any given time. This schedule performs a QR-decomposition of an n-by-n matrix in $2n^2+n-1$ cycles.

3.4.2 Scheduling of the single element QR-decomposition architecture

The scheduling of the single processing element architecture is, as in chapter 2, based on the schedule of the linear array architecture shown in figure 2.10. Since only one processing element is used in the single processing architecture, several ongoing computations as in the linear and the triangular array architecture are not possible. The schedule is therefore mapped onto the single PE as shown in chapter 2. The only difference in the scheduling is the purge phase as discussed above. This schedule performs a QR-decomposition of an n-by-n matrix in $n^2(n+1)$ cycles. For a more detailed description of the scheduling of the SPE array architecture, please refer to chapter 2.

3.5 FPGA implementation of complex valued QR-decomposition

Two architectures, the linear array architecture and the single processing element architecture handling complex valued input matrices, have been implemented in hardware. Hardware implementations handling real valued matrices can easily be derived by exchanging the arithmetic building blocks in the processing element.

3.5.1 Hardware architecture of the linear array architecture

As explained in the previously in chapter 2 the triangular array structure consists of two different kinds of processing elements, boundary cells and internal cells. These two elements are combined into a single processing element in the linear array architecture by the mapping procedure from a triangular to a linear array architecture. An internal view of the combined processing element is shown in figure 3.5.

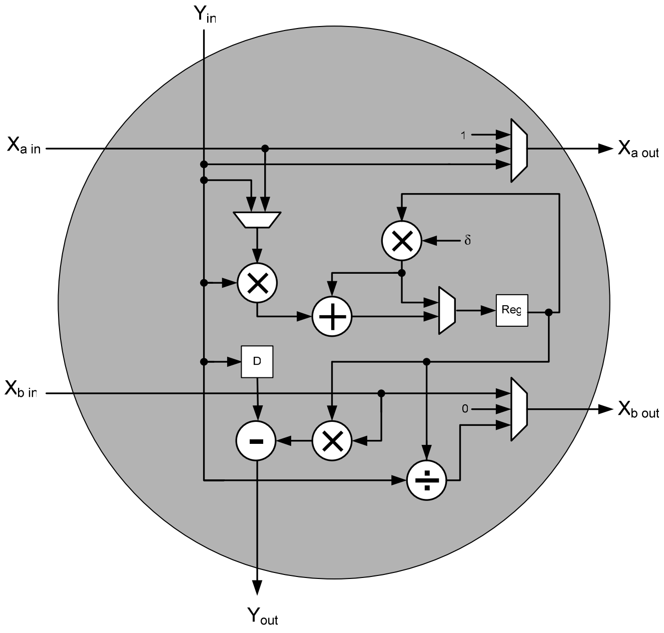


Figure 3.5 Internal cell view of the linear array architecture.

The processing elements are capable of functioning in two modes, one sub-mode, and a purge mode. A state machine in the control circuitry is keeping track of when to use the different modes. Mode 2 with sub-mode 2 is triggered by the arrival of a diagonal matrix element to the processing element, while mode 1 is used for every other case. When the ***R*** matrix is produced the purge mode is activated. For one cycle the processing element is set to purge mode. This will mimic loading an identity matrix into the architecture. The sub-mode is only used when the processing element in the linear array architecture is acting as a boundary element in the triangular array architecture. The different modes are shown in table 3.1. In the table δ is the scaling factor which is implemented as a constant value.

The data flow control system of the QR-decomposition architecture is the same as the one in chapter 2. It is comprised of data buses, registers, and multiplexers and is controlled by a state machine. The data flow between two processing elements is shown in figure 3.6. The dashed lines shows how a processing element located at either end of the array should be connected. The feed-back loops ensure that the data is circulated in the right manner.

Table 3.1 Summary of the different operation modes of the QR-decomposition processing element.

Mode	Operations
Mode 1	$X_{a\,in} \cdot Y_{in} + \delta \cdot Reg \rightarrow Reg$
	$Y_{in} - X_{b\,in} \cdot Reg \rightarrow Y_{out}$
	$X_{a\,in}, X_{b\,in} \rightarrow X_{a\,out}, X_{b\,out}$
Mode 2	$Y_{in}^{2} + \delta \cdot Reg \rightarrow Reg$
	$Y_{in} \rightarrow X_{a\,out}$
	$Y_{in}/Reg \rightarrow X_{b\,out}$
Sub-mode 2	$Y_{in}^{2} + \delta \cdot Reg \rightarrow Reg$
	$Y_{in} \rightarrow X_{a\,out}$
	$0 \rightarrow X_{b\,out}$
Purge mode	$\delta \cdot Reg \rightarrow Reg$
	$1 \rightarrow X_{a\,out}$

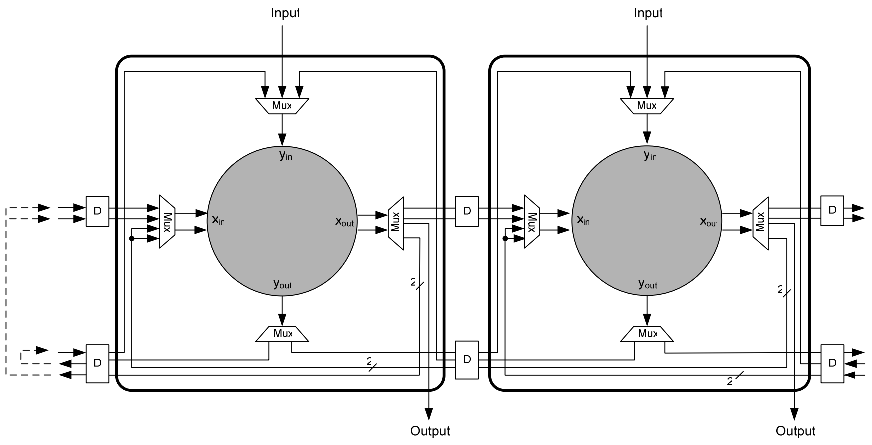


Figure 3.6 Interconnection between two neighboring processing elements in the linear array architecture.

The complex valued linear array architecture and the complex valued single processing element architecture have been implemented in a Xilinx Virtex II XC2V1000 FPGA with speed grade 4. The same rounding scheme and saturation logic that was used in chapter 2 is used in this implementation.

3.5.2 Implementation of a complex valued linear array architecture for QR-decomposition

The implementation of the linear array architecture handling complex valued input matrices was implemented in fixed-point with an operand wordlength of 8 bits for the real part and 8 bits for the imaginary part. The complex valued divider was implemented using the multiplexed complex valued divider presented in part I chapter 2 of the thesis. The complex multiplier was implemented using the strength reduced computation scheme presented in part I, equation (3.2). The complex valued adder/subtractor was implemented using two real valued adder/subtractor units. The complex valued divider produce an output value in 8 cycles and the complex multiplication and the complex subtractor is internally pipelined so that they produce an output value in 4 cycles. The final scheduling was implemented with a delay of 8 cycles. The block diagram of the pipelined processing element is shown in figure 3.7.

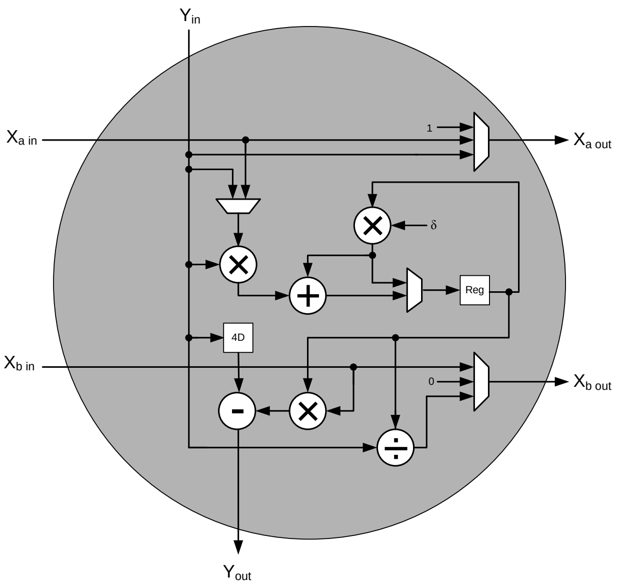


Figure 3.7 Internal cell view of the pipelined linear array architecture.

Table 3.2 shows the amount of resources needed to implement a single complex valued processing element and the complex valued linear array architecture performing a QR-decomposition in an FPGA.

Table 3.2 Consumed resources of the complex valued PE and the linear array architecture.

	Complex valued PE	**Complex valued Array**
Number of Slices	379 (7%)	1666 (32%)
Number of Slice Flip Flops	708 (6%)	2916 (28%)
Number of 4 input LUTs	668 (6%)	2912 (28%)
Number of BRAMs	2 (5%)	4 (10%)
Maximum frequency (MHz)	105	101

A complex valued QR-decomposition of a 4-by-4 matrix consumes in total 1666 slices which translates to roughly 32% of the total resources in the FPGA. The table 3.2 shows that four single elements can not estimate how much a linear array architecture consisting of four processing element consumes in terms of hardware. The difference between an estimate and the implemented design is as much as 150 slices. The increase is mainly due to the flow control circuitry and registers between the processing elements, and a poor place and routing of the design. The implementation can be run at a maximum clock frequency of 101 MHz. Figure 3.8 shows the routed floorplan of a single complex valued processing element in (a) and the four processing element linear array in (b). The processing elements are well clustered in both designs.

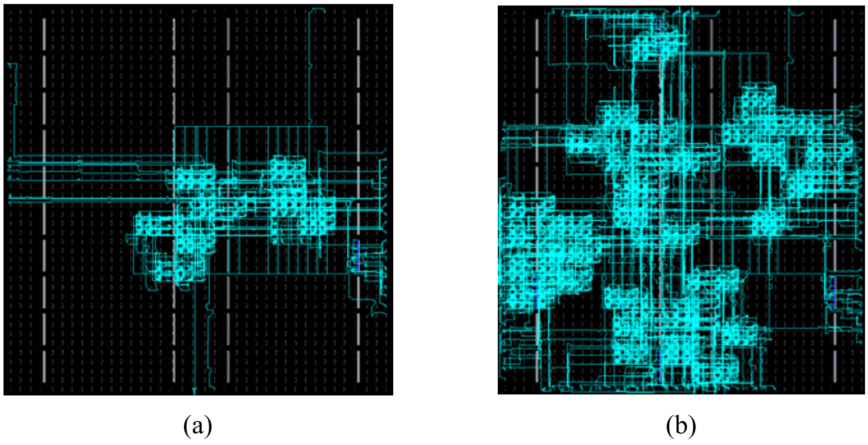


Figure 3.8 (a) Routed floorplan of a single complex valued processing element. (b) Routed floorplan of the complex valued linear array architecture.

3.5.3 Implementation of a complex valued QR-decomposition using a single element architecture

The hardware block diagram of the single processing element architecture is shown in figure 3.9. The hardware architecture consists of the same hardware building blocks as the SPE architecture use in chapter 2. The memory bank was moved to the other side in the figure so that it can be differentiated from the SPE architecture in chapter 2.

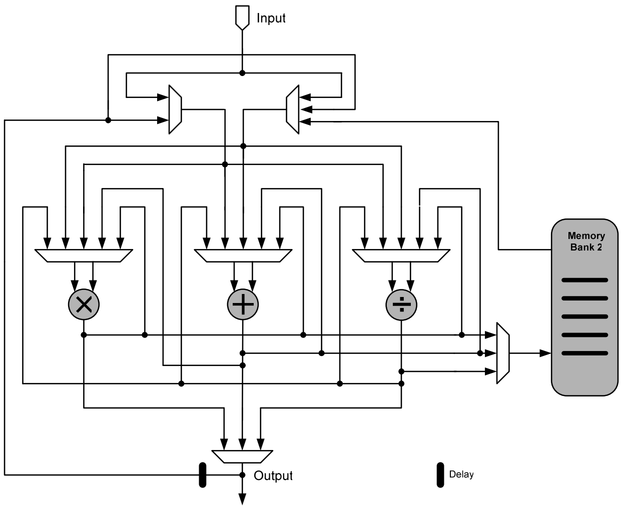


Figure 3.9 Hardware block diagram of the single processing element architecture.

The arithmetic units are implemented in the same way and with the same wordlength of the input operands as the arithmetic units in the linear array architecture. The only difference is that the multiplier and the adder/subtractor have been pipelined internally to match the complex valued divider. Since the Q matrix is stored in the processing elements of the triangular array architecture, the memory unit must also be used for storing the **Q** matrix. The memory is therefore expanded to be able to store the complete 4-by-4 complex valued **Q** matrix.

Table 3.3 shows the amount of resources the complex valued SPE implementation for QR-decomposition of a 4-by-4 matrix consumes in the FPGA.

Table 3.3 Consumed resources in the FPGA of the complex valued SPE implementation performing a QR-decomposition.

	SPE complex valued design
Number of Slices	417 (8%)
Number of Slice Flip Flops	729 (7%)
Number of 4 input LUTs	728 (7%)
Number of BRAMs	3 (7%)
Maximum frequency (MHz)	107

Table 3.3 shows that the design consumes 417 slices and 3 BRAMs. The corresponding design in chapter 2 consumed only 283 slices and 2 BRAMs. The difference of 134 slices and 1 BRAM is due to the extra memory and control circuitry. The maximum achievable clock frequency has also dropped from 127 to 107 MHz.

Figure 3.10 shows the routed floorplan of a complex valued single processing element able to perform a QR-decomposition on a complex valued matrix. The complex valued SPE implementation is clustered together at one end of the FPGA floorplan.

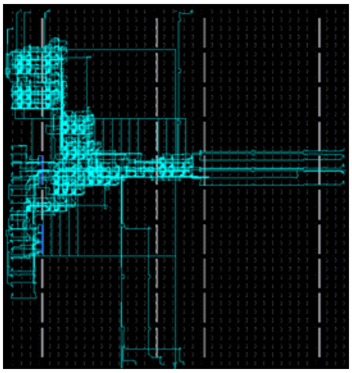


Figure 3.10 Routed floorplan of a complex valued SPE.

3.5.4 Summary of the hardware implementations

The complex valued linear array architecture was implemented using 1666 slices while the SPE architecture only needed 417 slices, i.e. 25%. However, the SPE architecture needs $O(n^3)$ cycles to complete a QR-decomposition, while the linear array architecture needs $O(n^2)$ cycles, which makes this a classic trade-off between throughput rate and resource consumption.

The linear array is modulized to its design and the control circuitry does not need to be redesigned if extra modules are added. If a larger matrix is to be processed by the SPE architecture the memory size must be adjusted and the control circuitry must be modified. Both designs have roughly the same maximum achievable clocking frequency.

The complex valued SPE implementation consumed 134 slices and 1 BRAM more than the equivalent real valued design in chapter 2. This is mainly due to the extra memory and control circuitry needed for the complex case. The maximum achievable clock frequency dropped from 127 to 107 MHz in the complex valued design compared to the real valued design.

Chapter 4

Matrix inversion

4.1 Introduction

Matrix inversion is, together with matrix addition and matrix multiplication, a fundamental matrix operation. However, due to its computational complexity, matrix inversion is often tried to be avoided in hardware implementations. If the matrix inversion can not be avoided it is common to implement the inversion in software instead of hardware, often resulting in poor performance. A complex valued division is even more computation complex and often requiring a lot of hardware resources to be implemented. In this chapter an architecture that is resource conservative, have good throughput rate, and is scalable, for complex valued division is presented.

In this chapter the triangular matrix inversion architecture in chapter 2 and the QR-decomposition architecture in chapter 3 are combined to form a complex valued matrix inversion architecture.

4.2 Definition of matrix inversion

In mathematics and especially linear algebra, an n-by-n (square) matrix A is called invertible, non-singular, or regular, if there exists another n-by-n matrix B such that

$$A \cdot B = B \cdot A = I, \tag{4.1}$$

where $\boldsymbol{I}$ denotes the n-by-n identity matrix and the multiplication used is ordinary matrix multiplication. If this is the case, then the matrix $\boldsymbol{B}$ is uniquely determined by $\boldsymbol{A}$ and is called the inverse of $\boldsymbol{A}$, denoted by $\boldsymbol{A}^{-1}$. A square matrix that is not invertible is called singular. However as a rule of thumb, almost all matrices are invertible.

4.3 Inversion algorithms

There are several ways of computing the inverse of a matrix. Gauss-Jordan elimination is an algorithm that can be used to determine whether a given matrix is invertible and to find its inverse [14]. An alternative method is LU-decomposition, which generates an upper and a lower triangular matrix, which are easier to invert [14]. There are also some analytic solutions that can be used.

Another way to invert a matrix is to use a method based on QR-decomposition. The first step in this method involves a QR-decomposition performed by the squared Givens rotations algorithm (SGR) presented in chapter 3. The SGR uses successive plane rotations to reduce the input matrix $\boldsymbol{A}$ to an upper triangular matrix, $\boldsymbol{R}$, and a matrix $\boldsymbol{Q}$**,** which is composed of the orthogonalized columns of **A**. The triangular matrix $\boldsymbol{R}$ is then inverted using a recurrence algorithm presented in chapter 2. The inverted triangular matrix $\boldsymbol{R}^{-1}$ is then multiplied with the matrix $\boldsymbol{Q}$ to form A^{-1}. A block scheme of the computation steps is shown in figure 4.1.

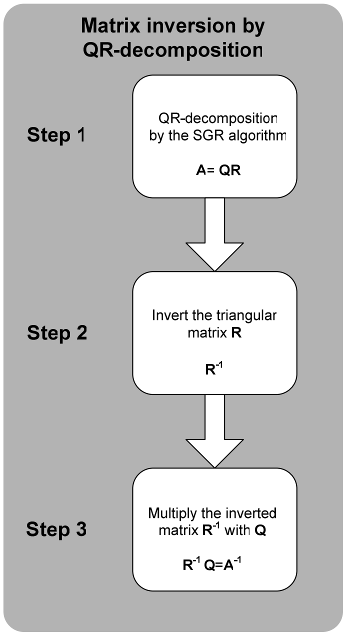

Matrix inversion by QR-decomposition
Step 1
QR-decomposition by the SGR algorithm
A= QR
Step 2
Invert the triangular matrix R
R⁻¹
Step 3
Multiply the inverted matrix R⁻¹ with Q
R⁻¹ Q=A⁻¹

Figure 4.1 Computation scheme of matrix inversion by QR-decomposition.

4.4 Matrix inversion architectures

Three different architectures of the triangular inversion and the QR-decompositions have been presented in chapter 2 and 3. These three architectures can be combined together to form three architectures for matrix inversion [47].

4.4.1 Triangular array architecture

The matrix inversion algorithm can be mapped onto two cascaded triangular array architectures as shown in figure 4.2.

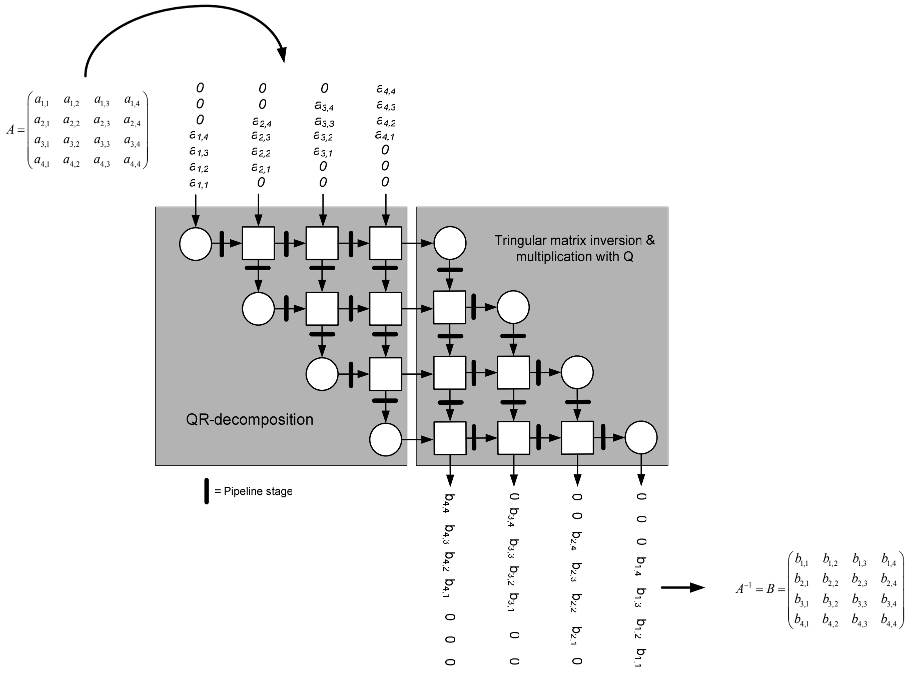


Figure 4.2 Two cascaded triangular array architectures for inversion of a 4-by-4 complex valued matrix *A*.

The architecture in figure 4.2 depicts a matrix inversion architecture for inverting a 4-by-4 complex valued matrix **A**. Each cycle one row of the input matrix ***A*** is feed into the structure in a slightly skewed manner and propagated through the structure. The factorization of ***A*** into the ***R*** and ***Q*** matrix is fully overlapped. The matrices continue without delay into the second triangular architecture which performs and stores the inversion of the upper triangular matrix ***R***. The stored inverted matrix ***R*** is then immediately multiplied with a row of ***Q***, thus producing one output row in the inverted matrix ***A***. With this architecture, a complete inversion of a 4-by-4 complex valued matrix could be performed in 20 cycles under the assumption that a processing element can be executed during one cycle.

The number of processing elements will grow rapidly with the number of inputs. In order to invert a *n*-by-*n* matrix, *n(n+1)* processing elements are required. This will results in a large consumption of gates in an FPGA even for small matrix sizes. Using this architecture in hardware implementations is not reasonable. A more suitable architecture is needed.

4.4.2 Linear array architecture

An alternative architecture based on the linear arrays presented in chapter 2 and 3 is shown in figure 4.3.

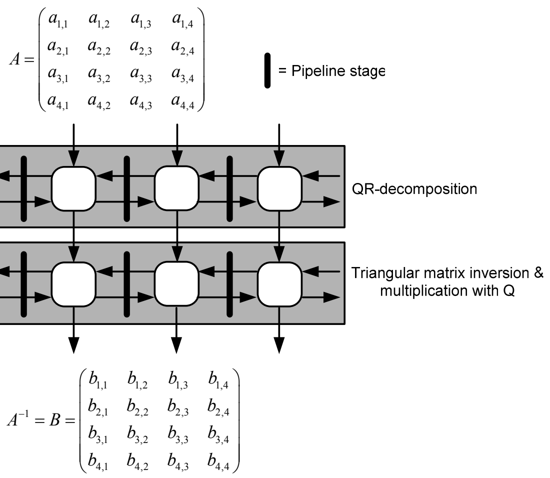

$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{pmatrix}$
= Pipeline stage
QR-decomposition
Triangular matrix inversion & multiplication with Q
$A^{-1} = B = \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} \\ b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} \\ b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} \\ b_{4,1} & b_{4,2} & b_{4,3} & b_{4,4} \end{pmatrix}$

Figure 4.3 Two cascaded linear array architectures for inversion of a 4-by-4 complex valued matrix ***A***.

As discussed before the linear array architecture avoids many of the problems associated with the triangular array architecture [49]. The two architectures are stacked on top of each other. The results from the QR-decomposition flow directly into the triangular matrix inversion architecture. The inverted triangular matrix is stored internally in the array and multiplied with the ***Q*** matrix before producing the inverted matrix $\boldsymbol{A}^{-1}$.

In the linear array architecture the inversion of an *n*-by-*n* matrix only requires 2*n* PEs, compared to *n(n+1)* PEs in the triangular array architecture, which will result in a huge savings of hardware even for small matrix sizes.

4.4.3 Single element architecture

A third architecture is the single element architecture [50]. As shown in figure 4.4, the function of a single pair of processing elements in the linear array structure is mapped onto two single element architectures. However, as shown in chapter 2 and 3 the SPE architectures are virtually the same, except for the memory sizes. Therefore, the two SPE architectures can be combined into one SPE architecture.

The combined SPE architecture begins with performing the QR-decomposition. During the computation the ***Q*** matrix is stored in a separate memory as discussed in chapter 3. Later in the process, when performing inversion of the triangular matrix ***R*** the computed values of ***R*** is directly multiplied with the ***Q*** matrix stored in the memory. This eliminates the process of storing both matrices.

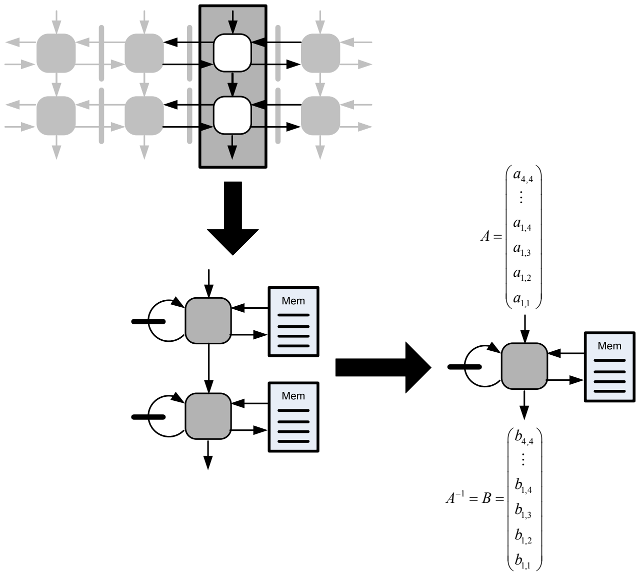

Mem
Mem
Mem
$A = \begin{pmatrix} a_{4,4} \\ \vdots \\ a_{1,4} \\ a_{1,3} \\ a_{1,2} \\ a_{1,1} \end{pmatrix}$
$A^{-1} = B = \begin{pmatrix} b_{4,4} \\ \vdots \\ b_{1,4} \\ b_{1,3} \\ b_{1,2} \\ b_{1,1} \end{pmatrix}$

Figure 4.4 Mapping of a section of the linear array onto a single processing element with a memory unit.

4.4.4 Comparison between architectures

In table 4.1 a comparison between the three different architectures are shown. The different architectures have there own pros and cons. If resource consumption is of concern the single processing element architecture should be chosen for implementation. The most versatile architecture is the linear array which combines low area consumption with high scalability. The triangular array architecture is not reasonable to use in hardware implementation since it consumes a large amount of resources if implemented.

Table 4.1 Comparison between the matrix inversion architectures.

Architecture	# of PEs	# cycles for inversion	Scalability
Triangular	$n(n+1)$	$5n$	low
Linear	$2n$	$3n^2+n-1$	high
Single element	1^*	$3n^2(n+1)/2$	medium

* The memory needed to store matrix data grows with the matrix size.

4.5 FPGA implementations of matrix inversion

The complex valued matrix inversion architectures have been implemented in a Xilinx Virtex II XC2V1000 FPGA with speed grade 4. The same rounding scheme and saturation logic was used as for the processing elements in chapter 2 and 3. Two complex valued matrix inversion architectures were implemented. The first implementation is based on the linear array architecture and the second is based on the single processing element architecture.

4.5.1 Implementation of linear array architecture

Table 4.2 shows the amount of resources needed to implement matrix inversion based on the linear array architecture. The table shows that the implementation of the matrix division architecture for inversion of a 4-by-4 complex valued matrix consumes 2948 slices or 57% of the resources in the FPGA. The maximum clocking frequency is 100MHz.

Table 4.2 Consumed resources of the complex valued matrix inversion architecture based on the linear array architecture.

	Complex matrix inversion Linear Array
Number of Slices	2948 (57%)
Number of Slice Flip Flops	5128 (50%)
Number of 4 input LUTs	5146 (50%)
Number of BRAMs	8 (20%)
Maximum frequency (MHz)	100

Figure 4.5 show the routed floorplan of the complex valued matrix inversion implemented using the linear array architecture.

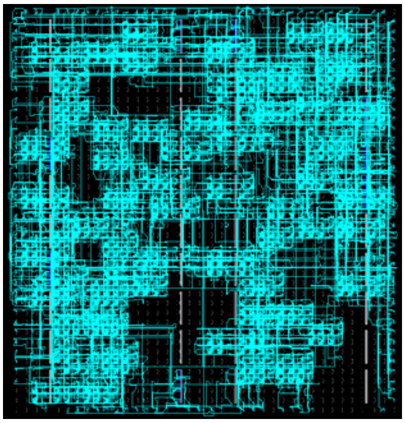


Figure 4.5 Routed floorplan of a complex valued matrix inversion implemented using the linear array architecture.

4.5.2 Implementation of single processing element architecture

Figure 4.6 shows a block diagram of the combined single element architecture. This design has two memories banks. One of the memories stores intermediate computations while the other stores the ***Q*** matrix which later in the process is multiplied with $\boldsymbol{R}^{-1}$.

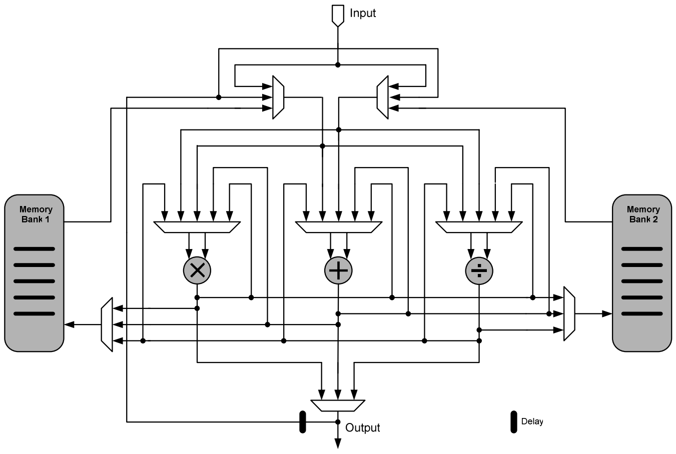


Figure 4.6 Routed floorplan of a complex valued matrix inversion implemented using the linear array architecture.

Table 4.3 shows the amount of resources needed to implement matrix inversion architecture for complex valued matrices using the SPE architecture. The SPE consumes only 10% of the total amount of resources in the design and the maximum clocking frequency is 102 MHz.

Table 4.3 Consumed resources of the complex valued matrix inversion.

	SPE Complex valued design
Number of Slices	512 (10%)
Number of Slice Flip Flops	923 (7%)
Number of 4 input LUTs	902 (7%)
Number of BRAMs	4 (10%)
Maximum frequency (MHz)	102

Figure 4.7 shows the routed floorplan of a complex valued single processing element able to perform a QR-decomposition on a complex valued matrix.

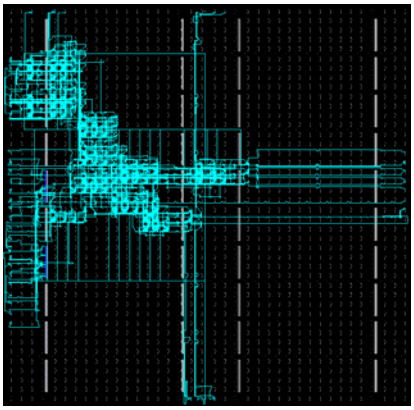


Figure 4.7 Routed floorplan of a complex valued matrix inversion implemented using the single processing element architecture.

4.5.3 Summary of the hardware implementations

The complex valued linear array architecture was implemented using 2948 slices while the SPE architecture only needed 512 slices. The difference between the linear array architecture and the SPE architecture is becoming very large. The SPE architecture uses only 17% of the slices needed to implement the linear array architecture. However, the SPE architecture needs $O(n^3)$ cycles to complete a QR-decomposition, while the linear array architecture needs $O(n^2)$ cycles. With a linear growth of the size of the linear array architecture an 8-by-8 complex valued matrix should consume 100% of the used FPGAs resources. If larger matrices would to be inverted the SPE architecture should be used or a larger FPGA is needed.

Chapter 5

Singular value decomposition

5.1 Introduction

Singular value decomposition (SVD) is an important factorization technique of a rectangular real or complex matrix, with several applications in digital signal processing, beamforming, adaptive sensor array processing, wireless communications, computational tomography, image processing, seismology, etc. [8],[36].

The SVD is a computationally demanding operation, which often must be performed in real-time, with high numerical accuracy. Since matrix sizes and types (real and complex valued) vary between applications, a flexible and scalable hardware architecture that can easily be adapted to different scenarios is highly sought for. To be able to comply with these requirements, special-purpose architectures must be developed.

5.2 Definition of SVD

Let A be a matrix of size $n \times n$. The singular value decomposition of the matrix $A \in \mathbb{R}^{n \times n}$ is then given by the factorization of the matrix into a product of the three matrices

$$A = U \Sigma V^H \tag{5.1}$$

where $U \in \mathbb{R}^{n \times n}$ is a matrix with orthonormal columns, $V \in \mathbb{R}^{n \times n}$ is an orthogonal matrix, and $\Sigma \in \mathbb{R}^{n \times n}$ is a matrix with non-negative diagonal elements, $\Sigma = diag(\sigma_1, \sigma_2, ..., \sigma_n)$. The numbers σ_i, $i = 1 \ldots n$, are the i^{th} singular value of the matrix and v_i, i=1…n, and u_i, i=1…n, are the right and left singular vectors corresponding to σ_i [14].

5.3 SVD algorithm

There are a number of numerically stable algorithms for computing the SVD of a matrix. The two most commonly used classes of algorithms are QR based or Jacobi rotations based [14]. The QR based algorithms are mostly used in sequential implementations since they are generally faster than the Jacobi-based algorithms. The Jacobi-based algorithms are generally more stable and accurate than the QR-based algorithms. Recently the Jacobi-based algorithms have attracted lot of attention due to the fact that they have a greater potential of being massively parallelized [52]. There are two main types of Jacobi algorithms, single-sided and two-sided. In this paper a modified version of the standard two-sided Jacobi algorithm is used.

5.3.1 Jacobi algorithm

The Jacobi method exploits the relationship

$$A = U\Sigma V^H \Leftrightarrow \quad U^T AV = \Sigma \tag{5.2}$$

to produce the matrices U and V by performing a series of two sided plane rotations of the input matrix as shown by equation 5.3.

$$A_{i+1} = J_i^{l^T} A_i J_i^r \tag{5.3}$$

The two matrices J_i^l and J_i^r are referred to as the Jacobi rotation matrices. After each iteration, the matrix A_{i+1} gets more diagonal than the previous one A_i. After n iterations the matrix A is transformed into a diagonal matrix A_n where

$$A_n = \Sigma \text{ and } V = \prod J_i^l \text{ and } U = \prod J_i^r. \tag{5.4}$$

The rotations are carried out in the p,q plane by rotating θ degrees. The rotation matrices $J(p,q,\theta)$ take the form

$$\begin{array}{cc} \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} & \begin{array}{c} \\ \\ p \\ \\ q \\ \\ \\ \end{array} \\ \begin{array}{ccccccc} \quad & & p & & q & & \quad \end{array} & \end{array} \quad \text{where } J_{ij} = \begin{cases} (p > q) \\ J_{pp} = \cos\theta \\ J_{pp} = \cos\theta \\ J_{pq} = \sin\theta \\ J_{qp} = -\sin\theta \end{cases} \tag{5.5}$$

The Jacobi algorithm divides the $n \times n$ symmetric matrix in (5.5) into $n/2$ sub-problems indexed by (*p,q*). The basic step is then to find the angles *θ*, which solves the 2×2 SVD sub-problem shown in (5.6).

$$\begin{pmatrix} a'_{pp} & 0 \\ 0 & a'_{qq} \end{pmatrix} = \begin{pmatrix} \cos\theta_l & \sin\theta_l \\ -\sin\theta_l & \cos\theta_l \end{pmatrix}^T \begin{pmatrix} a_{pp} & a_{pq} \\ a_{qp} & a_{qq} \end{pmatrix} \begin{pmatrix} \cos\theta_r & \sin\theta_r \\ -\sin\theta_r & \cos\theta_r \end{pmatrix} \tag{5.6}$$

An *n=8* matrix can subsequently be divided into the following $n/2$ sets of pairs

$$(p,q) = \begin{Bmatrix} (1,2) & (1,4) & (1,6) & (1,8) & (1,7) & (1,5) & (1,3) \\ (3,4) & (2,6) & (4,8) & (6,7) & (8,5) & (7,3) & (5,2) \\ (5,6) & (3,8) & (2,7) & (4,5) & (6,3) & (8,2) & (7,4) \\ (7,8) & (5,7) & (3,5) & (2,3) & (4,2) & (6,4) & (8,6) \end{Bmatrix} \tag{5.7}$$

where each row represents a set of isolated sub-problems which can be computed in parallel [14]. The Jacobi algorithm then performs 2×2 SVD for all possible pairs (*p,q*), which is referred to as a sweep. This process is then repeated for as many sweeps as necessary to form the singular values.

5.3.2 Jacobi for complex matrices

By a series of transforms, the Jacobi algorithm for real matrices can be extended to complex valued arithmetic [14]. A unitary 2×2 complex matrix may be expressed as

$$A \triangleq \begin{bmatrix} a_x + ia_y & b_x + ib_y \\ c_x + ic_y & d_x + id_y \end{bmatrix} = \begin{bmatrix} c_\phi e^{i\theta_\alpha} & s_\phi e^{i\theta_\beta} \\ -s_\phi e^{i\theta_\gamma} & c_\phi e^{i\theta_\delta} \end{bmatrix} \tag{5.8}$$

where $\theta_\alpha, \theta_\beta, \theta_\delta, \theta_\gamma \in \mathbb{R}$. An arbitrary complex 2×2 matrix can not be always be diagonalized in only one transformation. However, two transformations are sufficient for computing the SVD. By Givens rotations, the first transformation renders the bottom row of the matrix real and the lower left element zero. The first transformation can be expressed as

$$\begin{bmatrix} c_\phi e^{i\theta_\alpha} & -s_\phi e^{i\theta_\beta} \\ s_\phi e^{i\theta_\alpha} & c_\phi e^{i\theta_\beta} \end{bmatrix} \begin{bmatrix} Ae^{i\theta_a} & Be^{i\theta_b} \\ Ce^{i\theta_c} & De^{i\theta_d} \end{bmatrix} \begin{bmatrix} c_\psi e^{i\theta_\gamma} & s_\psi e^{i\theta_\gamma} \\ -s_\psi e^{i\theta_\delta} & c_\psi e^{i\theta_\delta} \end{bmatrix} = \begin{bmatrix} We^{i\theta_w} & Xe^{i\theta_x} \\ 0 & Z \end{bmatrix} \tag{5.9}$$

where $\theta_\alpha = \theta_\beta = -(\theta_d + \theta_c)/2$, $\theta_\gamma = -\theta_\delta = (\theta_d - \theta_c)/2$, $\theta_\psi = \tan^{-1}(C/D)$, and $\theta_\phi = 0$. The second transformation converts the main diagonal elements to real values and converts the upper-right to zero [14]. The second transformation can be expressed as

$$\begin{bmatrix} c_\lambda e^{i\theta_\xi} & -s_\lambda e^{i\theta_\eta} \\ s_\lambda e^{i\theta_\xi} & c_\lambda e^{i\theta_\eta} \end{bmatrix} \begin{bmatrix} We^{i\theta_w} & Xe^{i\theta_x} \\ 0 & Z \end{bmatrix} \begin{bmatrix} c_\rho e^{i\theta_\tau} & s_\rho e^{i\theta_\tau} \\ -s_\rho e^{i\theta_\omega} & c_\rho e^{i\theta_\omega} \end{bmatrix} = \begin{bmatrix} P & 0 \\ 0 & Q \end{bmatrix} \tag{5.10}$$

with $\theta_\xi = -(\theta_x + \theta_w)/2$, $\theta_\eta = (\theta_x - \theta_w)/2$, $\theta_\tau = (\theta_x - \theta_w)/2$, and $\theta_\omega = (\theta_w - \theta_x)/2$. After the two transformations the 2×2 matrix is real and can be diagonalized as discussed in section 3.1 [14].

5.4 SVD architecture

5.4.1 BLV architecture

The most commonly used hardware architecture, for both real and complex valued matrices, is the Brent-Luk-VanLoan systolic array architecture (BLV) [52],[53]. For a $n \times n$ matrix the BVL architecture employs *n/2*×*n/2* locally connected processing elements (PEs) arranged in a square configuration. A BLV with a matrix size of *n=8* is shown in figure 5.1. The horizontal arrows indicate the transmission of the left rotation parameters and the vertical arrows indicates the transmission of the right rotation parameter. The two-way arrows indicate the direction of inter-processor communication between neighboring processors. The BLV architecture solves *n/2* sub-problems in parallel and performs a sweep in *n-1* steps. It takes O(*n*×log *n*) time for the architecture to complete a SVD of a matrix [52],[53].

Initially each processor in the architecture is loaded with a 2×2 sub-matrix (complex or real). The execution of the two transforms of the two-sided Jacobi algorithm is initiated by the diagonal processing elements. In the complex case, each transformation requires the computation and application of twelve angles. Six of the angles are propagated along the rows and columns of the processors on the main diagonal. When the first wave of rotation angles are produced the production of the second is started. The two transforms are propagated down, one behind the other, on either side of the diagonal processing elements. Each off-diagonal processing element applies a two-sided rotation to the local 2×2 sub-matrix using the angles produced by the diagonal processors in the same row and column with respect to its location in the array. After the computation the local matrices are exchanged and a new sweep is initiated.

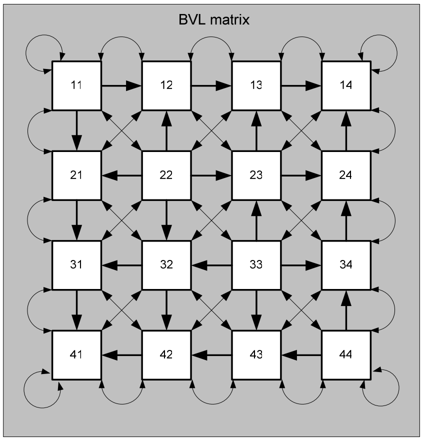


Figure 5.1 A square BVL architecture (n=8) with locally communicating processing elements for solving a SVD.

5.4.2 BVL architectural drawbacks

Unfortunately the BVL architecture in figure 5.1 suffers from serious drawbacks. The number of processing elements will rapidly grow with the number of inputs. In order to invert an *n-by-n* matrix, $(n/2)^2$ processing elements are required. This will result in a large consumption of gates in an FPGA or chip area in an ASIC implementation. A large number of PEs will also result in a long delay time due to the long propagation path through the architecture, which could effectively lower the maximum achievable clock frequency. The architecture also lacks an ease of scalability. For instance, if fewer or more inputs are needed, a completely new architecture must be derived and downloaded into the FPGA or a new ASIC must be manufactured. These drawbacks, combined with the growing need in communications and signal processing applications for scalability, high throughput, and area conservation designs, make the BVL array architecture unsuitable for hardware implementation of large matrix sizes. A compact and scalable array architecture is therefore highly sought for.

5.4.3 Linear BLV architecture

An alternative linear architecture is presented in figure 5.2. For a $n \times n$ matrix the linear architecture employs $n/2+((n/2)-1)$ locally connected processing elements, reducing the number of PEs dramatically for large matrix sizes. The linear array architecture consists of two sets of processing elements, diagonal PEs (DPE) and off-diagonal PEs (PE), arranged in two columns as shown in figure 5.2.

The functionality of the DPEs and PEs is roughly the same as in the BVL architecture with some added registers and flow control circuitry.

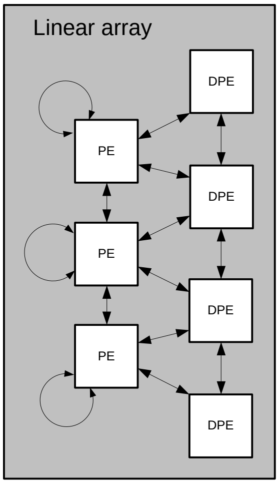


Figure 5.2 A linear array architecture (*n*=8) with locally communicating processing elements for solving a SVD.

The computational procedure is the same as in the BVL case, but instead of propagating the computed rotation angles, produced in the DPEs, down along the off-diagonal PEs, the computations are recycled *n/2-1* times in the PE column. Thus, one sweep in the BVL equals *2*((*n/2*)-*1*) recycles in the linear array. The linear array architecture can easily be scaled by adding a DPE and a PE for each new column in the input matrix. The linear array architecture trades computation speed for area and scalability. The gain is significant for large matrix sizes.

5.5 FPGA implementation

A block diagram of the SVD architecture implemented on a Xilinx FPGA is shown in figure 5.3. The design consists of a central core with a linear array structure capable of performing a 6-by-6 complex valued SVD. Data is input and output in a word serial manner. The memory holds the inputted matrix values and stores intermediate results during computation. The global control unit controls the data flow between the block and handles the I/O.

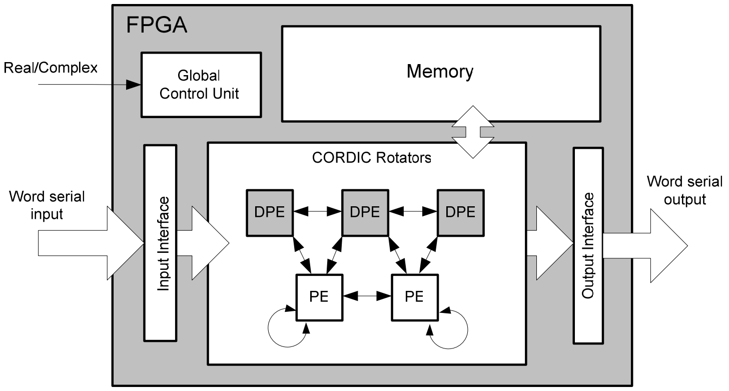


Figure 5.3 Block diagram of a 6-by-6 SVD implemented on an FPGA.

SVD algorithms are known to have high computational complexity and to require expensive arithmetic operations, such as division and square root [33]. The CORDIC building blocks allow such complex operations to be carried out effectively. An overview of the CORDIC approach can be found in part II of the thesis and also here [33],[34],[54].

The processing element is constructed of four CORDIC rotators capable of operating on both complex and real values (figure 5.4). The operation mode is chosen externally before commencing the computation. A good overview of the functionality of a complex CORDIC can be found in Hemkumar [55].

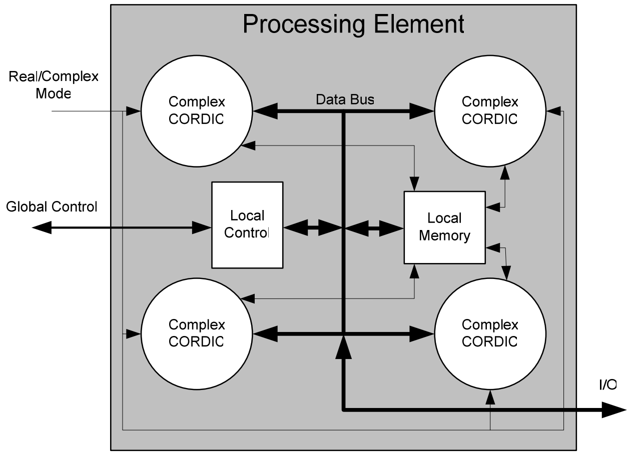


Figure 5.4 Block diagram of the processing elements consisting of four real/complex CORDIC units.

The four CORDIC rotators perform the necessary rotations on the internally stored 2×2 sub-matrix as described in section 5.3. Intermediate results are stored in the local memory and are accessible for all four CORDIC units. A local control unit handles the internal routing of data and communication between the CORDIC units. The local control unit also handles the I/O and communication with the global control unit. All hardware building blocks were implemented using a wordlength of 16 bits (8 bits for the real part and 8 bits for the imaginary part).

Table 5.1 Consumed resources of the complex valued SVD implementation.

	Complex valued SVD
Number of Slices	4045 (79%)
Number of Slice Flip Flops	9213 (70%)
Number of 4 input LUTs	9278 (70%)
Number of BRAMs	34 (85%)
Maximum frequency (MHz)	92

Table 5.1 shows the consumed resources of the design. The design occupied 4045 slices in the FPGA and was run at 92 MHz. The floorplan of the SVD is shown in figure 5.5.

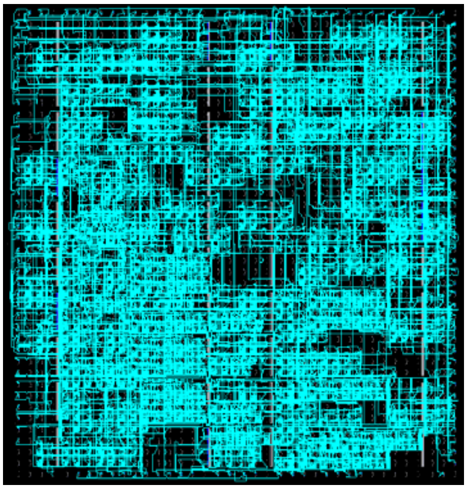


Figure 5.5 Routed floorplan of the SVD architecture implemented in an FPGA.

Part III

Implementation of the Capon beamformer

Chapter 1

Introduction

In this part of the thesis, some of the matrix building blocks developed in part II will be used in an implementation of the Capon beamformer algorithm. In the first chapter, the Capon beamformer algorithm is defined and discussed. The Capon beamformer algorithm can be decomposed into four sequential steps, which are the basis for the two architectures presented in chapter 2. One of the architectures makes use of the complex valued division presented in part II, while the other one uses the single processing element block also presented in part II. In chapter 3, the implementation of a covariance processor is discussed, followed by the presentation of a hardware implementation of the Capon beamformer algorithm. Chapter 4 analyzes the implementation in terms of the impact of reduced wordlength in the different building blocks of the architecture, and a minimum wordlength architecture is derived. In chapter 5, the implementation is put to the test in a real application of a channel sounder.

1.1 Capon's beamformer method

The minimum variance method (MVM), also known as Capon's beamformer method, is a well-known spectral-based algorithm for direction of arrival (DOA) estimation [6],[7]. In the early years of DOA estimation, a general approach based on the Fourier transform, which later became known as conventional beamforming, was given much attention. The advantage of this method is that it is very easy to implement, since only a fast Fourier transform is needed. However, one of the main disadvantages of conventional beamforming is its inability to resolve two sources that are positioned close together. This weakness is a consequence of the fact that the resolution of the conventional beamformer is limited by the size of its antenna array.

This limitation gave rise to a new class of high-resolution methods. In these methods the resolution is not limited by the size of the antenna array, but they are known to be sensitive to modeling errors and noise. In particular, they are based on precise knowledge of the array in terms of its sensor location, sensor gain and phase, mutual coupling between array elements etc. If applied with incorrect array parameters, these methods give poor results. Thus, repeated calibration with respect to array parameters is a prerequisite for these techniques. Another drawback is that the methods are computationally heavy and therefore expensive in terms of clock cycles and/or hardware resources.

Well-known high-resolution methods include ESPRIT, MUSIC, SAGE, and Capon [6]. Capon's beamformer is a spectral-based search method, which determines an angular spectrum for each so-called look direction by minimizing the power contribution by noise and the interference from other directions than the look direction.

1.2 Signal model

Before the Capon beamformer method is analyzed, a mathematical model for the array and incoming signals is needed. Consider a uniformed linear array (ULA) with *L* equally spaced antenna elements as shown in figure 1.1.

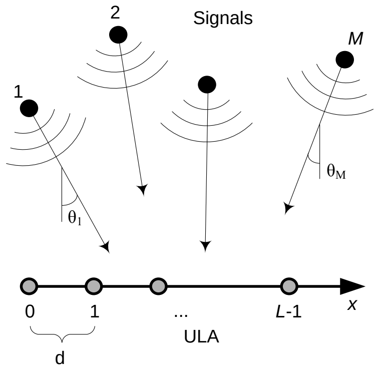


Figure 1.1 *M* signal sources impinging on a uniformed linear array antenna (top view) with *L* antenna elements.

If *M* signals impinge on the *L*-dimensional ULA at distinct DOAs, $\theta_1,\ldots,\theta_M$, the array output vector can be expressed as

$$\boldsymbol{x}(t) = \sum_{m=1}^{M} \boldsymbol{a}(\theta_m) s_m(t)\,, \tag{1.1}$$

where $s_m(t),\ m=1,\ldots,M$ denotes the baseband signal waveforms and $\boldsymbol{a}(\theta_m)$ is the steering vector of a signal at the DOA θ_m. The steering vector is expressed as

$$\boldsymbol{a}(\theta) = \begin{bmatrix} 1 & e^{-j\phi} & \cdots & e^{-j(L-1)\phi} \end{bmatrix}^T, \quad \phi = kd\cos\theta \tag{1.2}$$

where *d* denotes the distance between the elements and *k* is the wave-vector. A steering vector characterizes the relative phase response of each antenna array element to an incident signal with DOA θ. The steering vector is determined by calibration of the antenna array. Assume that N snapshots of the array output are recorded. The output vector including additive noise can then be written (in a compact form) as

$$\boldsymbol{X}(t) = \boldsymbol{A}(\theta)s(t) + \boldsymbol{n}(t), \quad t=1, 2, \ldots, N. \tag{1.3}$$

1.3 Capon's approach

The idea of Capon's beamformer is to localize signal sources by steering the array in one direction at a time and measuring the output power. The locations with the maximum power yield the DOA estimates. The array response is steered by forming linear combinations of the outputs

$$y(t) = \sum_{\mathrm{i=1}}^{\mathrm{L}} \boldsymbol{w}_i x_i(t) = \boldsymbol{w}^H \boldsymbol{x}(t). \tag{1.4}$$

where $\boldsymbol{w}^H$ is the Hermitian transpose of the weighting vector. Given the samples *y(1), y(2), ...,y(N)* the output power can be described by

$$P(\boldsymbol{w}) = \frac{1}{\mathrm{N}}\sum_{\mathrm{m=1}}^{\mathrm{N}} |\mathrm{y(t)}|^2 = \frac{1}{\mathrm{N}}\sum_{\mathrm{m=1}}^{\mathrm{N}} \boldsymbol{w}^H x(t) x^H(t) \boldsymbol{w} = \boldsymbol{w}^H \hat{\boldsymbol{R}} \boldsymbol{w} \tag{1.5}$$

where $\hat{\boldsymbol{R}} = \boldsymbol{X}\boldsymbol{X}^H$ is the sample covariance matrix, which replaces the covariance matrix $\boldsymbol{R}$**,** since in practice only the sample estimates are available. The Capon optimization problem [6] was formulated as

$$\min_{\boldsymbol{w}} P(\boldsymbol{w}) \quad \text{subject to} \quad \boldsymbol{w^H a}(\theta) = 1. \tag{1.6}$$

The optimal weighting vector, ***w,*** can be found using Lagrange multipliers [6]] and results in

$$\boldsymbol{w}_{\boldsymbol{Cap}} = \frac{\hat{\boldsymbol{R}}^{\boldsymbol{-1}}\boldsymbol{a(\theta)}}{\boldsymbol{a^H(\theta)\hat{R}^{-1}a(\theta)}}. \tag{1.7}$$

Inserting the optimal weight $\boldsymbol{w}_{Cap}$ into (1.5) gives the Capon power spectral estimator for a given direction θ

$$P_{Capon}(\theta) = \frac{1}{\boldsymbol{a^H(\theta)\hat{R}^{-1}a(\theta)}}. \tag{1.8}$$

If M signals are present in the collected data, then θ_m, $m=1,\ldots,M,$ scan values are obtained as the argument of the M largest peaks of $P_{\mathrm{Cap}}(\theta)$. The Capon power estimation equation will in this thesis be referred to as the Capon algorithm.

Chapter 2

Capon architectures

In this chapter, two architectures implementing the Capon algorithm are presented. The chapter starts with an analysis of the Capon algorithm from a hardware computational perspective. The algorithm is decomposed into four steps that are executed in a sequence. The decomposition makes it possible to identify which building blocks are needed to implement each step. Finally, the two architectures implementing the Capon algorithm are presented.

2.1 Algorithm decomposition

The Capon algorithm presented in equation 1.8 can be decomposed into four steps as shown in figure 2.1. The steps are executed in sequence, describing the computational order of the algorithm.

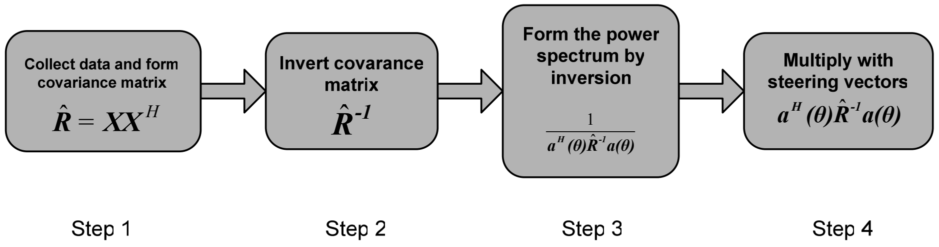

Collect data and form covariance matrix
$\hat{R} = XX^H$
Invert covarance matrix
$\hat{R}^{-1}$
Form the power spectrum by inversion
$\frac{1}{a^H(\theta)\hat{R}^{-1}a(\theta)}$
Multiply with steering vectors
$a^H(\theta)\hat{R}^{-1}a(\theta)$
Step 1
Step 2
Step 3
Step 4

Figure 2.1 Computational scheme of the Capon beamforming algorithm.

The first step in the estimation of DOAs is the forming of a covariance matrix from the sampled signal data. The computation of the a sample covariance matrix can be expressed as

$$\hat{\boldsymbol{R}} = \boldsymbol{X}\boldsymbol{X}^H = \frac{1}{N}\sum_{m=1}^{N} \boldsymbol{x}_m \boldsymbol{x}_m^* \ . \tag{2.1}$$

One covariance matrix is produced from each snapshot of the array output data. High-resolution algorithms such as the Capon algorithm are very sensitive to ill-conditioned (near singular) covariance matrices. This can happen in situations when the received signals, coming from different directions, all originate from the same source. The covariance data will then be correlated making it impossible to invert in the next computation step. This problem could for example arise in applications such as channel sounding.

There are several ways of reducing the sensitivity of the covariance matrix. One way is to average over several covariance matrices, hoping that subtle changes of the channel will reduce the sensitivity of the covariance matrix. Another technique is to induce an “error” by adding (or subtracting) a small value (1 LSB) into the diagonal elements of the covariance matrix before inversion. This technique can be expressed in matrix format as

$$\hat{\boldsymbol{R}}_{cov} = \hat{\boldsymbol{R}}_{avrage} + \delta \boldsymbol{I} \tag{2.2}$$

where δ is a small real value (error) and $\boldsymbol{I}$ is the identity matrix. The cost of implementing equation (2.2) in hardware is an adder (subtractor) and will be discussed in chapter 3. There are also other techniques involving subarray averaging (spatial smoothing) [58],[59]. However, these techniques will cost more to implement in hardware than the technique in equation (2.2) and will also decrease the effective size of the antenna array.

In the second step, the complex valued sample covariance matrix is inverted. The inversion can be performed in a number of ways, for example by QR-decomposition as shown in equation (2.3).

$$\hat{\boldsymbol{R}}_{cov} = \boldsymbol{Q} \times \boldsymbol{R}_{\Delta} \rightarrow \boldsymbol{R}_{\Delta}^{-1} \times \boldsymbol{Q} = \hat{\boldsymbol{R}}_{cov}^{-1} \tag{2.3}$$

In the third step, the inverted covariance matrix is multiplied with the complex conjugate of the steering vector, $\boldsymbol{a}^{H}(\theta)$, and the steering vector, $\boldsymbol{a}(\theta)$. This is done for every angle θ, going from 0° to 180° degrees in 1° steps. The operation is a vector-matrix-vector operation, which in practice can be performed as a number of parallel vector-vector operations.

In the fourth step, the computed scalar values of $\boldsymbol{a}^{H}(\theta)\hat{\boldsymbol{R}}^{-1}\boldsymbol{a}(\theta)$ are inverted. The scalar value of the Capon power spectral estimation, $P(\theta)$, and the corresponding angle θ are produced.

2.2 Architectures

From the computation scheme described in section 2.1, two architectures were derived. The first architecture is based on the sequential data flow and consists of four hardware blocks, one for each step, performing the described computations. In this thesis the architecture is referred to as the linear data flow architecture (LDF) [60]. The second architecture is based on the single processing element architecture described in part II chapter 4. In this thesis the single processing element architecture is referred to as the SPE architecture [61].

2.2.1 Linear data flow architecture

The LDF architecture is derived from the computation scheme in figure 2.1 and is shown in figure 2.3. The sample data coming from the antenna array is first stored in a buffer before sent to a covariance processor computing the covariance matrix. The computation involves a multiplication of an element vector with its complex conjugate transpose, producing a matrix for each set of data. The values over and under the diagonal of the covariance matrix are mirror images, and the number of computations can therefore be reduced, since only the upper or the lower part needs to be computed.

An architecture for implementing the covariance processor is presented in figure 2.2. The architecture consists of *k* covariance processing elements (CovPE), one for each input vector of sample data operating on the sample data vectors stored in the buffer.

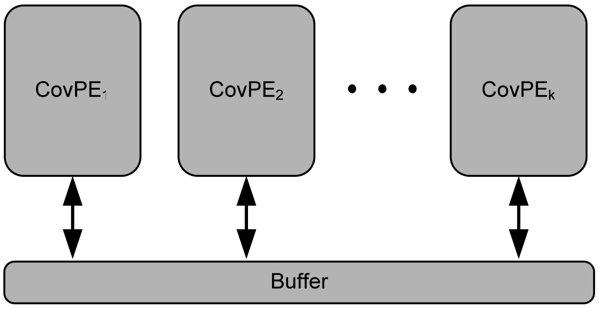


Figure 2.2 Covariance architectures with *k* covariance processors communicating with local memories.

The stored sample data is read and processed through the CovPEs, producing the matrix elements in the covariance matrix. When the first snapshot of data has been processed, the next snapshot is read into the buffer and the computational procedure starts over. The covariance matrix produced by the new sample data is added to the previous covariance matrix, forming the averaged covariance matrix. This iterative procedure is repeated for the number of snapshots set by the user (in the range 1-4096 iterations). When the averaged covariance matrix is produced, a small error is induced in the diagonal elements by adding one bit to the least significant bit of the diagonal element. The covariance processor architecture in figure 2.2 is easy to scale by adding one CovPE for every input vector (antenna element).

The resulting covariance matrix is sent one row at the time to the matrix inversion block. The matrix inversion is done by using the complex valued matrix inversion architecture presented in part II chapter 4.

The inverted matrix is sent to the steering vector block, where it is stored in a buffer before being processed in two steps. To process the multiplication of the steering vectors with the inverted covariance matrix, a similar architecture as the one used in the covariance processor is adopted. The steering vector process consists of $k+1$ StPE for processing of $k+1$ vectors at a time. The steering vector is either stored in a ROM or has been read from an external source and stored in a RAM. The vector $\boldsymbol{a}^{H}(\theta)$, together with the columns of the inverted matrix, is fed into the processing elements producing a vector. The resulting vector is then fed into another processing element together with $\boldsymbol{a}(\theta)$, producing a single value for a given direction θ. In continuous run the two steps are performed simultaneously producing one output value for every 181 angles. The resulting values are sent to the inversion block, where the power estimation values are produced, along with the corresponding angle, using an inverter consisting of two special Booth multipliers and a lookup table (appendix A).

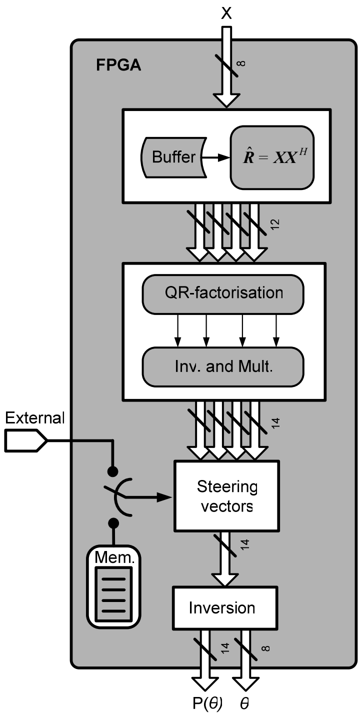


Figure 2.3 Block diagram of the linear data flow architecture of the Capon beamforming algorithm.

The power estimation values for each angle can be plotted on a display showing a power spectral estimation diagram with peaks indicating the DOAs. If the DOAs in particular are interesting, a search module must be added to the output to collect the power estimation values and search for the highest peaks in the data. The search function was not implemented in this thesis.

2.2.2 Single processing element architecture

The single processing element architecture (SPE) implementing the Capon algorithm is presented in figure 2.4. In this architecture the sample data coming from the antenna array is stored in a global memory. The covariance processor computes the covariance sample matrix as described in section 2.1, with the difference that it is operating and storing results directly in the global memory. The matrix inversion is performed by the SPE as described in part II chapter 4. The SPE loads the stored values of the covariance matrix one by one and performs the necessary operations, and then writes the result back into the global memory. Since the steering vector processor and the covariance processor perform the same operations, they are combined into one unit. The combined unit reads the inverted matrix from the memory and the stored steering vectors and performs the necessary vector-vector operations. The inversion computation can also be performed by the SPE architecture. The resulting value from the steering vectors is read by the SPE from the global memory and inverted, forming the power estimation values for each angle. The power estimation values along with the corresponding angles are output from the FPGA.

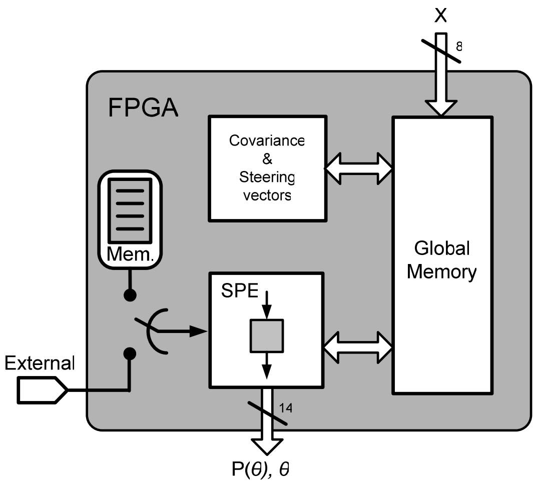


Figure 2.4 Block diagram of the Capon beamforming algorithm based on the single processing element architecture.

Chapter 3

FPGA implementation

In this chapter, an FPGA implementation of the Capon algorithm based on the linear data flow architecture (LDF) is presented. The chapter starts with a discussion about the hardware implementation of the covariance processing element used in the LDF architecture. This is followed by a look at the hardware implementation of the LDF architecture and the various design parameters. The chapter ends with a discussion about the scalability of the design.

The prerequisites of the hardware implementation are set by the target application, which in this case is to operate in a channel sounder testbed. The hardware design was adapted for a uniformed linear antenna array with four antenna elements. Sampled input data, consisting of 120 samples in four vectors, is represented in fixed-point with 8 bits for the real part and 8 bits for the imaginary part. The number of snapshots was set to 120. The hardware implementation of the Capon beamformer will operate in real-time, computing the DOAs of the impinging signals.

The target FPGA was a Xilinx Virtex II XC2V1000 (speed grade 4) clocked at 100 MHz. With the chosen FPGA the hardware resources are limited to 5120 slices. All building blocks implementing the Capon algorithm are scalable.

3.1 Capon building blocks

The linear data flow architecture presented in chapter 2 consists of four major building blocks; covariance processor, complex valued division, steering vector, and inversion. The complex valued division and the inversion blocks are presented in detail in part II chapter 4 of this thesis, while the covariance processor and the steering vector block are presented below.

3.1.1 Covariance processor

The operations required when computing the covariance matrix are basically complex multiplication and addition/subtraction, as shown in equation (2.1). The covariance processor consists of four covariance processing elements (CovPE), one for each input vector, operating in parallel. A block diagram of a covariance processing element is shown in figure 3.1. Each covariance processor performs a complex valued multiplication between the complex valued vector element and its complex valued conjugate. A strength reduction scheme is used, reducing the number of multiplications needed by trading one multiplication for 3 additions, as shown in equation 3.1.

$$\begin{aligned} \Re &= \left(xz - yw\right) = xz - yw + xw - xw = x(z-w) + w(x-y) \\ \Im &= \left(xw + yz\right) = xw + yz + yw - yw = y(z+w) + w(x-y) \end{aligned} \tag{3.1}$$

The result from the complex valued multiplication is collected in an accumulator. When all the values in the two vectors processed by the CovPE are computed, the accumulated value is normalized by multiplying it with a preset constant *1/N*, where *N* is the length of the vector. The normalized result is written back into the memory. If the computed value is a diagonal value, 1 LSB is added (subtracted) before written back into the memory (not shown in the figure). The operation will cost either an adder or a subtractor. Another way of inducing the error in the diagonal elements is to invert the LSB of the diagonal elements. In this way not all diagonal elements will be affected equal since in some diagonal elements 1 LSB is added and in some 1 LSB is subtracted. The effect, if any, is not yet fully determined but initial simulations shows no adverse effect. This technique is cheaper to implement than an adder (subtractor).

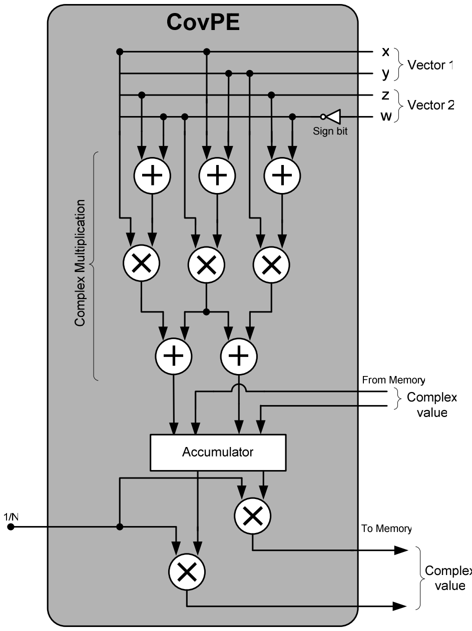


Figure 3.1 Block diagram of a covariance processing element.

All arithmetic blocks used in the CovPE design are Xilinx optimized building blocks. Table 3.1 shows the amount of consumed resources of a covariance processing element.

Table 3.1 Consumed resources of a covariance processing element.

	CovPE
Number of Slices	168 (3%)
Number of Slice Flip Flops	260 (2%)
Number of 4 input LUTs	298 (2%)
Maximum frequency (MHz)	130

One covariance processing element occupies only 168 slices in the FPGA and it can be clocked at 130 MHz. Figure 3.2 shows that the design has been placed and routed in a cluster, which will minimize long interconnections that would slow down the design.

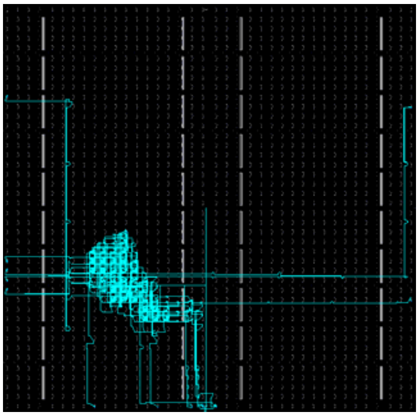


Figure 3.2 Routed floorplan of a covariance processing element.

3.1.2 Steering vector processor

The values in the steering vector are either programmed into a memory on the FPGA or loaded from an external source and stored in a RAM on the FPGA. The vector-matrix-vector multiplication can be broken down into four vector-vector multiplications, followed by a single vector-vector multiplication. The steering vector processor consists of five steering vector processing elements (StPE) and a memory. Figure 3.3 shows a block diagram of a StPE that performs a vector-vector multiplication and accumulation.

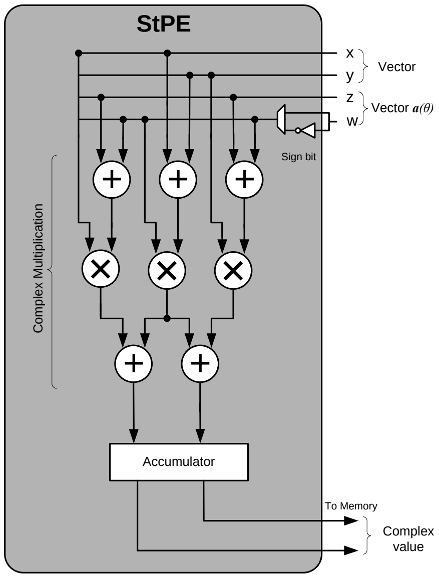


Figure 3.3 Block diagram of a steering vector processing element.

The steering vector and the vectors in the matrix are processed through the four StPEs, and the resulting vector is processed through a fifth StPE where it is multiplied with the conjugate of the steering vector, thus producing an output value. The sign bit inverter is used to convert the steering vector to its complex conjugate when needed.

All arithmetic blocks used in the StPE design are Xilinx optimized building blocks. Table 3.2 shows the amount of consumed resources of a StPE. The CovPE occupies 34 slices more than the StPE, since it also includes two multiplications and some control circuitry.

Table 3.2 Consumed resources of a covariance processing element.

	StPE
Number of Slices	134 (3%)
Number of Slice Flip Flops	175 (2%)
Number of 4 input LUTs	193 (2%)
Maximum frequency (MHz)	132

One covariance processing element occupies only 134 slices in the FPGA and it can be clocked at 132 MHz. Figure 3.4 shows that the design has been placed and routed in a cluster, which will minimize long interconnections that would slow down the design.

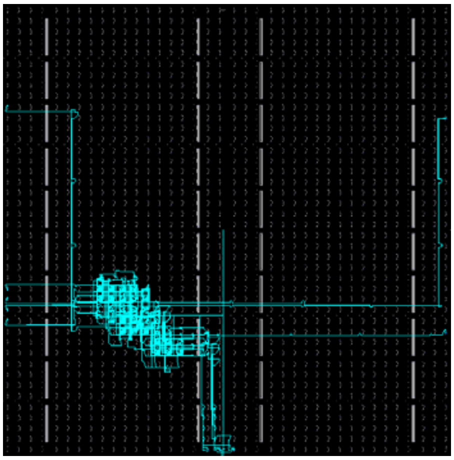


Figure 3.4 Routed floorplan of a steering vector processing element.

3.2 FPGA implementation of the LDF architecture

The linear data flow architecture performing the Capon algorithm was implemented in an FPGA [60]. The complex sample values coming from the antenna array are processed through the covariance processor and stored in a temporary buffer. This is repeated for as many times (1-4096) as decided by a 12-bit snapshot counter, which is set by the user. This is done to get an uncorrelated covariance matrix.

The implementation was clocked at 100MHz and consumed 4762 slices, which accounts for 93% of the FPGA resources. The sample covariance buffer accommodates 120 samples from each of the four antenna elements. Each complex sample, with a wordlength for the real and imaginary part of 8 bits, results in a temporary buffer of 7.7 kbits. Four guard digits are used to ensure numerical stability of the computation. The complex valued matrix inversion uses an additional 2 guard bits. The internal memory used in the computation with the steering vectors accommodates one memory slot for each θ. In total 181 memory slots with four 2*14 bit complex values (one for each antenna element) are used, resulting in a total of 20.3 kbits. An additional buffer (0.3 kbits) for storing the inverted matrix and intermediate computations was used. The computation with the steering vectors results in a real value represented with 14 bits, which is inverted to form $P(\theta)$. The numerical error of the Capon power estimation is in this case less than 1 ulp.

The floorplan of the Capon beamforming implementation is presented in figure 3.5. Nearly all resources in the FPGA are used in the implementation.

Table 3.3 Consumed resources of the different blocks in the design.

	Percentage of resource
Covariance processor	24%
Matrix inversion	62%
Steering vector processor	11%
Inversion	3%

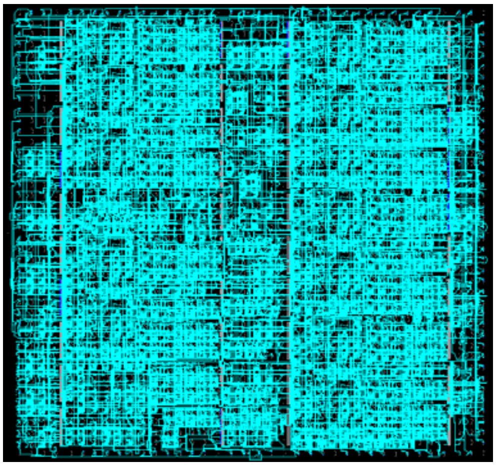


Figure 3.5 Routed floorplan of a Capon beamforming implementation.

3.3 Scalability of the designs

The hardware blocks implementing the linear data flow architecture are all modulized and scalable. This ensures that the design can be scaled if it is going to be used in any other application, or with another antenna configuration, etc. The design is scaled by adding the required number of processing elements.

The building blocks are written in a hardware description language in such a way that it is easy to change the wordlength without redesigning the whole building block. This adds to the scalability of the design. It is also possible to port the design to an ASIC without making to many changes. The only thing that needs to be replaced is the optimized arithmetic building blocks from Xilinx. These building blocks can easily be exchanged for other standard IP cores from an ASIC vendor.

Chapter 4

Capon implementation aspects

This chapter investigates the robustness of the FPGA implementation of the Capon algorithm presented in the previous chapter. To test the implementation, test data based on measured data with added noise from a ULA with four elements was used. The SNR was 10 dB and 120 snapshots of four sample data vectors containing 120 samples were recorded. Figure 4.1 shows a plot of the result with the normalized spectrum on the y-axis and the DOA in degrees on the x-axis.

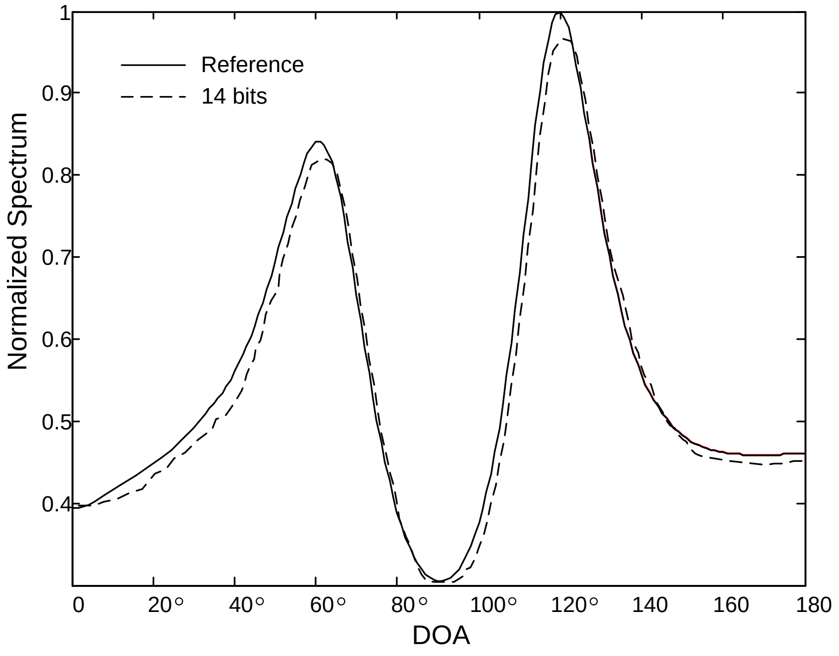


Figure 4.1 Normalized power spectrum of the Capon beamformer implementation plotted for different DOAs.

The reference curve is produced by MATLAB floating point simulations of the same data that was processed through the 14 bit FPGA fixed-point implementation.

The plot shows that the dashed line, originating from the FPGA implementation, follows the reference line and indicates two distinct maxima in the region of 60° and 120°. The deviation from the reference is only 2° for the peak at 60° and 1° for the peak at 120°. To be able to determine if a hardware design is accurate enough, a deviation limit of 5° from the reference was set. Since the deviation of the peaks in figure 4.1 is less than 5°, a 14-bit hardware implementation is sufficiently accurate. The small difference between the dashed and the solid line is due to the limited dynamic range of the hardware implementation. An increase of the number of bits used in the design would probably reduce the difference even more.

The decision of how many guard-bits to use in the design was based on a rough estimation of the needed precision. To determine the actual needed precision a more careful analysis of the each step in the computational scheme is needed. For this purpose, several implementations were constructed where the effects of a reduction of the dynamic range in individual building blocks were investigated. Each step in the computational scheme of the Capon algorithm was tested one at a time in order to determine the impact of the particular step.

4.1 Covariance matrix

In the implementation of the covariance processor presented in chapter 3, four guard digits were added to safeguard from induced numerical errors due to a limited dynamic range. In this test, only the wordlength affecting the computation of the covariance processor was changed, while all other building blocks in the other steps remained unchanged. The dynamic range was gradually reduced by reducing the number of bits used during computations. Figure 4.2 shows a plot of how the result varies with a reduced dynamic range in the building block. With reduced wordlength the peaks shift to the right, inducing an error in the detection of the angle. As stated in the previous section, a maximum deviation of 5° from the reference was tolerated and a deviation over 5° was treated as a failed detection. The shortest wordlength that can be used to compute the covariance matrix in the test case is 6 bits. The 6 bit wordlength was very close to the reference in the diagram but due to the resolution of the plots it was not included here. As seen in figure 4.2, when using a 5 bit wordlength one of the peaks deviate more than 5° from the reference, which is unacceptable. Using a wordlength of only 4 bits will result in a deviation of more than 10° of both peaks. When the wordlength gets even shorter, the detection collapses (not shown in the picture).

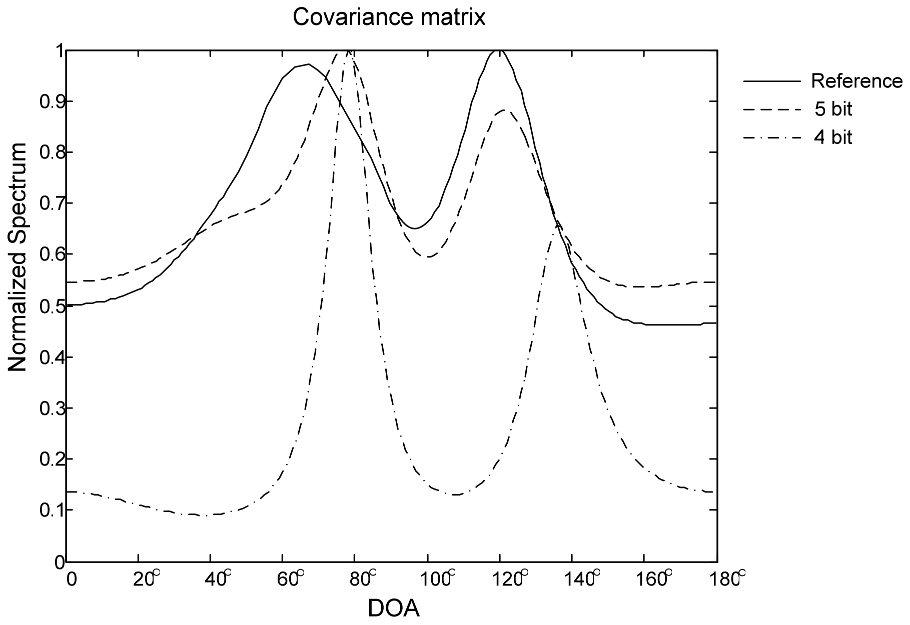


Figure 4.2 Normalized power spectrum for the covariance matrix block with different wordlengths.

4.2 Matrix inversion

The matrix inversion turned out to be more robust than the covariance processor block. When the dynamic range gets smaller the peaks start to drift. As shown in figure 4.3, a wordlength of 5 bits is still inside the allowed boundary of 5° from the reference, but with a wordlength of only 4 bits the deviation from the reference gets too big. When the dynamic range gets smaller than 4 bits the detection collapses. The 6-bit wordlength was very close to the reference in the diagram but due to the resolution of the plots it was not included here.

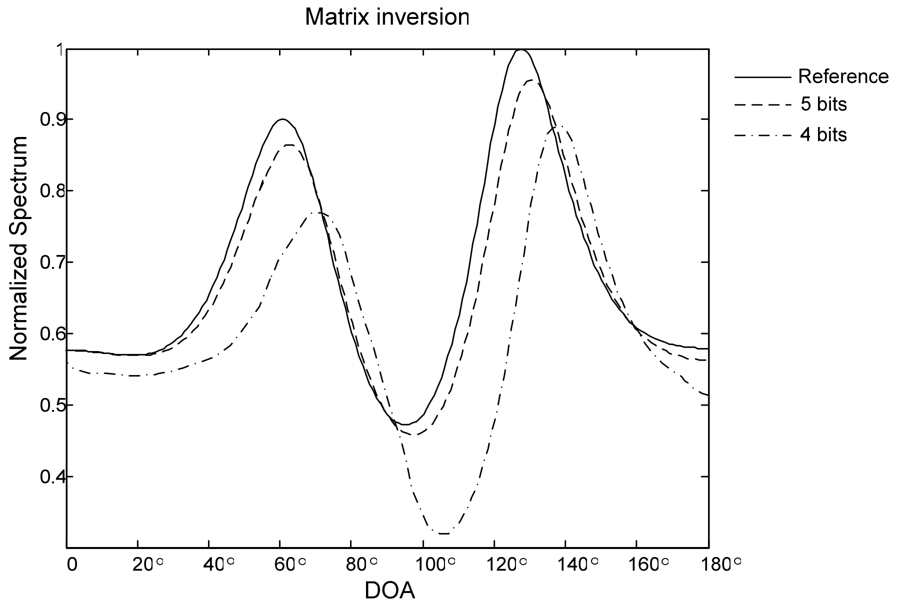


Figure 4.3 Normalized power spectrum for the matrix inversion block for different wordlengths.

4.3 Steering vector

The effect of reduced dynamic range in the steering vector block appears quickly. Even for a small decrease in dynamic range the peaks disappear and get virtually undetectable. A wordlength of 7 bits produced a power estimation diagram close to the reference and without the jaggedness, but a wordlength of 6 bits changed the diagram significantly. As seen in figure 4.4, the diagram became very jagged making it hard for a search algorithm to determine the number of peaks. Therefore, a 6-bit wordlength is too short to use in applications. However, the basic shape of the peaks was still visible. Some kind of smoothening operation can be applied to the diagram in order to restore the shape. When a wordlength shorter than 6 bits was used, we were unable to distinguish between the reference and the plotted line.

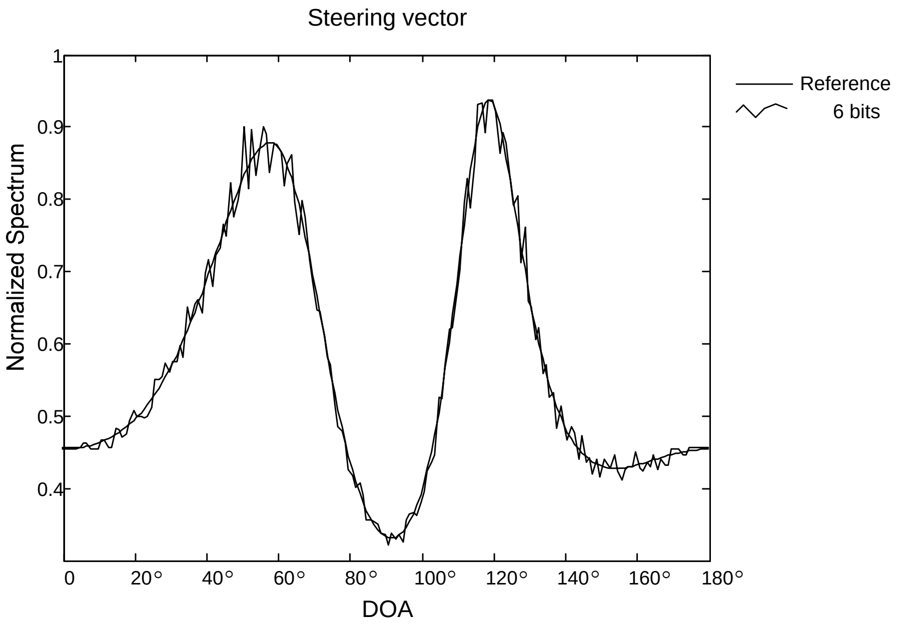


Figure 4.4 Normalized power spectrum for the steering vector block for different wordlengths.

4.4 Inversion

The effect of reducing the wordlength in the inversion step was not significant, as can be seen in figure 4.5. When the wordlength decreased, the diagram became more “digitized” in appearance. This is a result of the difference between neighboring values disappearing after the inversion due to the reduced dynamic range. It is still possible to detect the peaks with a wordlength of 5 bits. It is even possible to use 4 bits, but when the wordlength is reduced to 3 bits, the computation collapses and no peaks are detectable. In this case a wordlength of 5 bits was viewed as the limit.

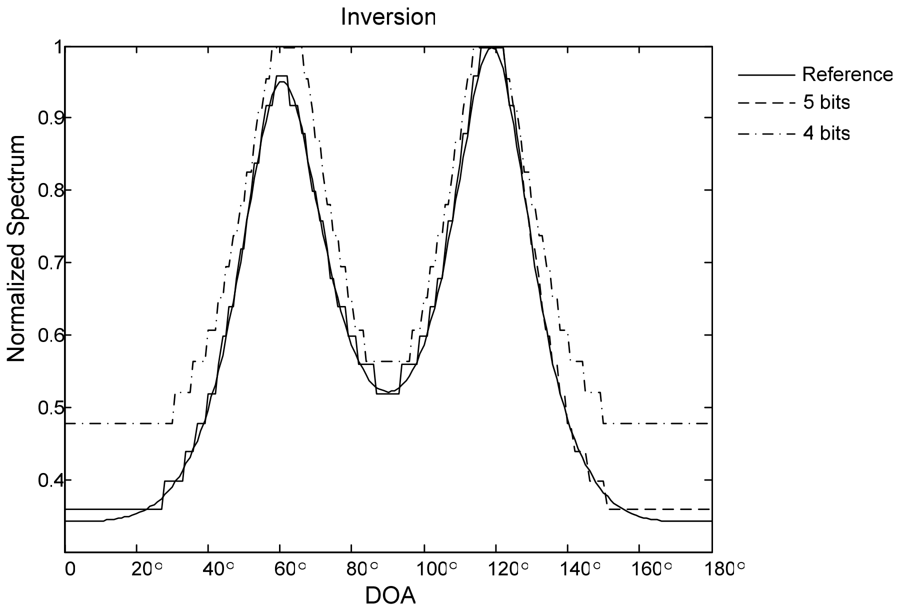


Figure 4.5 Normalized power spectrum for the inversion block for different wordlengths.

4.5 Conclusion

The step including multiplication of the steering vectors is the one most sensitive to reductions in the dynamic range. The group of high-resolution methods that the Capon algorithm belongs to is known to be sensitive to errors in the steering vectors. The inaccuracy induced in the computation with the steering vectors is comparable to using steering vectors from a badly calibrated antenna array.

The second most sensitive operation is the first step when creating a covariance matrix. When the dynamic range in the covariance matrix reduces the variation between the element values in the matrix, it becomes ill-conditioned and in worst case singular.

The dynamic range of the matrix inversion and the inversion operations does not significantly affect the overall computation. This indicates that the relation between the matrix elements is more important than the precision of a single element. The wordlengths can therefore be reduced significantly in these operations. A design with a minimum wordlength can be derived based on the information above.

4.6 Minimum wordlength design

From the results above, the wordlength in the steps can be reduced significantly. However, the accumulated effect must be taken into account when changing the wordlength of all blocks in the design. Therefore, more bits were needed in some of the operations than indicated by the diagrams in the previous section. Several FPGA implementations were analyzed before a minimum wordlength design could be established. The minimum wordlength design is implemented using the following wordlengths:

- The covariance processor was implemented using a wordlength of 7 bits.
- The matrix inversion was implemented using a wordlength of 6 bits.
- The steering vector processor was implemented using a wordlength of 7 bits.
- The inverter block was implemented using a wordlength of 6 bits.

Figure 4.6 shows the power spectrum of the derived minimum wordlength design. The deviation from the reference is less than 3°. The minimum wordlength design occupies 18% less slices than the original implementation of the LDF architecture, which is a significant reduction in resource consumption. The choice of wordlength of the original design was based on a perceived need of precision. The precision needed for a specific application may differ, which it did in this case, from the perceived precision of a single block.

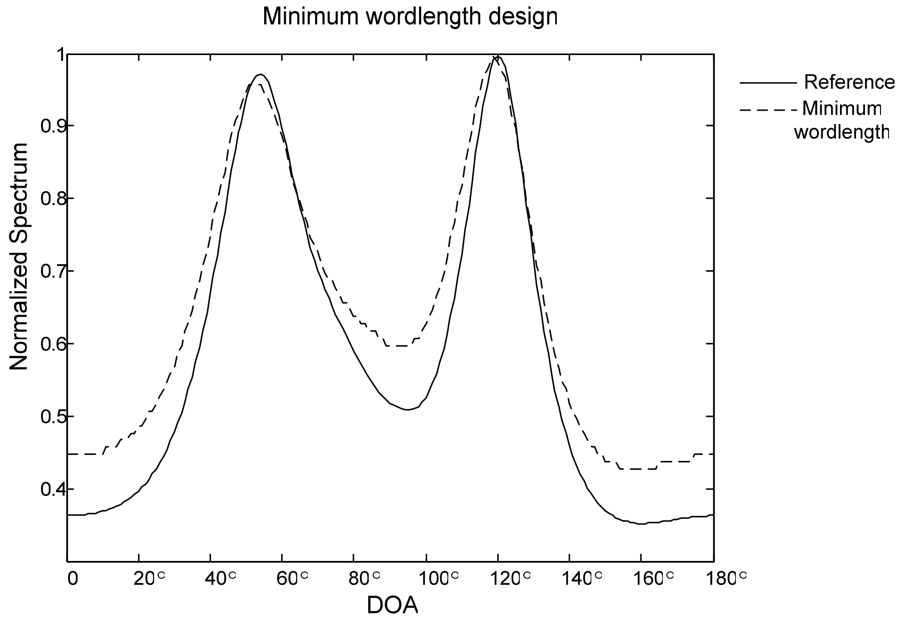


Figure 4.6 Normalized power spectrum of the derived minimum wordlength design.

Chapter 5

Application: Channel sounding

In this chapter, an application utilizing the Capon beamformer is discussed. Channel sounding is an important tool in the development of radio channel models, which are used in research in wireless communication. In the first section of the chapter the importance of channel sounding is discussed, followed by a brief description of a channel sounder. Finally, the FPGA implementation of the Capon algorithm is used for DOA estimation on real measurement data.

5.1 Channel sounding

Studying radio propagation in indoor and outdoor scenarios is an important step in research and development of wireless communication systems. To gain knowledge of radio propagation in different environments, a measure system such as a channel sounder is of key importance. Channel sounding is therefore the first step in understanding the radio channel with respect to new applications.

A transmitter sends out a signal that excites the radio channel. The output of the channel is observed by the receiver and stored. From the knowledge of the transmitted and received signals the time-variant impulse response or other deterministic system functions are obtained. The impulse response can be seen as one snapshot or sample of the radio channel. In order to track the channel, these snapshots must be taken sufficiently often but in a small time frame so that the channel does not change.

Real world measurements help in creating useful radio channel models for the development of new algorithms in mobile radio communications. To be able to perform measurements, a MIMO channel sounding system must be used. Real-time measurements with the channel sounding system require high-speed hardware implementations of MIMO and beamforming algorithms with good numerical precision. The hardware architecture must also be flexible and scalable to comply with different antenna configurations used in a variety of measurement scenarios.

5.2 Channel sounder system

The channel sounder system at the Department of Electroscience is a real-time radio channel impulse response measurement system based on the switched array principle [5]. Due to the modular construction of the system, it can easily be configured for a wide range of MIMO measurements using different kinds of antenna configurations.

Figure 5.1 shows a block diagram of the channel sounder system. In the transmitter, arbitrary signals can be generated by a waveform generator and transmitted via the configurable antenna array.

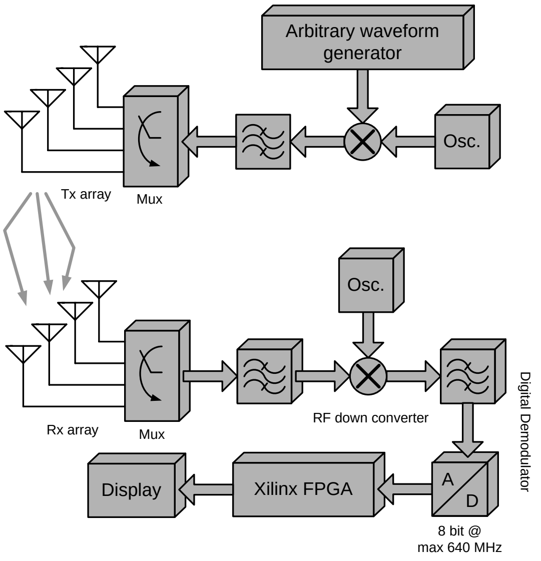


Figure 5.1 Block diagram of the transmitter and receiver unit of the channel sounder system.

The received signal, arriving at the receiver antenna array, is down converted, filtered, and sampled into a digital datastream. The 8-bit A/D-converter has a maximum sample rate of 640 MHz and the sampled data is collected and temporarily stored on a hard drive.

The Capon beamformer algorithm is implemented in hardware, as described in previous chapters, using a Xilinx FPGA. The FPGA has not yet been integrated into the channel sounder. The computed power spectrum of the received signal is displayed on an external monitor. The supported radio frequency bands of the system are UMTS at 1.8 to 2.5 GHz and WLAN at 5 to 6 GHz. The maximum bandwidth of the measurement system is 240 MHz and the maximum Tx power is 10W. Figure 5.2 shows the channel sounder equipment and a flat 32 element and a circular 64 element patch antennas built by Ericsson microwave.

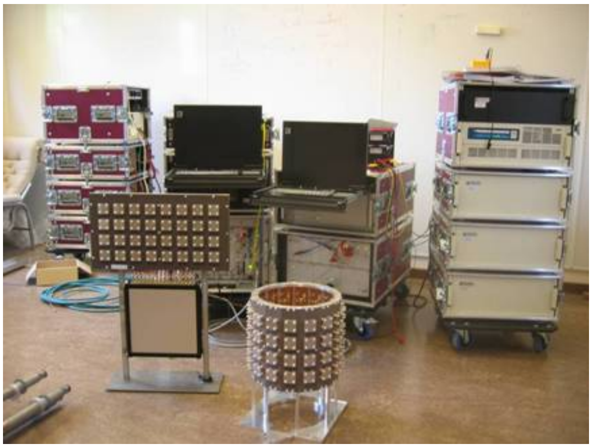


Figure 5.2 Channel sounder system and antennas.

5.3 Measurement

The Capon beamformer implementation was tested together with measured data from the channel sounder. A uniformed linear array with four antenna elements and a SNR of 12 dB was used in the measurement. The sampled data coming from the antenna array consisted of four vectors with 120 samples of complex valued data, of which 120 snapshots were taken.

Figure 5.3 shows a plot of the result with the normalized spectrum on the y-axis and the DOA in degrees on the x-axis. A sharp peak is found in the direction of the source that was placed 120° in respect to the ULA. A second source, which in this case is a reflection on one of the walls, impinges from 87°. The reference plot in MATLAB (solid line in the diagram) may also show a minor reflection around 150°. However, this reflection is not detected by the Capon beamformer implementation. This may be due to too low resolution depending on the minimized wordlength used in the implementation. An increase in the wordlength would probably reduce the difference between the reference plot and the measured plot.

The FPGA implementation of the Capon beamformer has not yet been incorporated into the channel sounder. The goal is to be able to make measurements with the FPGA incorporated into the channel sounder, delivering the power spectrum diagram in real-time.

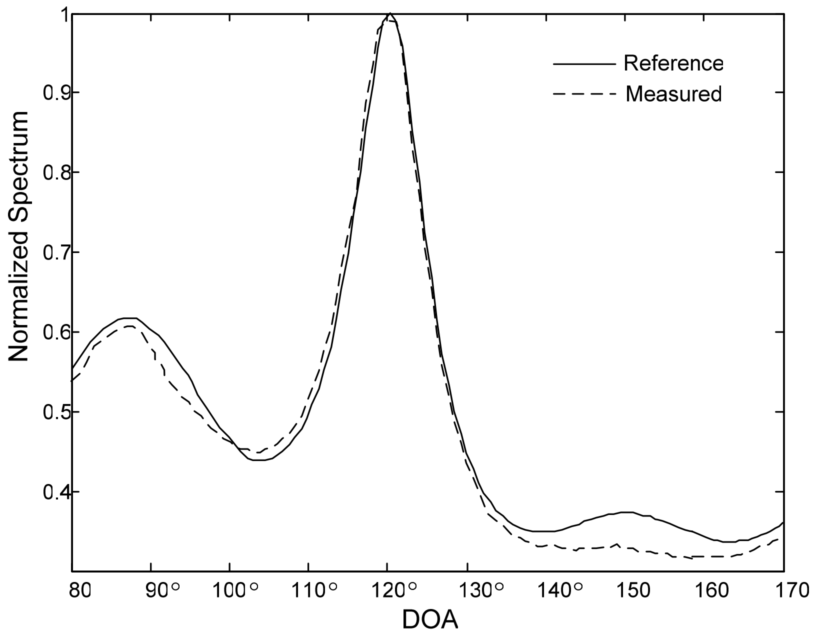


Figure 5.3 Measured power spectrum of the Capon beamformer algorithm plotted for different DOAs.

5.4 Summary

In this part of the thesis we have seen that the building blocks for multiantenna algorithms can be used in practice. The direction of arrival algorithm Capon's beamformer has been implemented, analyzed, and successfully tested with measured data from a channel sounder.

The analysis showed that the computation of the correlation matrix and the computation with the steering vectors were the most critical steps in the Capon algorithm in terms of precision.

The configuration of the channel sounder constantly changes, depending on which types of measurements to make. The flexibility and the scalability of the matrix building blocks make them perfect to use in such an application.

High-resolution DOA estimation algorithms are not only applicable in channel sounders. They are also important in many other types of systems, such as radar, sonar, mobile communications, and electronic surveillance.

Part IV

Design Methodology

Chapter 1

Introduction

This part begins with a brief discussion about the importance of using a design methodology when designing in hardware. In chapter 2, a design methodology developed specially for designing with FPGAs is presented. The design methodology has been developed and refined by the author during his research years. In chapter 3, the system modelling and the hardware tools used in the hardware implementations are discussed.

1.1 Why do you need a design methodology?

Developing and implementing designs in hardware without a design methodology is like putting together IKEA furniture without a manual or baking a cake without a recipe, it is possible but not clever.

In the author's opinion a good design methodology is an absolute must when implementing in hardware. Using a design methodology will dramatically increase the chances of creating an implementation that will function correctly and efficiently. Following the design methodology will also structure your work and it will be much easier to optimise and debug your design, resulting in savings in both time and resources.

A design methodology usually consists of a number of different steps describing the order in which tasks are to be carried out, e.g. write a specification, simulate, write code, test the design, and so on. Before starting to use a design methodology one should customise it to ones own needs and lay down the ground rules of how it is going to be used. One or more steps may need to be omitted or rearranged to suite your design project. However, once you have started to work within a design methodology it is important not to deviate from it, especially if more than one person is involved in the design process. A deviation could in worst case lead to time consuming debugging or serious problems in later steps.

A design methodology slightly differs depending on whether the target is an integrated circuit (ASIC) or a programmable device (FPGA). The target for most of the designs presented in this thesis is an FPGA. The design methodology described in this chapter has been derived and refined by the author throughout the work that is the basis of this thesis. It has been developed especially for programmable devices and it is referred to as the *Design Methodology for Programmable Devices,* or short DMPD. The DMPD can, with some modifications to individual steps, be used in ASIC design.

Chapter 2

DMPD

The DMPD design flow consists of a number of steps that enables one to reach designated design goals. The different steps of the DMPD are shown in figure 2.1. Each individual design has of course its own variation of the DMPD, but essentially the steps are the same for all FPGA implementations.

The DMPD consists of two major phases and one sub-phase. These phases are the *high level design*, the *low level design*, and the *design verification*. The different phases are marked with grey and white boxes including several steps in figure 2.1.

A hardware implementation project is usually prompted by an idea, a request, a specification, or some other external input. The first phase in a project is the high level design where a system model of the idea is created, simulated, and verified in a software environment. The second phase is the low level design where the verified system model is implemented in hardware. This phase also includes a design verification sub-phase in which the hardware implementation is verified and integrated onto the FPGA. The design phase will result in a hardware design implemented on the FPGA, which will be ready for testing and integration into a larger external system.

A brief description of the individual steps in the various phases is given below.

2.1 High level design

The DMPD is divided into two major blocks, the high-level design and low-level design. In this thesis, high-level design denotes the algorithm level where the idea or specification is modelled and simulated using digital signal possessing (DSP) algorithms.

2.1.1 DSP system specification

Creating a DSP system specification is essential if one is going to be able to make the right decisions later on in the design process. The specification will also help to fully understand and to think through the design in detail. This is most important and will

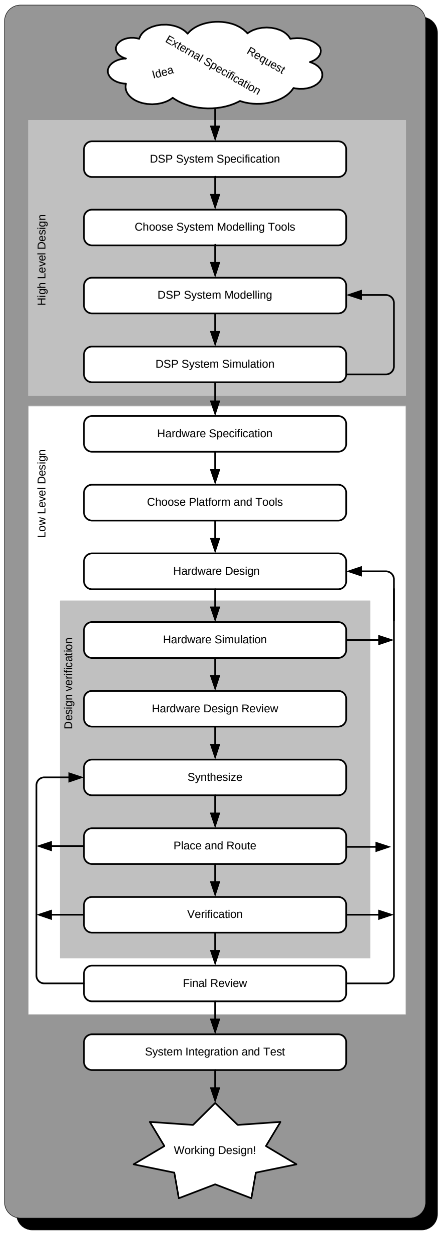


Figure 2.1 The design flow of the DMPD used in the development of FPGA hardware implementations.

save you time and help you to avoid obvious mistakes. A DSP system specification should at least contain the following information:

- A basic block diagram of all the major building blocks in the design (arithmetic units, memories, data busses, etc.) and how they are connected together.
- A basic block diagram of how your design fits together with the external system.
- A description of the I/O system connecting your design to the external system.
- A description of how the software system model is going to be tested and which test data is going to be used. If a specific test bench will be constructed, a basic block diagram of that test bench should also be included.

It is also a god idea to review the DSP system specification together with people involved in the same project, or with co-workers, to discover any errors or omissions in the specification.

2.1.2 Choosing system modelling tools

The following two criteria should be considered when choosing system modelling tools.

1. The tool should be able to model the hardware designs at a fairly low level (bit or word level) and be able to handle fixed-point and floating-point arithmetic. This will generate a sufficiently accurate hardware model enabling you to make valid design decisions [62]. Creating a model that simulates the hardware in a detailed fashion will also facilitate the process of converting the model to actual hardware.
2. Choose tools that are commonly used at your workplace, as this will help you get in house support of the tool. If you are working with other research groups you should choose a tool, if possible, that both you and the other research groups are familiar with. This will enable a greater understanding between the groups in terms of discussing and revising the system model.

However, in some cases the choice of modelling tool is not yours to make. It could be that you have been given an algorithm to implement, modelled in a specific tool. The choice is then to either translate the algorithm into a more familiar environment, with the risk of introducing errors or simplifications, or to learn the new tool. This decision should always be made together with every one involved in the project.

2.1.3 DSP system modelling

The specification produced in the previous step will act as a blueprint for the modelling of the system. During modelling it is a good idea to break down the requirements set by the system specification and build a system model which is modular in its design. By doing so it will be much easier to find and eliminate bugs and to analyse and optimize parts of the designs. In many cases, system modelling is not separable from the next step, which is the simulation of the system model.

2.1.4 DSP system simulation

In most cases, several iterations between the DSP system modelling step and the DSP system simulation step are needed in order to reach a design that meets the requirements of the DSP system specification. The simulation will provide valuable design information, such as required dynamic range, word sizes, memory sizes, bus sizes, timing constraints, number of I/O pins, and design type (asynchronous or synchronous). This step concludes the high level design phase.

2.2 Low level design

In low-level design the algorithm modelled and simulated in the high-level design is translated into a hardware language, describing how it is going to be implemented.

2.2.1 Hardware specification

The hardware specification is written with the support of the results from the high level simulations. This step is sometimes neglected by hardware designer. However, this step is very important, since faults and oversights may have serious implications on the hardware design in a later step. Spending time correcting errors originating from faults in the hardware specification may be very expensive. The hardware specification should at least contain the following information:

- A high level block diagram of all the major hardware building blocks.
- A detailed level block diagram containing realisations of blocks.
- The choice of clocking strategies, e.g. synchronous or asynchronous timing, and clock domain (global or multiple).
- A description of the I/O realisation such as the number of pins needed, external timing constraints, and data formats.
- Timing estimates of the hardware design, such as clock cycle time, propagation times, and setup time where applicable.
- Basic area estimation to aid in the choice of FPGA.
- Basic power estimation.

- Which kind of test procedure to use (test bench, self-testing circuitry, or a verification tool).

This step should also be reviewed with people involved in the same project, or with co-workers, to eliminate as many mistakes as possible.

2.2.2 Choosing platform and tools

Usually, two different scenarios arise when it is time to choose which FPGA platform to use.

1. You already have a specific FPGA that you have to work with. The challenge is then to fit the design into that particular FPGA.
2. You can choose freely which FPGA to use. The hardware design specification will then be the guide for choosing the right FPGA. The area and timing estimations, together with the number of I/O pins needed, will give you the framework needed to be able to choose the right FPGA.

When the FPGA has been chosen it is time to determine which tools to use for the development of the hardware and the programming of the FPGA. This includes choosing a HDL as well as tools for doing the synthesis of the design, the place and route, and the simulation and verification. A good strategy is to use tools that are supplied by the FPGA vendor, or use tools from a third-party company co-operating with the vendor and thereby avoiding tools that produce non-compatible formats [62]-[65].

2.2.3 Hardware design

Before you start to write code in a HDL, some coding and design rules should be set up. These rules will help you in designing good and reliable code. The following coding rules were used in the author's research work:

- Design with the device architecture in mind. In a specific situation one module of code may result in a design occupying fewer slices in the FPGA than another module of code [66]. The best way of determining this is by continuous testing of the code.
- Only synchronous designs. A synchronous design will reduce the probability of problems such as race conditions, hold time violations, glitches, and delay dependencies [67].
- Modulise your HDL code. This will help you get a better overview of your code and it will also be much easier to debug [67].
- Design code modules that are flexible and optimised. The modules should be easy to expand in terms or wordlength, number of inputs, etc.

- Design for reusability. Parts of the code can often be used in several designs. If these parts are designed with some kind of generality it will be easy to reuse in other designs [70].
- Design for testability, especially if the design is going to be large. Testing of your design could either be carried out by using test vectors or by inserting test logic into your design. However, the 10/10 rule which says that you should not spend more than 10% of your time on designing test logic and not more than 10% of the logic of the FPGA should be used for testability, should be obeyed [66].
- Keep it simple.
- Always make comments in the code. The code should be like a story book.

The compiled HDL code will produce register transfer level (RTL) code used in later steps.

2.2.4 Hardware simulation

Simulation and hardware design is an iterative process. It is wise to continuously simulate small parts of your HDL code separately before hooking them up to larger parts. It is also much more time-consuming to find errors in large code chunks than in smaller. By modulizing your code and design it for testability, by for instance using test logic, it is much easier both to debug it and to optimize it in a later stage. The hardware simulation is the first step in the design verification phase incorporated into the low level design phase.

2.2.5 Hardware design review

Once the design and the simulations have been completed, a design review should be performed to verify and ensure the correctness of the design. In this step it is very important to be thorough and ensure that the function of the hardware design is in compliance with the hardware design specifications before implementing it on the FPGA.

2.2.6 Synthesis

The synthesis involves using tools that translates the design into a gate level design that can be mapped to the logic blocks on the FPGA. This step may also include choosing and specifying switches and optimization parameters of the synthesis software that will affect the end result. It is good to play around a bit with the software since the same setting in the software will not produce the same result in two different designs.

After a successful synthesis the design must be verified against the RTL code. You can either re-simulate the design using the gate level output of the synthesis tool or you can use a verification tool that logically compares the RTL description to the gate level description.

2.2.7 Place and route

In this step the design is physically mapped (placed) and connected together (routed) on the FPGA. The place and route tool figures out how to program the FPGA to physically implement the design. If the placement for some reason fails, the design can often be tweaked so that it can be placed successfully but sometimes you have to make radical changes and re-design to be able to place it. This is especially true when the design is so big that it nearly consumes all the cells in the FPGA.

Once the place and route is successful, a timing analysis must be done to determine whether the design meets the timing goals. Typically, certain parts needs to be changed or the timing specifications must be altered in order to get the design to work properly.

2.2.8 Verification

At this stage the design must be checked thoroughly to verify that it behaves in the same manner as the RTL code. This can be done either by simulation of the lower level design or by using formal verification (or sometimes both).

The most commonly used method is to re-simulate the final circuit using the test data that was used to simulate the original circuit. This procedure is called regression testing.

Formal verification is the process of mathematically checking that a design is behaving correctly. There are two types of formal verification:

1. Equivalency checking is the process of comparing two designs descriptions to determine if they are equivalent. Equivalency software will check if the RTL description inserted into the synthesis software is functionally equivalent to the outputted gate level description [67].
2. Functional verification is the process of proving whether specific conditions occur in a design. These conditions could be legal conditions, that a certain signal is generated when an address has valid data or illegal conditions such as overflow during a computation or a forbidden state in logic. The functional verification tool checks if all legal and illegal assertions occur in the design [67].

2.2.9 Final review

At this point the final review should just be a formal matter and a sign-off that the design has been coded, simulated, synthesized, placed and routed, and verified that it is now completed and ready to be integrated with the external system. At this stage no serious problem should arise, but if it does parts of the design may need to be re-designed or re-synthesised.

2.3 System integration and test

When the design has passed the final review, the FPGA hardware implementation is ready to be used and integrated into the external system. Hopefully there has been a constant communication between all parts involved in the design of the system, avoiding any serious compatibility problems between the system and the FPGA. The FPGA design will now evolve together with the system.

Chapter 3

Tools used in the research

3.1 System modelling tools

MATLAB and Simulink [62] were used as the primary system modelling tools for all the designs in this thesis. The choice of MATLAB as the system modelling tool was not a difficult one to make. MATLAB is among the leading modelling and implementation tools for high performance DSP systems and together with the Simulink environment makes a good platform for model-based design. Simulink provides an interactive graphical environment and a customizable set of block libraries, called toolboxes, which makes it easy to create models [62]. Specialised toolboxes, like the fixed-point toolbox, enable design and simulation of signal processing systems using fixed-point arithmetic, which is crucial for hardware design. The product overview in figure 2.2 shows the variety of toolboxes available for the MATLAB and Simulink environment. The design interface also allows you to create your own toolboxes. Many hardware vendors also supply their own specialized and optimized toolboxes for Simulink.

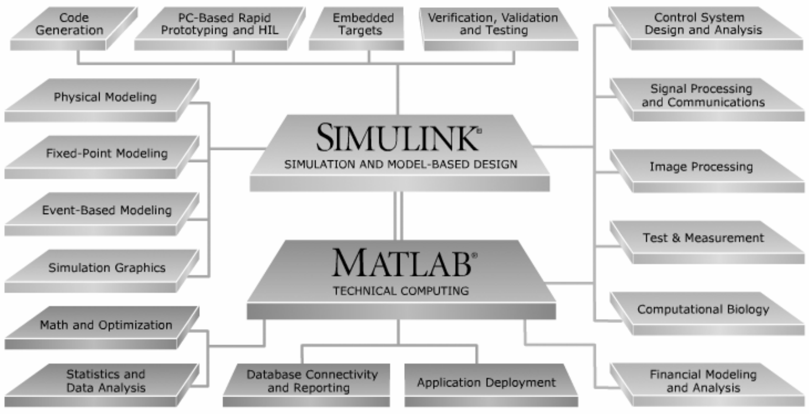

Code Generation
PC-Based Rapid Prototyping and HIL
Embedded Targets
Verification, Validation and Testing
Control System Design and Analysis
Physical Modeling
Signal Processing and Communications
Fixed-Point Modeling
SIMULINK
SIMULATION AND MODEL-BASED DESIGN
Image Processing
Event-Based Modeling
Test & Measurement
Simulation Graphics
MATLAB
TECHNICAL COMPUTING
Computational Biology
Math and Optimization
Statistics and Data Analysis
Database Connectivity and Reporting
Application Deployment
Financial Modeling and Analysis

Figure 3.1 MathWorks product overview [62].

One such vendor is Xilinx, who offers a specialized toolbox, Xilinx System Generator [68], for Simulink that enables development and bit and cycle accurate simulations of DSP systems for integration into Xilinx FPGAs [65]. Figure 3.2 shows a CORDIC based divider architecture modelled in Simulink with the Xilinx System Generator hardware blocks. By double-clicking on the hardware blocks in the design you can easily change hardware parameters such as word sizes, rounding technique etc., which makes it quick and easy to simulate different configurations. Specialized blocks, such as the Resource estimator in the upper right corner, will give you useful information about the estimated hardware cost when comparing similar implementations.

The System Generator tool also provides the possibility for a high-level abstraction to be automatically compiled into an FPGA at the push of a button [62],[65]. The tool will automatically generate synthesizable HDL code mapped to Xilinx pre-optimized building blocks, which can be synthesized for implementation on Xilinx FPGAs. System Generator can also provide automatic generation of a HDL test bench, which enables design verification during implementation. However, the automated compile feature in Xilinx System Generator was not used in any of the hardware designs in the thesis, due to limitations in the control of the lower level hardware, such as timing etc. This feature was only used in construction of test benches.

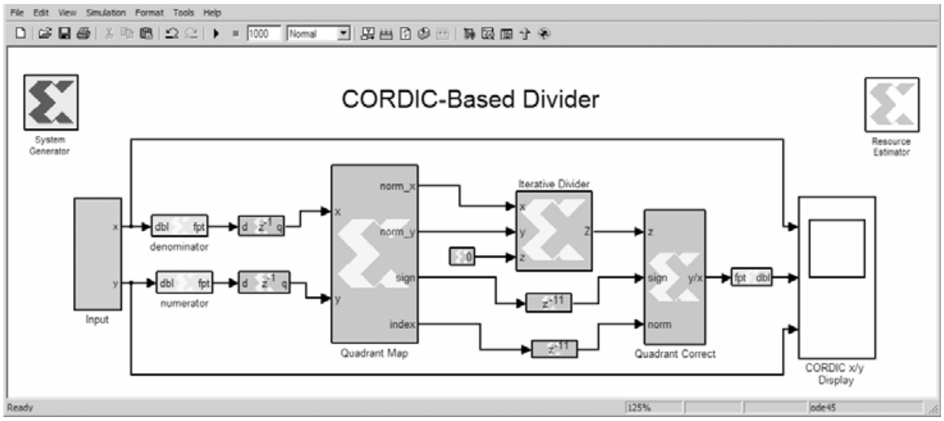


Figure 3.2 A CORDIC-based divider designed and simulated using MATLAB Simulink and Xilinx System Generator.

3.2 FPGA and hardware design tools

The choice of FPGA vendor was easy since the department had access to Xilinx FPGAs. Xilinx is one of the largest and oldest FPGA vendors in the world, which makes it easy to get support and to find useful information in various hardware designer forums.

The most widely used HDLs [71] are Verilog HDL [69] and VHDL [70]. Which language to choose depends on which HDL is used in your design team, on which level (system, behavioural, etc.) the code will be written, and on your own preferences. VHDL is a very strong language when it comes to behavioural and system level design construct. However, Verilog is easier to master than VHDL and is stronger when it comes to designing on the structural and functional level. Figure 3.3 shows a comparison between the level of abstraction of VHDL and Verilog HDL. VHDL is also able to replicate a number of instances of the same design-unit or some sub part of a design, and connect them appropriately. There is no equivalent to the generate statement in Verilog. Together with the advantages in the behavioural and the system level over Verilog, the choice for the research fell on VHDL.

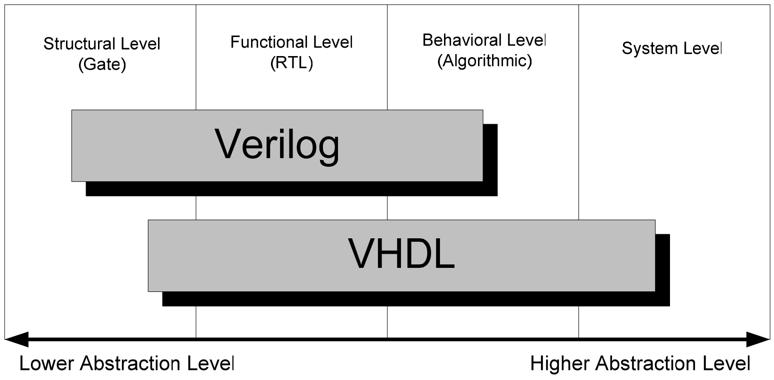


Figure 3.3 A comparison of the level of abstraction between the two HDLs Verilog and VHDL.

As discussed in the previous chapter it is wise to use the FPGA vendor's tools or third-party tools that are compliant with the FPGA. The synthesis tools used in the designs in this thesis include Xilinx own ISE Foundation [72] and Synplify from Synplicity [64]. Synplify is a high-performance logic synthesis engine that utilizes the proprietary Behavior Extracting Synthesis Technology (B.E.S.T.) to deliver fast and efficient FPGA designs. The Synplify tool takes HDL as input and outputs a netlist in Xilinx FPGA vendor format. In some designs the Synplify tool generated a 16% increase in efficiency compared to the ISE Foundation tool. The place and route of the design was performed by the Xilinx ISE Foundation tool [72].

When it comes to verification of the FPGA implementation, Xilinx ChipScope Pro offers good in-design testing capabilities [73]. Xilinx ChipScope Pro is a real-time verification tools that provide on-chip debug at or near operating system speed [73]. This tool inserts a logic analyser, bus analyser, and Virtual I/O low-profile software cores directly into your design. This allows you to view any internal signal or node, including embedded hard or soft processors. Wanted signals are captured and brought out through the programming interface and can be analyzed through the ChipScope Pro Logic Analyzer [73]. The tool is very potent and useful but the down side is that it consumes resources (area) in your FPGA. If ChipScope Pro is to be used during the design phase, it should be included in the hardware design specification so that adequate hardware resources can be allocated. The logic added to your design when you use ChipScope should be removed when the final design is tested and ready to be used. In cases where ChipScope Pro can not be used, regression tests must be used to confirm the FPGA implementation. If you have designed for testability the simulation process will be much easier.

Another important tool is some kind of version handling system. When writing and simulating on system level or at the hardware level, there will be many different versions of the same system or code. A version handling program can prove to be very helpful tool to keep track of changes of your design. The first version handling software used in with the design methodology was the Concurrent Versions System (CVS) tool [74]. The program is an open source initiative and there exists clients for the most common computer platforms (Windows, Unix, Linux, OS/2, etc.).

Later on CVS was exchanged for another version handling tool called Subversion [75]. This is also an open source program and free to use. The choice of which tool to use is a matter of taste but Subversion has a better tracking system for different types of files (.txt, .bin, .doc, …) than CVS and it also keeps track of files that changes name.

If a version handling tool is going to be used one may find it very useful to visit the Version Control System Comparison webpage [76]. This webpage keep track of and compare commercial and free version handling tools.

Appendix A

Appendix A: Real divider

Introduction

The real divider used in several designs in this thesis is based on an article [77], which uses Taylor series expansion to approximate the real division. As shown in figure A.2, the real divider is constructed of two redundant modified-Booth multipliers [78], and a small lookup table (LUT) with normalized and optimized table entries for high numerical accuracy. The total error of the real valued division is below 1 ulp.

Algorithm

Consider the real operation *A/B*. The *2m* bit operand *B* is split into two operands, B_{Hb} with the *m+1* higher order bits, and B_{Lb} with the *m-1* lower order bits, as shown in figure A.1.

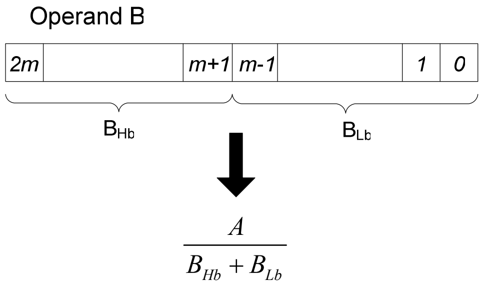


Figure A.1 Division of the operand *B* into higher and lower bits.

The expression in figure A.1 can be Taylor series expanded at B_{Lb}/B_{Hb}

$$\frac{A}{B_{Hb}+B_{Lb}} = \frac{A}{B_{Hb}} \cdot \left(1 - \frac{B_{Lb}}{B_{Hb}} + \frac{B_{Lb}^2}{B_{Hb}^2} - \cdots\right). \tag{A.1}$$

Combining the two first terms in the series will result in the approximation shown in equation (A.2) [77].

$$\frac{A}{B} = \frac{A}{B_{Hb}+B_{Lb}} = \frac{A(B_{Hb}-B_{Lb})}{B_{Hb}^2 - B_{Lb}^2} \approx \frac{A(B_{Hb}-B_{Lb})}{B_{Hb}^2} \tag{A.2}$$

The approximation in equation (A.2) can then be used to calculate a real division.

Hardware implementation

A block diagram of the hardware implementation of equation (A.2) is shown in figure A.2.

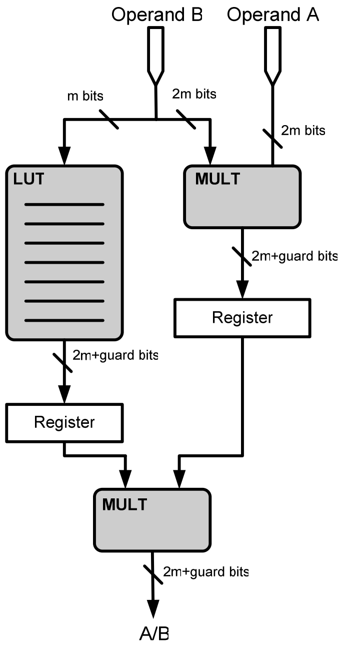


Figure A.2 Hardware schematics of a real valued divider.

The higher order bits of operand *B*, are used to look up the value of $1/B_{Hb}^2$ with a *2m+3* bit accuracy. The lookup table consists of $2^m(2m+1)$ bits and the table entries are normalized and optimized to maintain a high numerical accuracy. Operand *A* is multiplied with the partial operand $B_{Hb} - B_{Lb}$. The multipliers in the real divider implementation use a general redundant modified-Booth scheme to be able to perform the multiplication $A(B_{Hb} - B_{Lb})$ without actually calculating $B_{Hb} - B_{Lb}$. Two guard digits are used in the first multiplication for error compensation. The first multiplier produces a $2m+3$ bit answer, which is stored in a pipeline register until the next cycle. The second multiplier then produces the result in the second cycle.

References

References

[1] Tapan K., Sarkar, Michael C. Wicks, Magdalena Salazar-Palma, and Robert J. Bonneau, *Smart Antennas.* Wiley-IEEE Press, 2003.

[2] H.L. Van Trees, *Optimum Array Processing*. Wiley, New York, 2002.

[3] D.H. Johnson, and D.E. Dudgeon, *Array Signal Processing*. Prentice Hall, 1993.

[4] R.A. Monzingo, and T.W. Miller, *Introduction to Adaptive Arrays*. Wiley, New York, 1980.

[5] J.E. Hudson, *Adaptive Array Principles*. Peter Peregrinus, London, 1981.

[6] J.C. Liberti, and Jr., T.S. Rappaport, *Smart Antennas for Wireless Communications*. Prentice Hall, 1999.

[7] A. F. Molisch. *Wireless Communications*. John Wiley and Dons Ltd. , 2005.

[8] T.S. Rappaport, *Wireless Communications: Principles and Practice*. Prentice Hall, 2002.

[9] A.J. Paulraj, R. Nabar, and D. Gore, *Introduction to Space-Time Wireless Communications*. Cambridge University Press, Cambridge, 2003.

[10] "*Lucent Technologies, Bell Labs,*" http://www.bell-labs.com/project/blast/, 2006-01-01.

[11] D. Tse and P. Viswanath. *Fundamentals of Wireless Communication*. Cambridge University Press, Cambridge, 2004.

[12] S.M. Alamouti, "A Simple Transmit Diversity Technique For Wireless Communications," *IEEE Journal on Selected Areas in Communications*, v.16, N.8, pp. 1451 -1458, Oct. 1998.

[13] D. Gesbert, M. Shafi, D. Shiu, P.J. Smith, and A. Naguib, "From Theory to Practice: An Overview of Mimo Space-Time Coded Wireless Systems," *IEEE J. Selected Areas Commun.*, vol. 21, no. 3, 2003.

[14] G.H. Golub and C.F.Van Loan, *Matrix Computations.* The Johns Hopkins University Press, Maryland, 1996.

[15] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing; Principles, Algorithms, and Applications*. Macmillan Publishing Company, 1992.

[16] K. J. Ray Liu, and Kung Yao, *High-Performance VLSI Signal Processing Vol. 1 Algorithms and Architectures*. IEEE Press, 1998

[17] "*AMD*," http://www.amd.com/, 2006-01-01.

[18] "*Intel*," http://www.intel.com/, 2006-01-01.

[19] "*Texas Instruments*," http://www.texas-instruments.com/, 2006-01-01.

[20] "*Lyrtech Signal Processing*," http://www.lyrtech.com/, 2006-01-01.

[21] “*M-Tec Wireless*,” http://www.mtecwireless.com/, 2006-01-01.

[22] S.W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 1997.

[23] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*. John Wiley and Sons, 1999.

[24] R.L. Smith, “Algorithm 116: Complex division,” *Communications of the ACM*, 5(8), 1962.

[25] G. W. Stewart, “A Note on Complex Division,” *ACM Transactions on Mathematical Software*, 11(3):238-241, 1985.

[26] J. T. Coonen, “Underflow and Denormalized Numbers” *IEEE Tran. on Computers*, 13:68-79, 1980.

[27] J. L. Hennessy and D. A. Patterson, *Computer Architecture a Quantitative Approach*. Morgan Kaufmann Publ., 1990.

[28] B. Parhami, *Computer Arithmetic: Algorithm and Hardware Designs*. Oxford University Press, 2000.

[29] L. Wanhammar, *DSP Integrated Circuits*. Academic Press, 1999.

[30] F. Edman and V. Öwall, “Hardware Implementation of two Complex Divider Architectures," *Proceedings of RVK’05,* Linköping, Sweden, June 2005.

[31] F. Edman and V. Öwall, “Fixed-point Implementation of a Robust Complex Valued Divider Architecture,” *Proceedings of ECCTD’05,* Cork, Ireland, August 2005.

[32] J. Volder, “Binary Computaion Algorithms for Coordinate Rotation and Function Generation,” *Convair Report IAR 148 Aeroelectric Group*, June 1956.

[33] J. S. Walter, “A Unified Algorithm for Elementary Functions,” *Proceedings of Spring Joint Computer Conf.*, 1971.

[34] S. Wang and V. Puri, *A Unified View of CORDIC Processor Design*. Application Specific Processors, Kluwer Academic Press, 1996.

[35] P. Pirsch, *Architectures for digital signal processing*. John Wiley & Son, 1998.

[36] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing; Principles, Algorithms, and Applications 2:ed.*. Macmillan Publishing Company, 1992.

[37] N. Kalouptisidis, *Signal Processing Systems; Theory and Design.* John Wiley and Sons, Inc., 1997.

[38] A. El-Amawy and K. R. Dharmarajan, "Parallel VLSI algorithm for stabel inversion of dense matrices," *IEEE Proceedings on Computers*, vol. 136, 1989.

[39] J. Spars, *Principles Asynchronous Circuit Design*. Kluwer, 2002.

[40] R. L. Walke. *High Sample-Rate Givens Rotations for Recursive Least Squares.* Ph.D. thesis Warwick University, 1997.

[41] C. M. Rader, "MUSE: A systolic array for adaptive nulling with 64 degrees of freedom using Givens transformation and wafer scale integrastion," *Proc. of the Inst. Conf. of Application Specific Array Processors*, pp. 277-291, 1992.

[42] F. Edman and V. Öwall, "Implementation of a Scalable Matrix Inversion Architecture for Triangular Matrices," *Proceedings of PIMRC'03,* Beijing, China, September 2003.

[43] W. Givens, "Computation of Plane Unitary Rotations Transforming a General Matrix to Triangular Form," *J. Soc. Indust. Appl. Math.*, Vol 6, No. 11958.

[44] R. Döhler, "Squared Givens Rotation," *IMA Journal of Numerical Analysis*,11, p.1-5, 1991.

[45] D. Bindel, J. Demmel, W. Kahan, and O. Marques, "On Computing Givens Rotations Reliably and Efficiently,"*ACM Transactions on Mathematical Software*, 28(2):206–238, 2002.

[46] F. Edman and V. Öwall, "Implementation of a Highly Scalable Architecture for Fast Inversion of Triangular Matrices," *Proceedings of ICECS'03,* Sharjah, United Arab Emirates, December 2003.

[47] F. Edman and V. Öwall, "Implementation of a Full Matrix Inversion Architecture for Adaptive Antenna Algorithms," *Proceedings of WPMC'04,* Abano Terme, Italy, September 2004.

[48] Z. Guo, F. Edman, P. Nilsson, and V. Öwall, "On VLSI Implementations of MIMO Detectors for Future Wireless Communications," *Proceedings of IST-MAGNET Workshop,* Shanghai, November 2004.

[49] F. Edman and V. Öwall, "A Scalable Pipelined Complex Valued Matrix Inversion Architecture," *Proceedings of ISCAS'05,* Kobe, Japan, May 2005.

[50] F. Edman and V. Öwall, "Compact Matrix Inversion Architecture Using a Single Processing Element," *Proceedings of ICECS'05,* Gammarth, Tunisia, December 2005.

[51] L. H. Sibul and A. L. Fogelsanger, "Application of Coordinate Rotation Algorithm to Singular Value Decomposition," *IEEE Int. Symp. Circuit and Systems*, 1984.

[52] R.P. Brent, F.T. Luk and C. VanLoan, "Computation of singular value decomposition using mesh-connected processors". *J. VLSI, Comput. Sys. Vol. 1 no 3*, 1985.

[53] B. Yang andJ. F. Bohme, "Reducing the computations of the singular value decomposition array given by Brent and Luk", *SIAM J. Matrix Anal. Appl. Vol 12*, Oct., 1991.

[54] J.S. Walther, "A unified Algorithm for elementary functions". *Proc. AFIPS Spring joint Computer conference*, 1971.

[55] N. D. Hemakura, *A Systolic VLSI Architecture for Complex SVD*. Ph. D thesis, Rice University, Huston, Texas, 1991.

[56] J. Capon. “High-Resolution Frequency-Wavenumber Spectrum Analysis,” *Proc. IEEE*, 57(8):2408-1418, Aug. 1996.

[57] H. Krim and M. Viberg. “Two Decades of Array Signal Processing Research,” *IEEE Signal Processing Magazine*, July 1996,pp. 67-94.

[58] M. S. Bartelett. “Smoothing Periodigrams from Time Series with Continious Spectra,” *Nature*, 161:686-687, 1948.

[59] T. Shan, M. Wax, T. Kailath, ”On Spatial Smoothing for Direction-of-Arrival Estimation of Coherent Signals,” *IEEE Transactions on Acoustic Speech and Signal Processing*, Vol ASSP-33, No.4, pp 806-811.

[60] F. Edman and V. Öwall, “A Computational Platform for Real-time Channel Measurements using the Capon Beamforming Algorithm,” *Proceedings of WPMC’05,* Aalborg, Denmark, September 2005.

[61] F. Edman and V. Öwall, ”A Compact Real-time Channel Measurement Architecture Based on the Capon Beamforming Algorithm,” *Proceedings of DSPCS'05 and WITSP'05,* Nosa Heads, Australia, December 2005.

[62] “MathWorks”, http://www.mathworks.com, 2006-01-01.

[63] “Mentor Graphics”, http://www.model.com, 2006-01-01.

[64] “Synplicity”, http://www.synplicity.com, 2006-01-01.

[65] “Xilinx”, http://www.xilinx.com, 2006-01-01.

[66] B. Zeidman, *Design with FPGAs & CPLDs*. CMP Books, USA, 2002.

[67] C. Maxfield, *The Design Warrior’s Guide to FPGAs*. Elsevier, USA, 2004.

[68] “Xilinx System Generator”, http://www.xilinx.com/ise/optional_prod/system_ generator.htm 2006-01-01.

[69] S. Sutherland, S. Davidmann and P. Flake, *SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling*. Kluwer Academic Publishers, 2004.

[70] S. Sjöholm and L. Lindh. *VHDL för konstruktion*. Studentlitteratur, Lund, 2003

[71] “The Electronic Design Automation (EDA) and Electronic Computer-Aided Design (ECAD) one-stop resource on the WWW”, http://www.eda.org/, 2006-01-01.

[72] “Xilinx ISE Foundation tool”, http://www.xilinx.com/ise/logic_design_prod/ foundation.htm, 2006-01-01.

[73] “Xilinx Chipscope Pro”, http://www.xilinx.com/ise/optional_prod/cspro.htm, 2006-01-01.

[74] “Concurrent Versions System”, http://www.nongnu.org/, 2006-01-01.

[75] “Subversion”, http://subversion.tigris.org, 2006-01-01.

[76] “Version Control System Comparison”, http://better-scm.berlios.de/comparison /comparison.html, 2006-01-01.

[77] P. Hung, H. Fahmy, O. Mencer and M. J. Flynn, “Fast Division with a Small Lookup Table,” *Asilomar Conference on Signals, Systems and Computers, vol. 2, pp. 1465–1468, 1999.*

[78] C. N. Lyu and D. Matula, “Redundant Binary Booth Recording,” *Proc. of IEEE Symp. on Computer Arithmetic*, 1995.