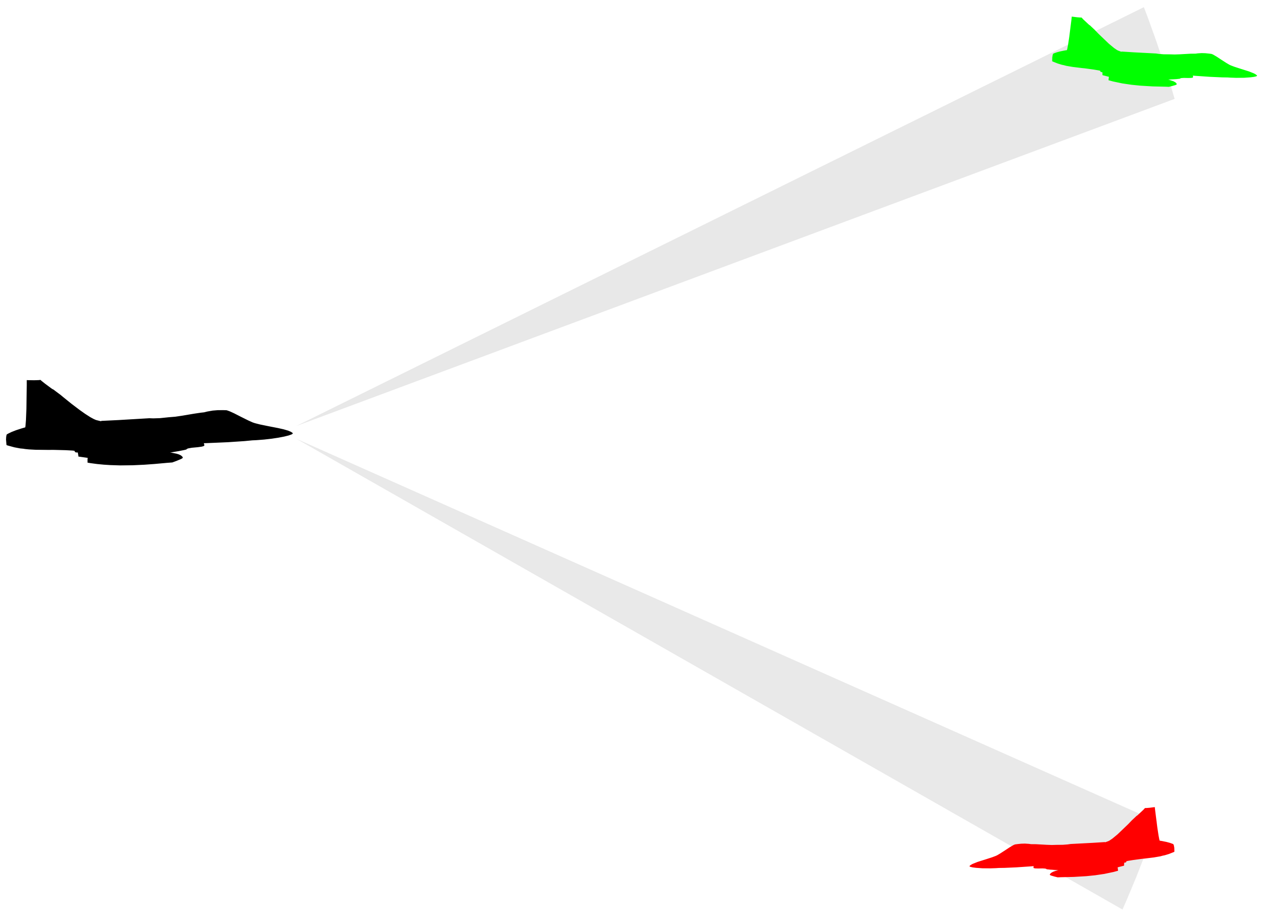
CHALMERS



Space-Time Adaptive Processing in FPGA

Master of Science Thesis in Embedded Electronic System Design

SABINA FRIBERG
PER PÅLSSON

Chalmers University of Technology
Department of Computer Science and Engineering
Gothenburg, Sweden, November 2012

Space-Time Adaptive Processing in FPGA
Master of Science Thesis in Embedded Electronic System Design

Sabina F. Friberg, Per C. T. Pålsson

Examiner: Lars Bengtsson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover: Illustration of a fighter aircraft tracking multiple targets.

Department of Computer Science and Engineering
Göteborg, Sweden November 2012

Abstract

Space-Time Adaptive Processing (STAP) enables very high performance radar processing but comes at a high price of computational requirements and can reach up to hundreds of TFLOPS. This makes it difficult to implement for limited spaces with low power consumption.

This thesis investigates the possibility to implement STAP in an FPGA with focus on the detection algorithm known as Kelly's Generalised Likelihood Ratio Test (GLRT). One of the main goals of the implementation was scalability and parallelism since the technology of the present time is not power efficient enough. A solution that is scalable and can utilize parallelization is possible to distribute over a larger device when the technology is present. Another goal was the comparison of fixed and floating point number representation in terms of performance and power.

The final design was implemented on a Xilinx Virtex-7 FPGA for both single precision floating point and 32 bit fixed point number representation. Three different design solutions were implemented. The final design resulted in a performance of 23.2 GFLOPS/W for the floating point design, 34.3 GFIOPS/W for the fixed point implementation using IP cores and 39.3 GFIOPS/W for the pipelined solution. Existing performance results from NVIDIA GTX 260 GPU the performance is 5.1 GFLOPS/W and for the FPGA co-processor Anemone the number is 19.2 GFLOPS/W.

The solution is scalable and the conclusion is that it is likely that an FPGA solution would be suitable for STAP when the technology exist. However, the support for floating point in the tools need further development to be competitive with the fixed point implementations.

Keywords: STAP, FPGA, VHDL, Radar, Kelly's GLRT.

Acknowledgments

We would like to thank our supervisors at Saab AB, Anders Åhlander and Kent Abrahamsson and our supervisor at Chalmers University of Technology, Lars Bengtsson for their help and feedback throughout the project. Also, we would like to thank Anders Bergåker for technical support.

List of Abbreviations

AESA	Active Electronially Scanned Array
ATC	Air Traffic Control
CFAR	Constant False Alarm Rate
CPI	Coherent Processing Interval
DSP	Digital Signal Processing
FF	Flip-Flop
FIOPS	Fixed point Operations/Second
FLOPS	Floating point Operations/second
GLRT	Generalized Likelihood Ratio Test
GPU	Graphic Processor Unit
LUT	Look Up Table
MSA	Mechanically Scanned Antenna
MTI	Moving Target Indicator
PRF	Pulse Repetition Frequency
RAM	Random Access Memory
RF	Radio Frequency
ROM	Read Only Memory
STAP	Space-Time Adaptive Processing
VHDL	Very High Speed Hardware Description Language
XST	Xilinx® Synthesis Technology

Contents

1. Introduction

This thesis studies parts of the Space-Time Adaptive Processing (STAP) technique for airborne radar systems. This chapter aims to give an introduction to radar systems and especially STAP and explain why this thesis is of interest, what has been done, how and the parts left out.

1.1 Background

Radar systems are used for detecting and tracking objects by using an antenna to send and receive radio waves. The basic function and design of a radar system is illustrated in Figure 1.1 where the antenna emits a radio wave generated in the transmitter. Between two pulses, the receiver listens for the echo of the signal. A small part of the energy in the transmitted radio wave is reflected off the potential target in the direction of the receiver. This small part that bounces back is to be detected among clutter, jamming and other noise and for this, the incoming signal has to be processed to detect the target.

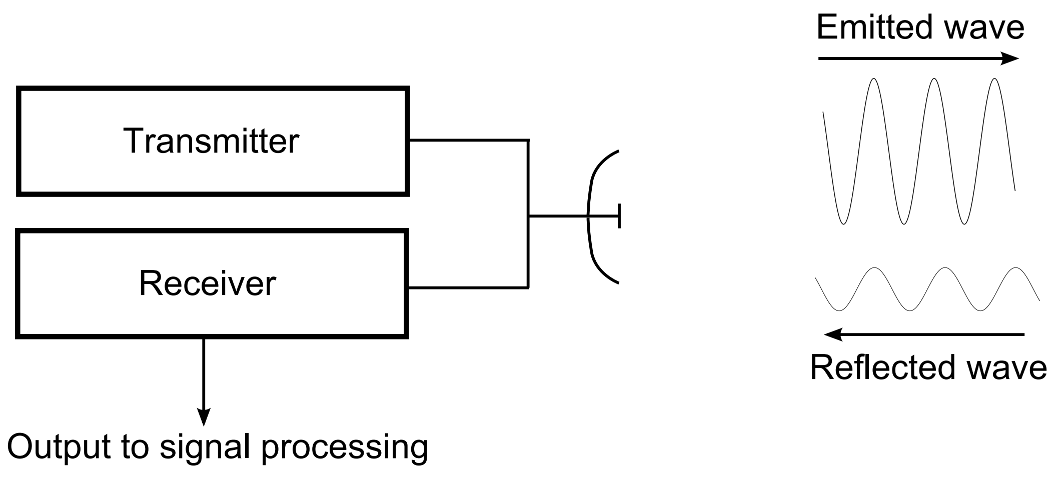


Figure 1.1: Illustration of the basic principle of a radar system.

The history of radar originates in 1886 when Hertz demonstrated the reflection of radio waves but it was not until the 1930s that the radar development accelerated [1]. After World War II the development focused on things not present or completely working in the radar systems during the war, such as the use of the Doppler effect in the moving target indication (MTI) radar and the electronically steered phased array antenna for rapid beam steering [2].

Applications for radar are many: Military, traffic control with radar speed meters, Air Traffic Control (ATC), navigation and ship safety among others [2]. For airborne radar systems the interference of clutter, jamming and noise is an even bigger problem than to the systems on ground, due to that the platform is in motion. To handle this and suppress clutter while detecting the target, advanced signal processing is used. One way to perform this advanced signal processing is to use Space-Time Adaptive Processing (STAP). In STAP the input is sampled in both time and space and correlated to achieve the desired result. When using STAP, the beamforming block is adaptive and calculate weights depending on the inputs to suppress the clutter and jamming and improve detection of targets by steering the main lobe direction. By expanding the radar called Active Electrically Scanned Array (AESA) with STAP, the radar performance can be improved.

The drawback of the STAP system is the enormous amount of calculations needed to provide full functionality. Computational demand in a STAP system is in the range of 1 GFLOPS to 50 TFLOPS, but can easily scale up to 100 TFLOPS and above depending on the system [3]. This makes it hard to implement the radar system using STAP in an energy efficient way while meeting the constraints of the limited space available for the system in an aircraft.

While the algorithms used in STAP are extremely computational heavy, they are also parallelizable making them suitable for implementation on parallel computer systems such as many-core processors or microprocessor arrays [4].

1.2 Purpose

For airborne applications and especially smaller air crafts, some of the issues with using STAP is the limited space and power consumption. A radar system using STAP is required to fit in the same space as the current radar systems and also consume a reasonable amount of power. For this to be achievable a specialized signal processing solution is required.

The technology available at present time is not able to meet the performance requirements for the system. With the evolution of computer systems in mind, it is most likely that the required technology will exist in the near future.

The purpose of this thesis project was to investigate the possibility to parallelize and implement STAP algorithms using FPGAs. The goal was to reason about the suitability of FPGA for implementation of STAP. This was done by suggesting a scalable design by implementing parts of STAP. The design tests the implementation of fixed point and floating point number representation and for different technologies. In addition, this thesis also examined how suitable FPGAs are for implementation of a STAP system compared to other devices such as many-core processors, graphic processor units (GPU) and microprocessor arrays.

1.3 Scope and Limitations

The thesis limits the implementation to parts of the STAP, focusing on some of the algorithms. The evaluation is based on the suitability to use an FPGA for partial implementation of the signal processing system. The work of this master thesis project is limited to look at one particular detection algorithm used in STAP, namely Kelly's Generalized Likelihood Ratio Test (GLRT). Other detection algorithms have not been studied and evaluated for this kind of implementation. All data and input signals are floating or fixed point numbers represented with a maximum of 32 bits. Double precision floating point number representation is not handled in this thesis.

The theoretical system was considered a very high performance system intended for a smaller aircraft, why the assumption was made that computational performance required approximately 100 TFLOPS. Also due to the limited space in the aircraft, power consumption was assumed to be limited. A reasonable assumption was that the power consumption should not exceed that of a high performance home computer, approximately 400 W.

1.4 Method

To fulfill the goals of this thesis, a thorough literature study was done to provide good basic knowledge of radar system as well as the relevant theory for the algorithm. The literature study was followed by an analysis of Kelly's GLRT detection algorithm to isolate the most computational parts to evaluate the suitability for parallelization.

After determining which parts of the algorithm that would benefit the most from parallelization, an implementation in VDHL was designed. The design was implemented by using a bottom-up approach for the ability to eliminate bottlenecks in an early stage as well as individual component testing and verification. The solutions were designed for both fixed point and floating point number representation and implemented for two target FPGAs, Virtex-5 and Virtex-7, focusing on the Virtex-7 device. This resulted in three individual designs created from the same structure. Two tools were used for synthesis, Synplify Pro and Xilinx ISE XST and the design suite Xilinx ISE was used for mapping, placement and routing of the hardware. Xilinx's XPower was used to estimate power consumption. A testbench was built and used to verify the designs.

The results were evaluated and fixed point was compared to floating point to give the most efficient design. The different technologies were also compared to one another.

The structure of the report follows the work method with some basic radar theory and signal processing for radar systems, followed by the mathematical theory behind the STAP algorithms. The algorithms are further analysed in chapter 4 to parallelize the different parts of the algorithm. The following part of the report is the implementation and design work and the corresponding result. In the end, the results are compared to other parallel solutions and discussed even further in the discussion chapter. The report ends with the conclusion and suggestion for future work.

2. Radar Fundamentals

This chapter deals with the relevant theory of this thesis. The first part describes basic radar theory to give a basic knowledge of radar systems used today and also to provide background information about how the Active Electrically Scanned Array (AESA) antennas operates in a radar system. It is briefly described how to steer the beam by Digital Adaptive Beamforming (adaptive DBF). The second part of the chapter describes signal processing for such radar systems, both in the conventional way but also elaborates on Space-Time Adaptive Processing (STAP).

2.1 Basic Radar Theory

The word radar means, and is originally an acronym for, radio detection and ranging. Radar systems use radio waves to detect objects and determine distance as well as relative speed. The radar system antenna emits pulses of radio waves that are reflected by the object. Some of that energy is reflected back to the radar system. This energy is processed and used to determine both distance to the object and a potential direction and velocity if the object is in motion. The radar system can be designed in many different ways, using different types of antennas that can be controlled and steered in different ways.

2.1.1 Range and Velocity Determination

The amount of energy that is reflected back to the radar system is given by equation (2.1).

$$P_{echo} = \frac{P_m A_e \sigma G}{16\pi^2 R^4} \tag{2.1}$$

P_m is the power of the transmitted wave, A_e is the effective aperture area of the antenna in square meters, G is the antenna gain, σ is the radar cross section of the target in square meters and R is the range [1][2]. This equation is known as the radar equation or the radar range equation and as seen, the energy received from the echo is attenuated $\sim \frac{1}{R^4}$. Even though the radar equation is quite complex, the determination of the distance to an object is following a simple formula. The radio wave in air basically has the same velocity as light. The range or distance to an object can therefore be calculated from equation (2.2) below.

$$R = \frac{ct}{2} \tag{2.2}$$

Where R is the range, c is the velocity $(3 * 10^8 m/s)$ and t is the roundtrip time interval between a transmitted wave and the reflected wave. The range information stored in the reflected wave can be illustated according to Figure 2.1.

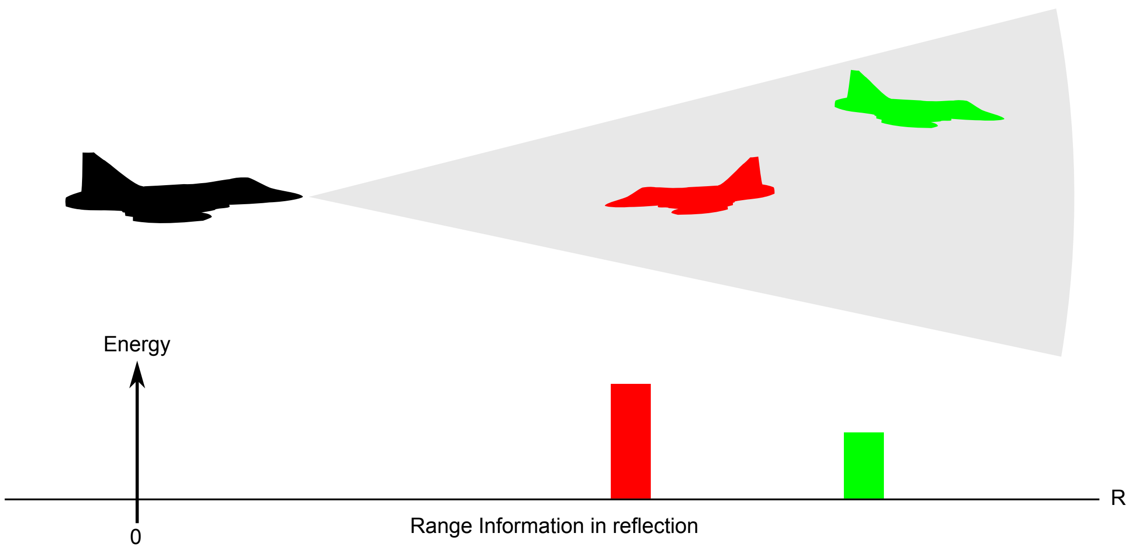


Figure 2.1: Energy and range information from two different moving targets.

However, the number of pulses transmitted per second, called the pulse repetition frequency (PRF) might give a misleading range, indicating that an object is closer to the system that it really is. This happens when an echo from the object arrives after a new pulse has been transmitted. The PRF gives the system a maximum distance to assure that the echo does not arrive after a new pulse has been transmitted. This maximum distance is determined by equation (2.3) [2][5].

$$R_{max} = \frac{c}{2PRF} \tag{2.3}$$

Objects that appear in the radar beam within that range are to be detected even when the object might be in motion. In that case, it is of interest to determine the direction and velocity of that object. This is done using the Doppler effect. The frequency of the reflected wave is altered depending on the velocity and direction of the target in motion. The result from this measurement is the velocity of the object relative to the radar system. If an object is moving away from the radar system, the reflected signal contains a lower frequency compared to the signal emitted from the radar and if the object is moving towards the system the frequency increases. This is briefly illustrated in Figure 2.2.

2.1.2 Active Electronically Scanned Array (AESA)

A radar system can use different antennas depending on it purpose and it can consist of one antenna element or arrays of many, smaller antenna elements. The latter can be divided into ones that share a single transmitter and receiver source and the ones where every single antenna element has an individual source. The radar can control its beam by mechanically moving parts or by electrically steering the beam by controlling and shifting the phase and amplitude of the wave. Electrically steering the beam makes the system more complex, but gives the possibility to change direction of the beam very fast without moving any parts. AESA is the type of radar that uses a large number of antenna elements arranged as a planar array and is also known as the active phased array radar. The antenna elements are steered by adjusting the phase and amplitude of the waves [6]. This method of steering the antenna elements electronically results in no mechanical motion, which makes it possible to change the direction of the beam very fast

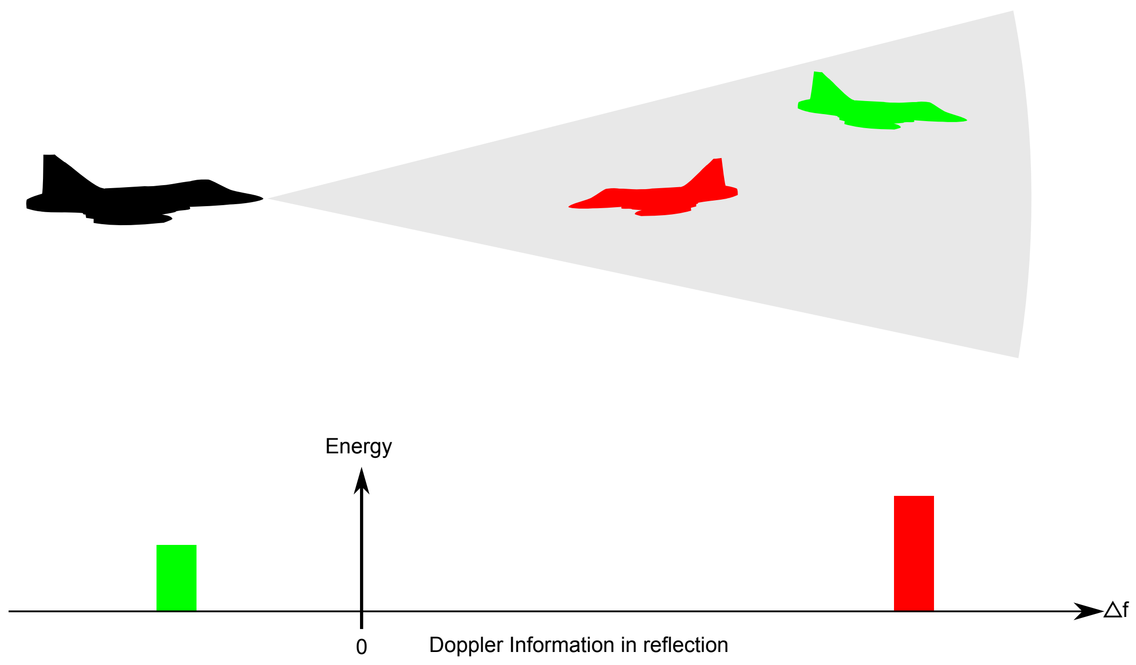


Figure 2.2: Energy and doppler information from two different moving targets.

compared to mechanically scanned arrays (MSA). Each antenna element has an individual transmitter and receiver and the beam is steered by adjusting the relative phases of the pulses sent out from different antenna elements. The main lobe direction is where they interfere constructively with each other. This is illustrated in Figure 2.3. The active part

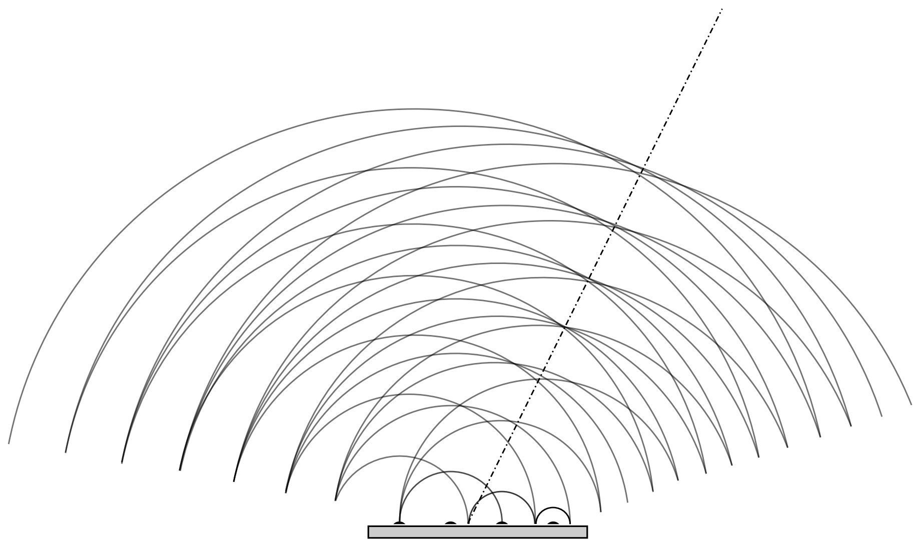


Figure 2.3: Illustration of AESA beam forming.

of the AESA means that not only does the antenna elements have individual transmitters and receivers, they also have individual radio frequency (RF) sources [2]. This makes each element capable of generating and emitting its own independent signal and divide the beam to a number of sub-beams if desired. With this structure every single antenna element is basically a miniature radar system.

2.2 Signal Processing for Radar

Radar signal processing is a vital part of the radar system. Signals reflected from the targets are attenuated and the presence of jamming and clutter further distort the signal. Figure 2.4 describes the signal processing chain in a simplified way, showing the most important steps from signal reception to detection. This section briefly describes these signal processing steps.

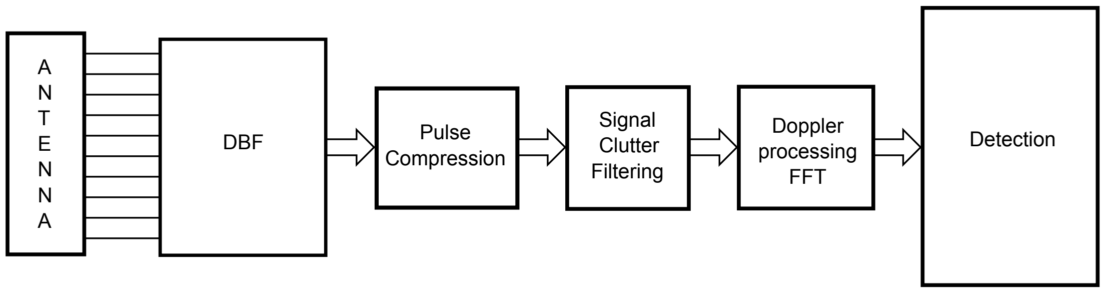


Figure 2.4: The signal processing chain for a radar system in a simplified way.

2.2.1 Digital Adaptive Beamforming

The steering of the radar beam, called beamforming is an other name for spatial filtering where the antennas can be steered by digital or analog signal processing. The analog method was both expensive and fragile due to the components used, so modern systems usually performs beamforming digitally [6].

To steer the beam the beamformer controls phase and amplitude of the antenna elements. This can be done in an adaptive way, where the received signals are used to calculate so called weights that makes it possible to follow a target and steer the beam in the desired direction. The digital adaptive beamforming is illustrated in Figure 2.5.

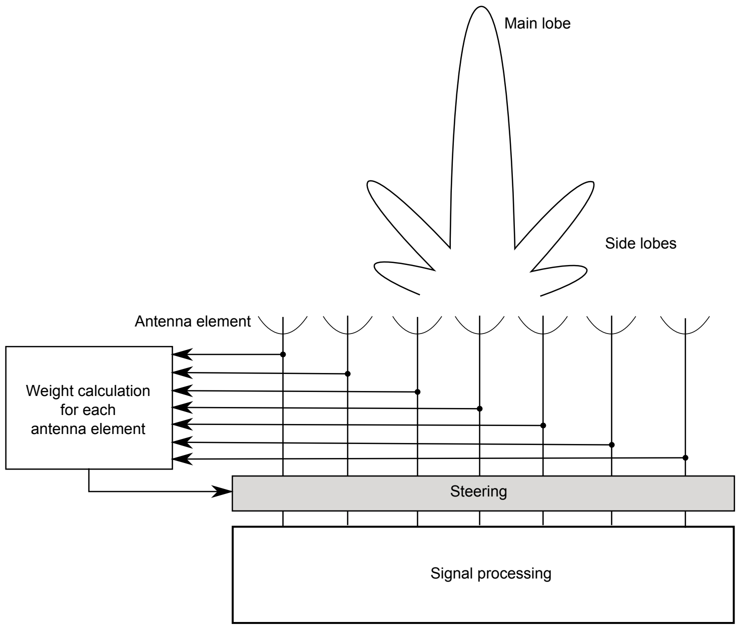


Figure 2.5: Adaptive beamforming using AESA and STAP.

2.2.2 Pulse Compression

Since the radar signal is attenuated $\sim \frac{1}{R^4}$, which can be seen in (2.1), one of the essential parts of the signal processing chain is pulse compression in order to obtain a pulse with higher energy density. In the pulse compression step, the received signal is filtered through a matched filter which achieves the highest response when the received signal matches the transmitted signal. Figure 2.6 illustrates the received signal before and after the pulse compression. The main purpose of the pulse compression is to take all energy from a target and store in a single range bin.

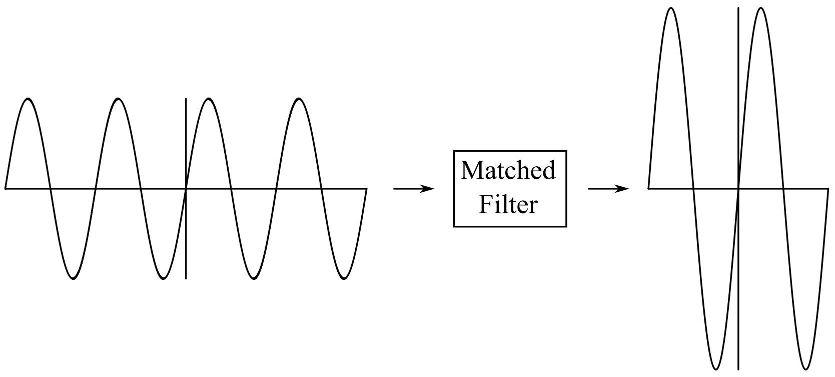


Figure 2.6: Pulse compression through matched filter.

2.2.3 Clutter Filter

The clutter filter is used to attenuate clutter caused by reflection against the ground etc. Figure 2.7 illustrates an implementation of a MTI-filter used for clutter reduction. The

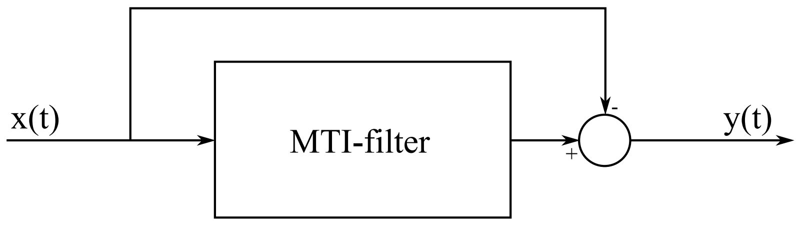


Figure 2.7: MTI-filter for clutter reduction.

filter is simply a high-pass filter that removes frequencies at and near the zero-Doppler frequency.

2.2.4 Doppler Processing

The Doppler processing consist of transforming the pulse-samples (slow time) into Doppler channels. The reason is to determine the relative speed of a potential target compared to the platform. The doppler effect changes the frequency of the signal received by the antennas depending on direction and speed of the target. The doppler effect combined with a filter bank contaning digital FIR filters results in the desired information [7]. This is done using the Discrete Fourier Transform (DFT) which can be implemented with the computational efficient Fast Fourier Transform (FFT). The DFT/FFT tranforms the pulses into Doppler channels according to Figure 2.8.

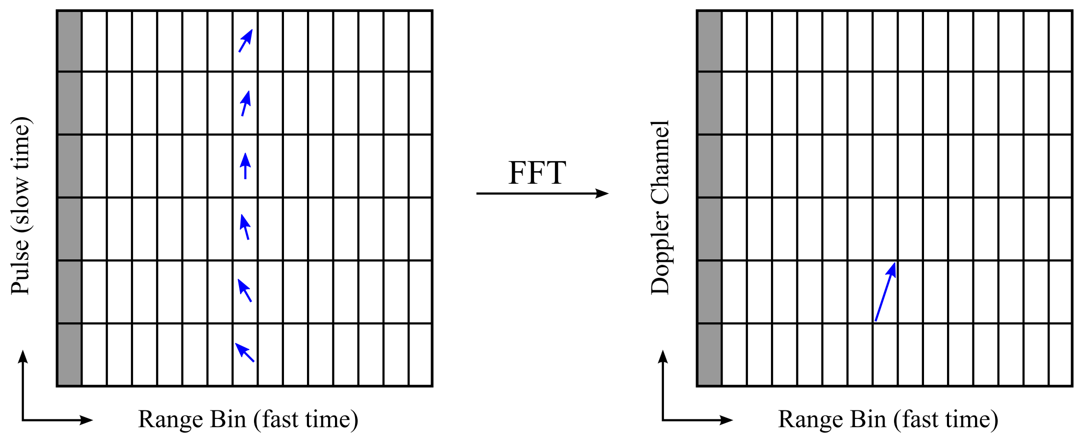


Figure 2.8: Transformation of Pulse to Doppler channel.

As illustrated by Figure 2.8 the Doppler processing improves the signal to noise ratio. This is due to the fact that the phase shift of the pulses in slow time is added constructively while the noise is added in random in the Fourier transform.

2.2.5 Detection

Techniques for detection of a target can be done in many ways. A common process is Constant False Alarm Rate (CFAR) detection. CFAR refers to a detection process that estimate data placed nearby a cell to calculate a threshold used to determine if a target is detected in the cell studied [1]. If the cell contains a value greater than the threshold it might contain a target. Traditional CFAR is not adaptive, but designed with the assumption of a constant noise and interference level. Especially when the platform is in motion, the CFAR need to be adaptive to provide useful information.

2.3 Space-Time Adaptive Processing

The principle of STAP is much like the signal processing described in the previous section, but in STAP the received signals are being processed in a space-time clutter filter. This filter is adaptive to remove the clutter from the signal in the most suitable way for that particular signal. The signal continues to a matched space-time filter that uses the adaptive DBF cascaded with a doppler filter bank that process the signal used in a test algorithm. The algorithm compares the output to a detection threshold to indicate when a target is detected [8].

The main difference between the traditional and STAP radar systems is therefore the algorithms for filtering the input signals and the adaptive feedback for weight calculations to optimize detection and suppress clutter, jamming and other noise.

Radar systems implemented using AESA radars apply the phase shift of each individual antenna element and sums the input signals providing one single input signal. The STAP processing keeps all the signals separated instead to provide statistical data to effectively filter out clutter and jamming signals.

There are a number of different adaptive detectors that can be used in STAP, each with its own detection algorithm. In the literature, 4 main detectors are mentioned apart from CFAR. They are the Adaptive Matched Filter (AMF), Kelly's Generalized Likelihood Ratio Test (GLRT), Adaptive Cosine Estimator (ACE) and Adaptive Sidelobe Blanker

(ASB) [9]. All 4 uses an estimated space-time covariance matrix in the detection algorithm. This thesis is limited to study only Kelly's GLRT.

In a STAP system, the signals are sampled in both space and time domains and performing the signal processing in this way is powerful. The suppression of jammer, noise and clutter can be even more effective when using the correlations between the two domains [10]. The benefits gained in STAP has a drawback in the computational load that needs a solution before a STAP system can be implemented and used in an airborne platform.

The following chapter describes the theory for STAP and Kelly's GLRT further and focuses on the critical parts of the signal processing chain.

3. Algorithm introduction for STAP

This chapter further discusses the mathematical theory used in the computational algorithms of STAP and explains the different steps of Kelly's GLRT detection algorithm. First a thorough description of Kelly's GLRT is presented, followed by a description of the estimated covariance matrix used in STAP.

3.1 Kelly's Generalized Likelihood Ratio Test (GLRT)

The GLRT algorithm utilizes statistics to attenuate clutter, jamming and noise signals. For each new sample $\boldsymbol{x}$, the algorithm is calculated with the corresponding estimated covariance matrix $\hat{\boldsymbol{R}}^{-1}$ in conjunction with all steering vectors $\boldsymbol{s}$. Due to mathematical manipulation in combination with the number of spatial channels these calculations result in high demand of computational performance. The immense amount of data results in computational needs that are very large in comparison to regular radar processing used today. Kelly's GLRT algorithm is defined by (3.1) and is a function of the covariance matrix $\hat{\boldsymbol{R}}^{-1}$ described further in chapter 3.2.

$$t_{Kelly} = \frac{\left|\boldsymbol{s}^H \hat{\boldsymbol{R}}^{-1} \boldsymbol{x}\right|^2}{\left(\boldsymbol{s}^H \hat{\boldsymbol{R}}^{-1} \boldsymbol{s}\right)\left(K + \boldsymbol{x}^H \hat{\boldsymbol{R}}^{-1} \boldsymbol{x}\right)} \tag{3.1}$$

Analysing parts of the algorithm, the covariance matrix $\hat{\boldsymbol{R}}^{-1}$ is a square matrix of the dimensions $M \times M$, the steering vectors $\boldsymbol{s}^H$ is a $M \times 1$ vector and $\boldsymbol{s}$ and $\boldsymbol{x}$ a $1 \times M$ vector. K is the number of adjacent range bins used for covariance matrix estimation. The detector algorithm consists of the three computations below

$$\boldsymbol{s}^H \hat{\boldsymbol{R}}^{-1} \boldsymbol{s} \tag{3.2}$$

$$\left|\boldsymbol{s}^H \hat{\boldsymbol{R}}^{-1} \boldsymbol{x}\right|^2 \tag{3.3}$$

$$K + \boldsymbol{x}^H \hat{\boldsymbol{R}}^{-1} \boldsymbol{x} \tag{3.4}$$

3.2 Covariance Matrix

One of the important steps in STAP signal processing include the calculation of the inverse covariance matrix, $\hat{\boldsymbol{R}}_{sp}^{-1}$, which is used to filter the received signals. The matrix is estimated using test statistics obtained from the sampled signals.

The covariance matrix is estimated by weighting the matrix product of the input signal vectors $\boldsymbol{x}$ multiplied with their hermitian conjugate $\boldsymbol{x}^H$, for all so called snapshots r_0. The resulting matrix is the sum of the products for all N snapshots from the input vector and a number of adjacent input vectors, described in (3.5).

$$\hat{\boldsymbol{R}}_{sp} = \frac{1}{N} \sum_{r=r_0}^{r=r_0+N-1} \boldsymbol{x}_r^H \boldsymbol{x}_r \tag{3.5}$$

After forming of the covariance matrix it is inverted followed by forming a matrix that is the Kronecker product of the inverted covariance matrix and an identity matrix, described in (3.6) and (3.7). The Kronecker product forms a matrix $\hat{\boldsymbol{R}}^{-1}$ that is a square matrix of dimensions $M \times M$ which matches the length M of the steering vectors $\boldsymbol{s}$ and samples $\boldsymbol{x}$ and is illustrated in Figure 3.1. This is the matrix used in Kelly's GLRT described in the previous section.

$$\hat{\boldsymbol{R}}_{sp} \rightarrow \hat{\boldsymbol{R}}_{sp}^{-1} \tag{3.6}$$

$$\hat{\boldsymbol{R}}^{-1} = \hat{\boldsymbol{R}}_{sp}^{-1} \otimes \boldsymbol{I}_L \tag{3.7}$$

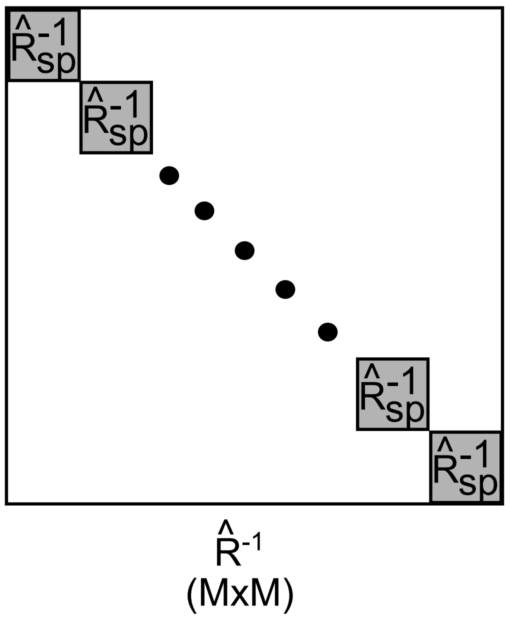


Figure 3.1: The resulting matrix $\hat{\boldsymbol{R}}^{-1}$, consisting of smaller matrices placed diagonally in the larger matrix.

4. Algorithm Analysis

This chapter examines possible ways to parallelize different parts of the algorithm. The algorithm is fortunately very parallelizable and uses relatively few input values for output calculation, which limits memory bandwidth when implementing for an FPGA since most of the data can be kept on chip. For the implementation to be applicable and functional it needs to be scalable as well as able to efficiently use most of the available resources on the chip. The investigations are made to make the most suitable choice of the parts that are intended for implementation.

4.1 Kelly's Generalized Likelihood Ratio Test

Looking at the equations in chapter 3.1, it is obvious that Kelly's GLRT detection algorithm is very parallelizable. However, the calculations need to be done for every combination of data input samples $\boldsymbol{x}$ and steering vectors $\boldsymbol{s}$. In addition to this, the covariance matrix needs to be updated with a certain frequency according to (3.5) in section 3.2.

4.1.1 Matrix Calculations - $\boldsymbol{s}^H \hat{\boldsymbol{R}}^{-1}$

Since the matrix $\hat{\boldsymbol{R}}^{-1}$ is the Kronecker product of the $N \times N$ covariance matrix $\hat{\boldsymbol{R}}_{sp}^{-1}$ and the LxL identity matrix $\boldsymbol{I}_L$, the elements of the resulting vector $\boldsymbol{s}^H * \hat{\boldsymbol{R}}^{-1} \equiv \boldsymbol{z}$ are the vector-matrix products of N_{sp} long vectors multiplied with with the $N_{sp} \times N_{sp}$ sized matrix $\hat{\boldsymbol{R}}_{sp}^{-1}$. This is due to the fact that all elements in the $\hat{\boldsymbol{R}}^{-1}$ are zero outside of the diagonally spaced covariance matrix $\hat{\boldsymbol{R}}_{sp}^{-1}$. Because of this it is possible to divide the vectors $\boldsymbol{z}$ and $\boldsymbol{s}^H$ into L vectors of length N_{sp} as illustrated by Figure 4.1 and equations (4.1),(4.2), where each of the vectors $\boldsymbol{z}_i$ is calculated according to (4.3).

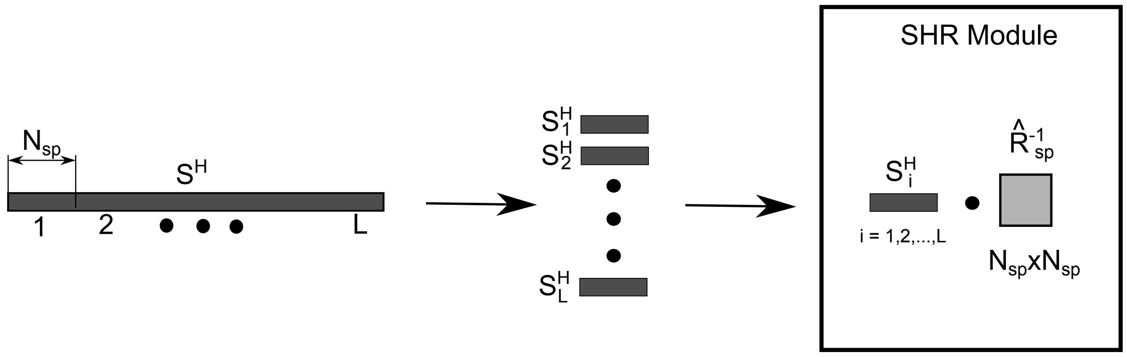


Figure 4.1: Dividing the steering vector and perform smaller matrix multiplications.

$$\boldsymbol{s}^H = \Big[\ [\boldsymbol{s}_0^H] \ \ ... \ \ [\boldsymbol{s}_i^H] \ \ ... \ \ [\boldsymbol{s}_{(L-1)}^H] \ \Big] \tag{4.1}$$

$$\boldsymbol{z} = \left[\ [\boldsymbol{z}_0] \quad ... \quad [\boldsymbol{z}_i] \quad ... \quad [\boldsymbol{z}_{(L-1)}]\ \right] \tag{4.2}$$

$$\boldsymbol{z}_i = \boldsymbol{s}_i^H \hat{\boldsymbol{R}}_{sp}^{-1} \tag{4.3}$$

This makes it possible to divide the calculation of (4.2) into a subset of matrix calculations for the partial vectors $\boldsymbol{s}_i^H$, where each of the vectors are multiplied with the covariance matrix. By dividing the calculation of an entire matrix in this way, it is possible to parallelize and reuse data by performing several calculations using the same covariance matrix in parallel.

Another way to parallelize the calculations are by performing several of the multiplications for one single matrix at the same time, either by multiplying one row of the matrix with the corresponding element in the vector or multiplying one vector column by column in the matrix.

Using the first approach would be to focus on one row in $\hat{\boldsymbol{R}}_{sp}^{-1}$ and multiplying with the corresponding element in $\boldsymbol{s}_i^H$ at the same time. After M iterations all multiplications are done and the resulting vector is calculated by adding the product elements from the same column in $\hat{\boldsymbol{R}}^{-1}$. The N_{sp} first elements in $\boldsymbol{z}$ will be presented at the same time, and a new block of N_{sp} elements will arrive after N_{sp} iterations of multiplying and adding the products together.

If the parallelization instead is designed after the columns in $\hat{\boldsymbol{R}}_{sp}^{-1}$, the multiplication can be performed for all elements of a vector $\boldsymbol{s}_i^H$ in parallel, performing all multiplications needed for an element in $\boldsymbol{z}$. This results in a new element ready each iteration consisting of the N_{sp} multiplications followed by accumulation the products. The resulting vector $\boldsymbol{z}$ needs M iterations using this method. The only thing that differ is that a new element is calculated each iteration. Figure 4.1 illustrates how this matrix multiplication can be divided by performing subsets of multiplications.

Using either one of the two methods described above, it is possible to parallelize it even further. This can be done by performing all multiplications done to the same column or row, at the same time, which would correspond to the entire vector matrix multiplication in parallel.

4.1.2 Remaining parts of Kelly's GLRT

The products of $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}\boldsymbol{s}$ and $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}\boldsymbol{x}$ result in a scalars. The multiplications are possible to perform in parallel, but are not performance critical operations for the algorithm since the computational load for these operations are not that heavy compared to the $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}$ described in section 4.1.1.

The last part of the GLRT algorithm is $K + \boldsymbol{x}^H\hat{\boldsymbol{R}}^{-1}\boldsymbol{x}$. All steering vectors are to be used for every single input data $\boldsymbol{x}$, why this step is only to be calculated one time and then can be kept constant for the rest of the calculations. This is a result from having a large number of steering vectors that all shall be applied on the input data. The resulting product of this calculation is also a scalar due to multiplication with the conjugate.

4.2 Covariance Matrix

The covariance matrix $\hat{\boldsymbol{R}}_{sp}$ and the inverted matrix $\hat{\boldsymbol{R}}_{sp}^{-1}$ is described in section 3.2. Each matrix product constituting the final matrix $\hat{\boldsymbol{R}}_{sp}$ is simple to parallelize since it is simply

$N \times N$ products of $N+N$ input values according to equation (3.5). In practice, the estimation of the covariance matrix is done in blocks of samples $\boldsymbol{x}_r$. Due to a relatively small amount of calculations, this implies that the computational load for the covariance matrix and its inverse is quite small compared to the following steps of STAP and thus, is not further analyzed in this thesis.

4.3 Fixed and Floating Point Number Representation

The choice of fixed or floating point number representation is also of importance when working with digital signal processing. When representing the data in a specific number of bits, this may result in some differences that the designer needs to be aware of. Floating point arithmetics provide much higher dynamic range and precision, especially for small numbers, compared to fixed point. Due to the "fixed" placement of the decimal point in the use of fixed point, the gaps between two adjacent numbers can be larger compared to floating point. When performing calculations in the digital signal processing the result must be rounded to the nearest value in the number representation used. This yields for larger rounding errors when using the fixed point number representation [11].

5. Hardware and Tools

Depending on the target hardware and the tools used for implementation, the results from the same design can differ. Therefore, it is of importance to have knowledge about the tools and technology used. There are a number of vendors on the FPGA market and this chapter describes the target hardware for this design as well as the tools used for synthesizing and mapping the design onto the specific target hardware.

5.1 Target Hardware

The implementations are synthesized against two target hardware platforms, Xilinx's *Virtex-7* and *Virtex-5*. The Virtex FPGA is Xilinx's top-of-the-line FPGA with version 5 beeing 4-5 years older than version 7. The main differences are as follows

Table 5.1: Comparison of Virtex-5 and Virtex-7 hardware characteristics.

	Virtex-5	**Virtex-7**
# DSP slices	1,056	3,600
# logic slices	37,440	153,000
# 36kb block ram	516	1,500
# user I/O	960	880

The increase of 1,056 DSP slices to 3,600 result in 3 times as many multipliers. The logic slice for Virtex-5 contains 4 Look Up Tables (LUTs) and 4 flip-flops (FFs), while the Virtex-7 device contains 4 LUTs and 8 FFs. This gives an even bigger boost of the number of FFs when going from Virtex-5 to Virtex-7. The actual numbers for LUTs and flip-flops are presented in Table 5.2.

Table 5.2: Comparison of Virtex-5/7 LUTs and flip-flops.

	Virtex-5	**Virtex-7**
# LUTs	149,760	612,000
# FFs	149,760	1,224,000

5.2 Synthesis Tools XST and Synplify Pro

Xilinx provides its own tool for synthesis delivered within ISE, called Xilinx$^{®}$ Synthesis Technology (XST). XST is a tool that synthesizes the VHDL code and creates a Xilinx-specific netlist file. This is done by checking the code for syntax errors, performing a HDL synthesis where the code is analysed and if possible transformed into building blocks or macros (MUXes, RAMs, adders and subtracters)[12]. In this step of the synthesis XST also performs a resource sharing check to reduce the amount of macros and increase the clock frequency while reducing the area. When this is done, a low level optimization is performed. In this step XST transform the design created in the HDL synthesis to a technology-specific implementation. The tool is to be used for Xilinx platforms, such as Virtex-5 and Virtex-7. The drawback of this tool is that the design becomes technology dependent and it is not possible to use this netlist on platforms from other vendors.

If Synopsys Synplify Pro is used for synthesis instead, the result is a design with a multi-FPGA vendor support. Synplify Pro also checks the code for syntax errors before using the Behavior Extracting Synthesis Technology (BEST^{TM}) algorithm to find structures such as RAMs, MUXes and arithmetic operators and turn them into building blocks or macros, just like XST. The design is optimized and finally mapped to technology-specific components. When this is done, it generates a netlist used for placement and routing. The advantage of Synplify Pro is that it it possible to use the same tool for synthesis even when the design is to be mapped onto another technology.

The two tools use different algorithms when performing the synthesis and optimization, and even when the same design is synthesised and to be mapped to the same target platform, the result after synthesis can differ.

5.2.1 Synthesis Attributes

For the ability to control which logic elements are inferred when writing VHDL code, synthesis attributes can be set to force the synthesis tool to use a specific kind of logic element. Synthesis attributes set constraints to force the synthesis tool to optimize and translate a piece of HDL code or a complete design in a specified way. To use the attributes in an efficient way, the designer needs knowledge of both the hardware target as well as the synthesis tool. By using attributes, the designer can specify how the synthesis tool should translate a certain piece of code [13]. If the designer wants to implement a specific component on a DSP slice instead of logic to increase speed the use_dsp48 attribute for XST can be used [14]. For Synplify Pro the attribute is called syn_dspstyle when designing with Xilinx [15]. As an other example Synplify Pro has difficulties inferring shift registers and generates slow logic for Xilinx devices. By setting an attribute to infer a Xilinx SRL primitive this problem can be avoided. Xilinx XST tool can also for instance move registers across connected DSP units in an unfavorable way, which also can be avoided by setting synthesis attributes. By using attributes the designer can optimize the design for a certain purpose and gaining a better result from a tool that usually is not optimal for this design.

5.3 Design Suite Xilinx ISE

MAP and PAR are programs in the design suite Xilinx ISE. The tool supports the Virtex-7 platform and is able to generate bitfiles to this hardware. These programs map the design to the hardware and place and route the design. There are a number of options that the designer need to consider to make the tool map, place and route the design in the most efficient way. In a design where speed is of importance, some of these options can be set to make the tool optimize for speed. An example is the overall effort level that can be set to *high* when running PAR. The runtime during PAR will then increase, but the quality of the result will be better in the aspect of timing. The PAR is set to place and route the design based on timing constraints set in a constraints file. A design with high demands on speed might have a very short period time set as a constraint and one risk with the tool is that it draws the conclusion that it is not possible to meet the constraints. This might lead to the tool basically giving up. If the timing constraints are set to a slightly higher period time, the maximum frequency can be higher because the tool does not give up as easily in that case.

Xilinx's tool XPower which is also a part of ISE has been used for estimation of power consumption.

5.4 DSP Slices, the Basic Building Block for Arithmetics

When designing arithmetic functions in VHDL that are to be implemented on an FPGA, the usual choice of implementation is by using DSP slices. The available DSP slices of an FPGA provide fast computation of both multiplications, addition/subtraction or multiply and accumulate operations. The DSP slices consist of fixed size multipliers and accumulators, so when the number of bits in the operands exceed that of the DSP slice, multiple slices can be cascaded in order to achieve the same functionality.

Figure 5.1 shows the DSP slice present in Xilinx's Virtex-5, which consists of a 25×18 bit signed multiplier and a 48 bit accumulator. Newer Virtex devices contain the improved DSP48E1 slice introducing new features and improved operation frequency.

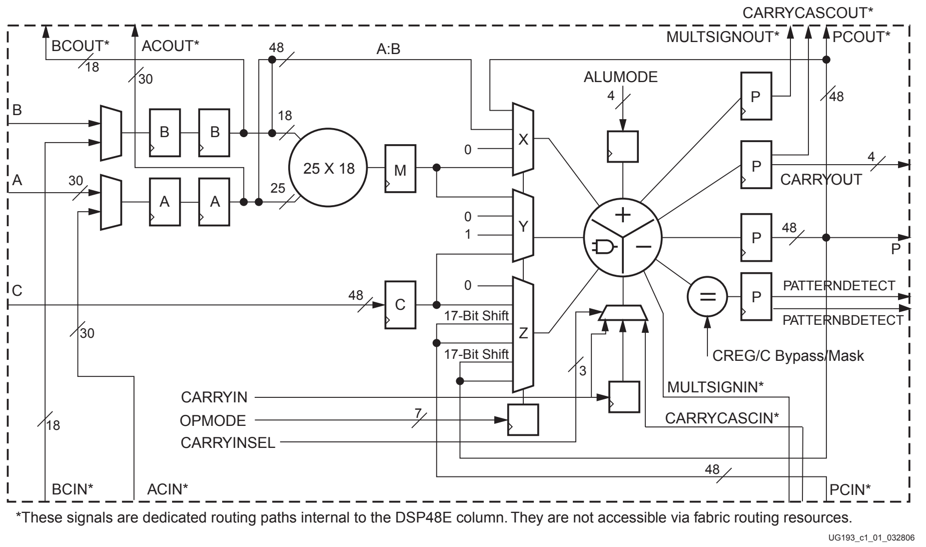
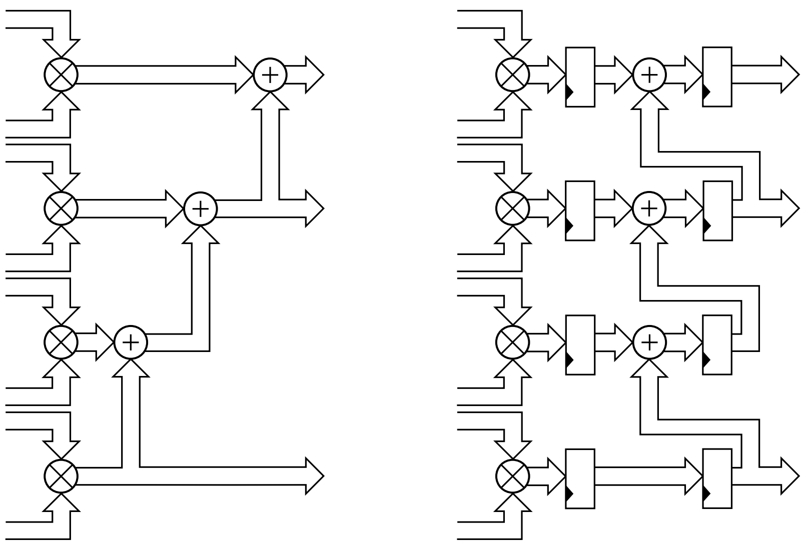


Figure 5.1: DSP48E[1] slice present in the Virtex-5 FPGAs, figure from [16].

For implementation of 32-bit fixed point multiplication using the DSP48E/DSP48E1 slices, a total amount of four slices are used in order to provide the results. Figure 5.2 illustrates the cascaded setup of a 32-bit signed multiplication where Figure 5.2a represents the direct implementation and 5.2b represents the pipelined implementation. Since the direct implementation suffers from a significantly higher path delay, the maximum frequency of the direct form is only about one sixth of the maximum DSP slice capability, while the pipelined form is able to run at full speed.

While the multiplications need pipelining to function at full speed the accumulators in the DSP48E/E1 slices support 48 bit addition/subtraction at full speed, which is sufficient for the 32 bit fixed point numbers.

[1]Xilinx (2012) *Virtex-5 FPGA XtremeDSP Design Considerations, User Guide.*



(a) Direct implementation. **(b)** Pipelined implementation.

Figure 5.2: Simplified illustraion of the implementation of a 32x32 bit multiplication using DSP48E slices.

The first step of design implementation in this thesis resides in the most basic components needed for the math operations in the STAP algorithm. All complex operations in the algorithm consist of additions, subtractions and multiplications so three basic math operations are the only needed components for designing the complex operators.

Since the purpose of this thesis is to “squeeze as much juice as possible” out of the FPGA, a pipelined multiplier is necessary. The IEEE libraries contain VHDL operators for additions and multiplications, both of which are concurrent implementations, resulting in poor performance when mapped onto the FPGAs DSP-slices. Thus, for higher performance, either a pipelined implementation in VHDL, or the use of vendor provided IP cores are the possible solutions. Xilinx provides a wide variety of IP-cores including multipliers and also have automatic pipeline generation when providing the multiplier operand with subsequent registers.

The same is also relevant for floating point operations since support for floating point operations in VHDL was not added until VHDL-2008 and many tools still have no support for the libraries. Due to the complexity of the floating point operations the package is not intended, nor suitable for synthesis, which leaves the options of designing floating point modules from scratch, or using IP cores as choices for implementation.

5.5 Xilinx CoreGEN IP Cores

Starting with VHDL-2008, floating point operations are included as IEEE standard functions allowing easy implementation and simulation of the operations. The drawback of the IEEE VHDL implementation of floating point operations is the fact that all operations are performed in one clock cycle which results in large hardware consumption as well as low clock frequencies.

The alternatives to the IEEE floating point library is to use either Xilinx Intellectual Property (IP) Core Generator or writing a floating point library from scratch that utilizes less hardware.

Xilinx provides the tool CoreGEN which has access to a large set of IP Cores, including floating point operations. The hardware can be instantiated with generic input widths and latency for the output. With higher latency the component uses less hardware between registers and is able to operate at higher clock frequency. The drawback of using CoreGEN is that the design becomes technology dependent. By using Xilinx CoreGEN, the design can only be implemented on a Xilinx FPGA. Also, the core generated for one hardware platform, for example Virtex-5 is not guaranteed to work for the next generation. This means that depending on which target hardware platform the design is implemented on, the generated cores differ from each other. The generated cores for the designs in this thesis are optimized for speed and have full usage of DSP slices.

5.6 Technology Independence of Arithmetic Operations

FPGA vendors usually complement their hardware with IP cores for various arithmetic operations utilizing the available DSP-slices, as well as through inference by using VHDL operators. Both Xilinx's XST and Altera's Quartus support automatic pipelining by adding registers prior to or after the VHDL math operator [14][17]. By reusing the implementations of the math components, the design can be made relatively technology independent. Only the base components need replacing which is made easy with IP core components and pipelining support in synthesis tools. Since the more complex math operations consist of the multiplication and accumulation those are the only two components necessary to replace when moving between platforms.

When designing components with high requirements on both performance and chip utilization it becomes harder to implement a solution that is truly technology independent. When implementing multipliers and accumulators the demand on high throughput makes pipe-lining a requirement which can be difficult to implement in an optimal way for technology independence.

6. Implementation of Generic Computation Units

One of the main directives of the hardware based algorithm is the scalability of the solution so that the implementation can be scaled to problems of different sizes as well as different hardware platforms. By designing the solution based on generics and with scalability as a directive the solutions can be designed efficiently. The final implementation consists of three different solutions with drawbacks and advantages of their own.

6.1 Fixed and Floating Point Implementation

One important parameter of the design is the choice of number representation. As mentioned in section 4.3, the floating point number representation provides higher dynamic range as well as precision compared to fixed point number representation. However, due to the complex structure of floating point number representation the calculations require substantially more logic and delays than fixed point. In this thesis work the designs for both 32 bits fixed point and single precision floating point are considered and compared.

6.1.1 Floating Point Implementation Designed for Specific Hardware

The base components were also designed for a floating point number representation of 24 bits in total. Out of those 24 bits, 17 are used for the mantissa and 7 for the exponent. This is of interest due to the reduced amount of hardware used. The DSP slice described in section 5.4 consists of one 25×18 bit signed multiplier so when using this number representation only one DSP slice is needed for the 17 bit mantissa multiplication instead of two when implementing single precision number representation. Depending on requirements for signal to noise ratio and the dynamic range of the system, a scaled down implementation can reduce the needed amount of hardware significantly.

6.2 Design Techniques

As mentioned in section 5.5, Xilinx provides designers with IP cores designed for specific purposes. Using Xilinx CoreGEN IP cores is effective and it does not take many man hours to implement a design using the cores. In this thesis work the floating point implementation uses IP cores from Xilinx CoreGEN, while the fixed point designs are implemented in two different ways. The first one uses IP cores in the design while the other one is designed to suit the hardware in a pipelined structure and written from scratch. The latter is further on called fixed point Non-Technology Dependent (NTD).

6.3 Complex Multiplier

Since the inputs to the algorithm consists of complex numbers the operations use more hardware, especially for multiplication. Addition and subtraction of complex numbers are only addition or subtraction of the real and imaginary part. Table 6.1 shows the total number of operations used for one complex multiplication and addition/subtraction.

Table 6.1: Computation blocks required for complex operations.

Operation	**# multipliers**	**# adders**	**total**
Multiply	4	2	6
Add/Subtract	0	2	2

The component implemented to solve a complex multiplication, thus, consists of 4 multipliers, one adder and one subtractor.

Figure 6.1 describes the final design in a simple block diagram. This single component performs the complex multiplication between the two inputs.

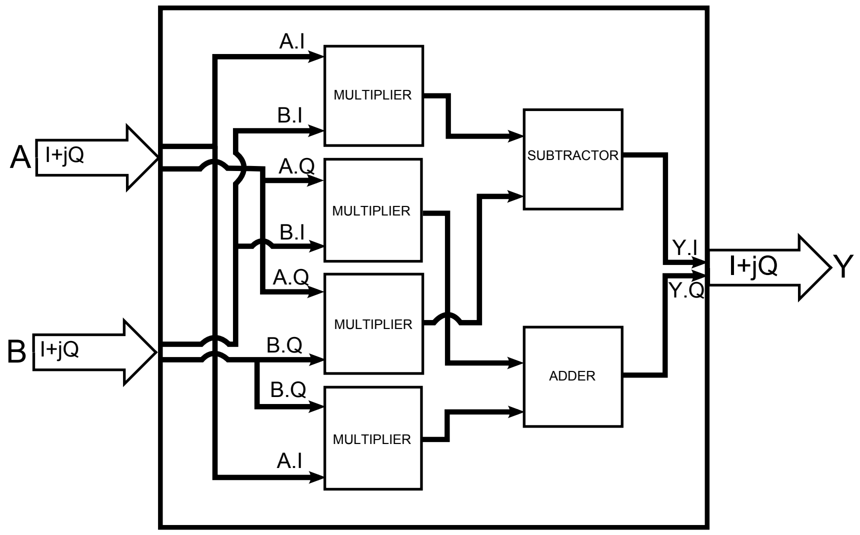


Figure 6.1: Block diagram of a complex multiplier.

6.4 Parallel Multiplier

The parallel multiplier is designed to solve the matrix multiplication between $\boldsymbol{s}^H$ and $\hat{\boldsymbol{R}}^{-1}$, although not the entire operation in parallel but column by column of $\hat{\boldsymbol{R}}^{-1}$. It utilizes the complex multiplier described in section 6.3 along with complex adders. N_{sp} multipliers are implemented in parallel and use an adder tree structure after the multipliers to calculate the sum of the multiplier products. This is illustrated in Figure 6.2. The resulting output is an element in the vector $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}$. Every iteration, k is increased until $k = N_{sp}$. The parameter k is controlling what column of $\hat{\boldsymbol{R}}_{sp}$ the parallel multiplier is processing. When the last column is done, it starts over with $k = 0$ and i is incremented to use the next part of the steering vector.

The number of adders used to sum the output from the multipliers adds up to $N_{sp} - 1$ and the latency through the adder tree, which results from registers between inputs and outputs of the adders, can be calculated by $log_2(N_{sp})$ multiplied with the latency of an

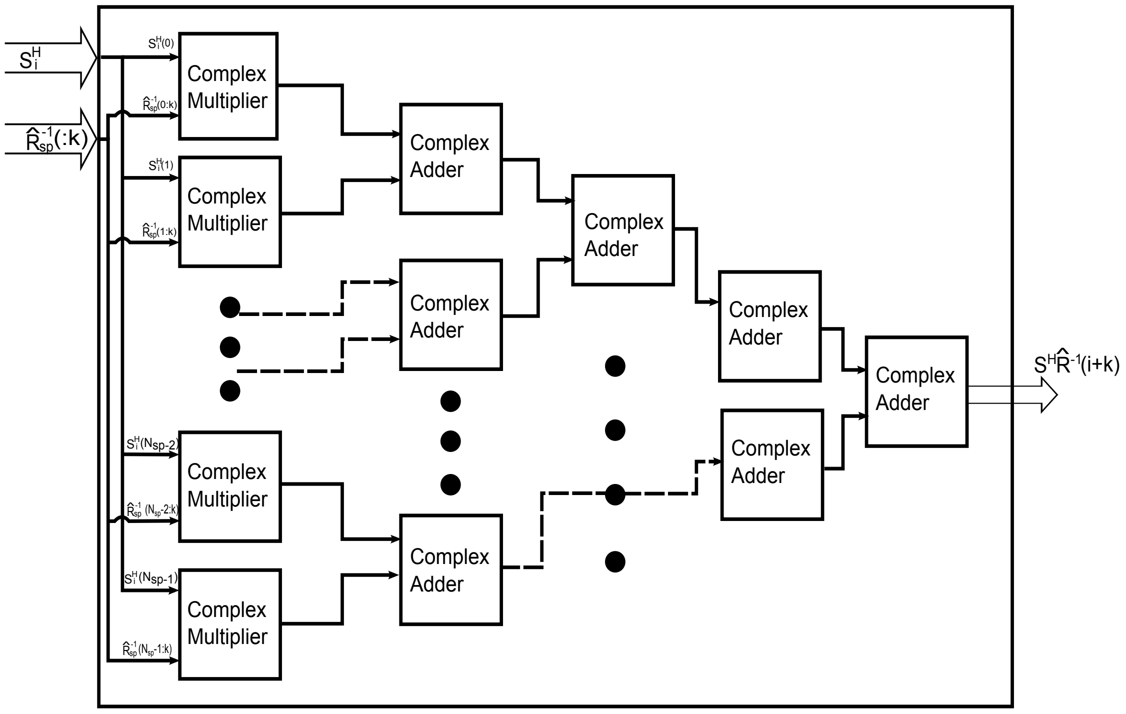


Figure 6.2: Block diagram of the parallel multiplier with adder tree.

adder. The parallel multiplication unit calculates the resulting vector $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}$ one element each clock cycle in the order from first to last. The output can then be pipelined to another complex multiplier that calculates the final scalar $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}\boldsymbol{s}$ or performing the multiplication of $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}\boldsymbol{x}$.

6.5 Calculations of $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}\boldsymbol{s}$ and $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}\boldsymbol{x}$

To perform the last matrix multiplications the resulting vector from the $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}$ multiplication is multiplied with the input vector $\boldsymbol{x}$ and the steering vector $\boldsymbol{s}$ in parallel. This results in two multiply and accumulate (MAC) functions as described in section 4.1.2. The component utilizes the parallel multiplier described in section 6.4, one MAC for $\boldsymbol{x}$, one MAC for $\boldsymbol{s}$, ROM storage for the steering vectors and some additional logic to create a pipeline. By introducing a delay between the parallel multiplier pipeline and the MAC units the same input can be used for the MAC unit as for the parallel multiplier. The MAC operations are performed for both $\boldsymbol{s}$ and $\boldsymbol{x}$.

Before calculations can be performed, the parallel input from the ROM storage needs to be filled, which is done by an initialization sequence that is triggered by a reset. After initialization the inputs are automatically updated for each new input of $\hat{\boldsymbol{R}}_{sp}^{-1}$ and $\boldsymbol{x}$.

6.5.1 Multiply and Accumulate for Floating Point

For fixed point a MAC function is quite simple to implement, but it is a challenge to implement it for floating point data. When using floating point number representation, the bits are divided into sign bit, exponent bits and mantissa. To be able to accumulate two floating point numbers, the mantissa with the lowest exponent has to be right shifted before the sum of the mantissas can be calculated. The alignment of mantissas needs pipelining with a few cycles delay to allow for high frequency operation.

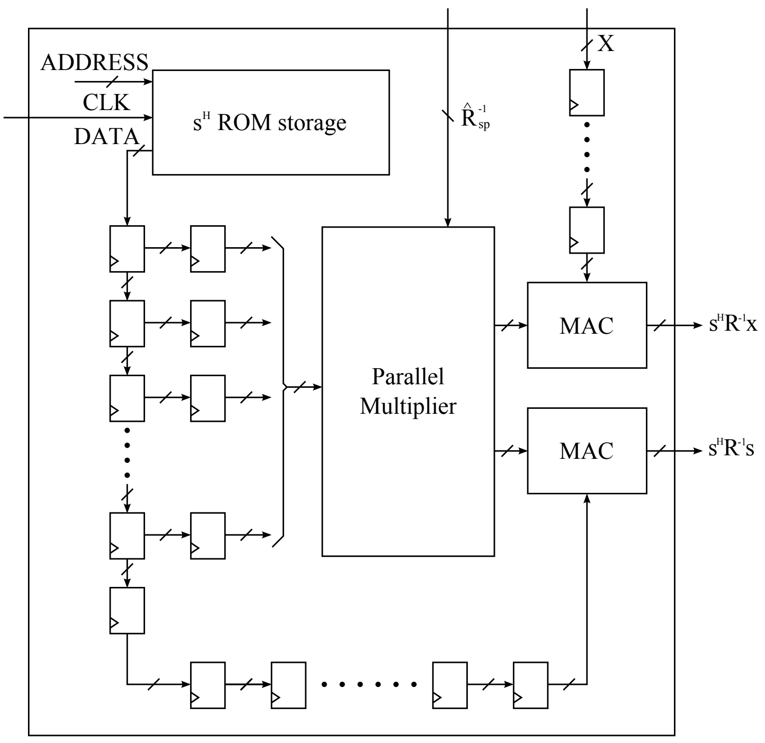


Figure 6.3: Parallel complex multiplier with Read-Only-Memory for storing the steering vectors.

The MAC component is connected to the output of the parallel multiplier which calculates a new value every clock cycle. This requires the performance of the MAC component to match the parallel multiplier. When adding a value to the accumulated sum every clock cycle, the previous sum need to be fed back in a single-cycle feedback loop. The latency needed for mantissa alignment results in difficulties to implement floating point MAC.

Possible methods to solve this problem include writing the input values for the accumulators to a buffer and reorder calculations to allow for the delay cycles in the additions. Restructuring of the algorithm to alter the data flow to allow for the delay cycles is also a possible solution.

For this thesis project the floating point MAC was implemented using floating point to fixed point conversion. The floating point data is converted to a fixed point value with a very high precision and connected to a fixed point accumulator of the same size allowing single-cycle feedback.

6.6 Remaining Parts of Kelly's GLRT

This design focuses on the computational heavy parts of Kelly's GLRT, namely the functions described previously in this chapter. The remaining parts of the algorithm consist of one addition, two multiplications and one division. They only need to be calculated when a new value is produced from $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}\boldsymbol{s}$ and $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}\boldsymbol{x}$. This means that they occur less frequently and thus, these calculations are not included in the design to be implemented in the FPGA. The result produced by the design needs processing outside the FPGA to perform the last calculations.

The complete design to be implemented consist of a wrapper mapping up a number of $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}\boldsymbol{s}$ and $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}\boldsymbol{x}$ blocks, as the one in Figure 6.3, in parallel. The number of blocks is set by a generic in the wrapper entity. The wrapper also contains RAM storage units for $\hat{\boldsymbol{R}}_{sp}^{-1}$ and $\boldsymbol{x}$, described further in section 6.7.

The design is generic and controlled from this wrapper. The ports and generics for the final component are found in Table 6.2 and 6.3. The signal type called complex is a record formed of two 32-bit data words of type std_logic_vector. The generics make it possible for the user to adjust the design according to case specific parameters or depending on the amount of hardware available.

Table 6.2: The ports for the entire design.

Name	**I/O**	**Type**	**Details**
clk	Input	std_logic	System clock
reset	Input	std_logic	Synchronous reset
Rsp_in	Input	complex	Complex input for $\hat{\boldsymbol{R}}_{sp}$
x_in	Input	complex	Complex input for $\boldsymbol{x}$
write_en_Rsp	Input	std_logic	High when a new Rsp value is written
write_en_x	Input	std_logic	High when a new $\boldsymbol{x}$ value is written
sHRs_s	Output	complex	Outputs the result for $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}\boldsymbol{s}$
sHRx_s	Output	complex	Outputs the result for $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}\boldsymbol{x}$
ready_x	Output	std_logic	Signal to indicate when the $\boldsymbol{x}$ buffer is ready to receive data
ready_rsp	Output	std_logic	Signal to indicate when the $\hat{\boldsymbol{R}}_{sp}^{-1}$ buffer is ready to receive data
finished	Output	std_logic	Finished flag that is set high when new, valid results are on the outputs

Table 6.3: The generics used to control the entire design.

Name	**Type**	**Details**
g_Nsp	Natural	Generic for the spatial channels
g_kronecker	Natural	Generic for the Kronecker multiplicand. The number of $\hat{\boldsymbol{R}}_{sp}^{-1}$ in the $\hat{\boldsymbol{R}}^{-1}$.
g_fifo_size	Natural	Generic to control the $\hat{\boldsymbol{R}}_{sp}$ buffer
g_data_width	Natutal	Generic that sets the number of bits for data
g_rows_per_unit	Natural	Generic for the number of rows from the steering vectors each instance shall use
g_parallel_rows	Natural	Generic that sets the number of instances mapped in parallel
g_type	String	Generic that sets the data_type and which design to use. Shall be "fixed_ntd", "float" or "fixed"

The final wrapper containing the designs is built up from several files and components written in VHDL. Figure 6.4 represents the top module which utilizes the components described in this chapter. As seen in the figure the design contains a Parallel-In, Serial-

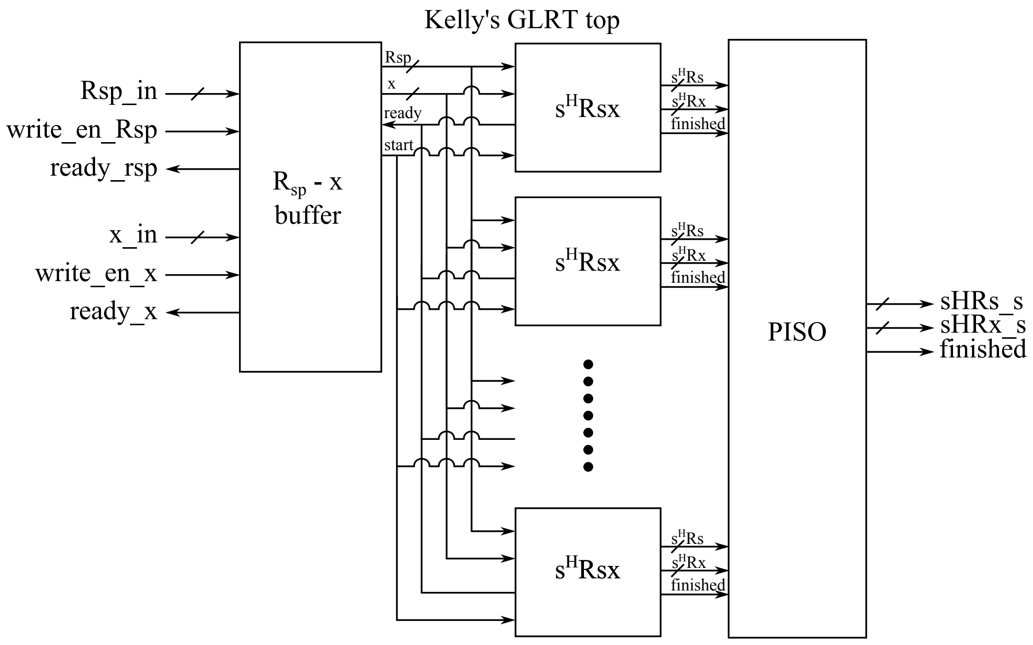


Figure 6.4: Block diagram of design top module, utilizing the sub-design units.

Out shift register to reduce the number of output pins on the device. The final design structure is described further in appendix A, where an illustration over the files used for the different designs is presented.

6.7 Memory Management

To provide the arithmetic units with data and ensure a high utilization, memory management is needed alongside with buffering and control logic. For the $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}\boldsymbol{s}$ and $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}\boldsymbol{x}$ implementation, there are three variables that need to be stored in memories, the static steering vectors, the dynamic covariance matrix and samples $\boldsymbol{x}$.

The most simple and implementation friendly solution is using block ram to implement the memory elements due to on-chip accessibility with no need for complex memory controllers for off chip memory.

Since the $\boldsymbol{s}^H$ vectors are static they can be implemented with single port block RAM with only read access and during synthesis the ROM memory elements can be preprogrammed with the memory contents so nothing more than reading access is needed for the implementation. Because each of the $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}\boldsymbol{s}$ and $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}\boldsymbol{x}$ units use independent vectors the ROM memory components are implemented alongside the parallel complex multipliers. This is also illustrated in Figure 6.3.

As for the $\boldsymbol{x}$ vector and $\hat{\boldsymbol{R}}_{sp}^{-1}$ matrix, a fifo buffer is needed so that computations can be performed while writing new input data to the buffer. Depending on the number of rows to be processed the same vector is required to be fetched from the buffer several times, as well as the covariance matrix that is reused several time for each row.

The covariance matrix buffer consists of N_{sp} parallel block ram components with individual read ports but share the same write port with individual write signals to allow

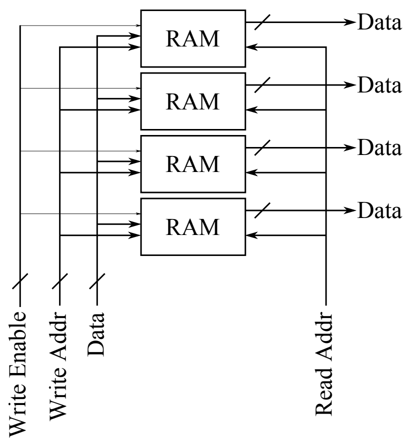


Figure 6.5: Parallel block rams with shared addresses, input data and parallel output.

writing to the individual blocks which is illustrated in Figure 6.5. With this approach the required input pins are significantly reduced and together with the re-utilization of data in the buffer the arithmetic pipeline can be run without any waiting states.

The buffer is implemented by reading and writing each row of the $\hat{\boldsymbol{R}}_{sp}^{-1}$ to an individual RAM memory, this way, one element from each row can be read simultaneously, and one column of $\hat{\boldsymbol{R}}_{sp}^{-1}$ can be clocked into the pipe each clock cycle. For the buffer part, the RAM cell is divided into banks, Figure 6.6, where each bank can fit one row of $\hat{\boldsymbol{R}}_{sp}^{-1}$, so that when the first set of banks have been written, the next set of rows can be written to the next bank while the arithmetic operations are still processing data from the previous bank.

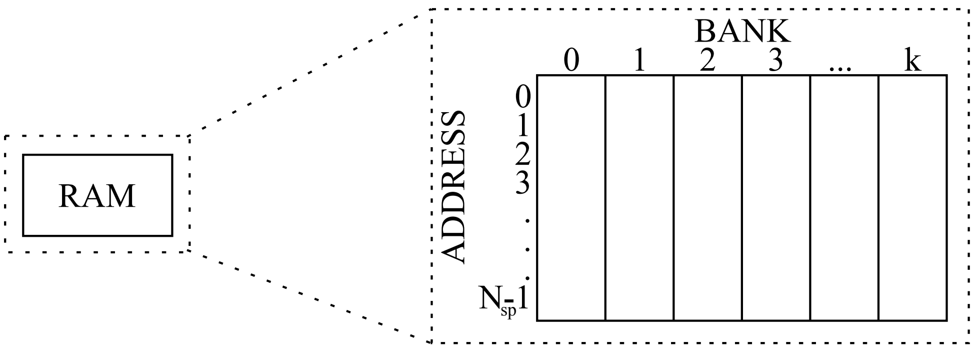


Figure 6.6: Illustrates the structure of the RAM, where different banks are used for storing rows of $\hat{\boldsymbol{R}}_{sp}^{-1}$.

The addressing of the RAM modules are done by concatenating the bank address with the relative address in each bank providing an absolute address in the block ram memory. The same principle applies to the $\boldsymbol{x}$-buffer, with the difference that there is no need to read more than one value each clock cycles so only one memory component is needed.

Since $\boldsymbol{x}$ and $\hat{\boldsymbol{R}}_{sp}^{-1}$ are the only input data, the control logic uses the buffers' status flags to control the arithmetic operations. As long as both buffers are not empty, data are read and fed to the pipe together with a start signal to trigger the next calculation.

6.8 Functional Verification using MATLAB and Testbench

All components are tested to verify their functional behaviour. For smaller components and functions, simulations in ModelSim are done. For the larger components, such as the parallel multiplier and the final wrapper, a generic testbench was written in VHDL. The stimuli data is generated by running a matlab script before running the testbench the first time. All stimuli data and results are saved in text files and read by the testbench. The testbench feeds the components with stimuli data and checks the result. For a more specific description of the testbench see appendix B.

7. Implementation Results

This chapter presents the results from the implementation described in chapter 6. The results are basen on Virtex-7 as target hardware using XST as synthesis tool. The complete results for Virtex-5 and Synplify Pro can be found in appendix C.1 and C.2.

The components without the memory management are not possible to place and route due to the large amount of I/O pins. Thus, only the result after synthesis is presented for the smaller components. For the final design the results from both synthesis and place and route are presented.

Example parameter sizes are used to achieve results that reflect reality. The parameter g_Nsp in this example is 16 and the Kronecker parameter is set to 30. The two parameters set the length of vectors $\boldsymbol{s}$ and $\boldsymbol{x}$ to 480 elements and $\hat{\boldsymbol{R}}^{-1}$ the size of 480×480. These parameters are used to obtain the results presented in this chapter.

Results for single precision floating point and fixed point of 32 bits are presented for all relevant components designed in this thesis followed by the result from a comparison between Virtex-5 and Virtex-7 as well as XST and Synplify Pro. Finally the result regarding the scalability of the design is presented.

7.1 Complex Multiplier

The results presented in Table 7.1 is from when synthesizeing the complex multiplier against the Xilinx Virtex-7 device XC7VX980T using XST.

Table 7.1: Resource utilization for complex multiplication after synthesis with ISE XST.

Design	**DSP-slices**	**LUTs**	**FFs**	**Maximum frequency**
Floating point	12	1123	1338	526.6 MHz
Fixed point Core	16	261	365	634.9 MHz
Fixed point NTD	16	165	323	623.3 MHz

7.2 Parallel Multiplier

The parallel multiplier is N_{sp} complex multipliers in parallel. Their resulting products is added up in a tree of adders as described in chapter 6.4. Using the example parameters the design required 16 complex multipliers to perform the calculations as intended. When synthesizeing the parallel complex multiplier against the Virtex-7 device XC7VX980T using XST the result presented in Table 7.2 was gained.

Table 7.2: Resource utilization for parallel complex multiplication after synthesis with XST.

Design	**DSP-slices**	**LUTs**	**FFs**	**Maximum frequency**
Floating point	252	26878	31458	526.6 MHz
Fixed point Core	256	6156	7344	634.9 MHz
Fixed point NTD	256	9649	6175	629.3 MHz

7.3 Calculations of $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}\boldsymbol{s}$ and $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}\boldsymbol{x}$

The component performing $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}\boldsymbol{s}$ and $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}\boldsymbol{x}$ uses the previously described components as well as a ROM for storing the steering vectors. The resulting number of operations in this component is 138. Table 7.3 shows the result from synthesis with XST against the Virtex-7 device XC7VX980T.

Table 7.3: Resource utilization for the $\boldsymbol{s}^H\hat{\boldsymbol{R}}_{sp}^{-1}\boldsymbol{s}$ and $\boldsymbol{s}^H\hat{\boldsymbol{R}}_{sp}^{-1}\boldsymbol{x}$ component after synthesis with XST.

Design	**DSP-slices**	**LUTs**	**FFs**	**Maximum frequency**
Floating point	276	44812	78558	212.8 MHz
Fixed point Core	288	8848	1332	629.3 MHz
Fixed point NTD	288	5349	14116	629.3 MHz

7.4 Final Design - Kelly's GLRT

The final design, containing RAM storages for $\boldsymbol{x}$ and $\hat{\boldsymbol{R}}_{sp}^{-1}$ was synthesised, placed and routed against the Virtex-7 device XC7VX980T for all three implementations. The timing constraints used was a period time on 4 ns. The results from the synthesis are presented in Table 7.4.

Table 7.4: Resource utilization for the final design after synthesis with XST.

Design	**# Units**	**DSP-slices**	**LUTs**	**FFs**	**Maximum frequency**
Floating point	3	828 (23%)	129036 (21%)	206331 (17%)	159.9 MHz
Fixed point Core	12	3456 (96%)	84051 (13%)	187261 (15%)	421.4 MHz
Fixed point NTD	12	3456 (96%)	42075 (6%)	195926 (16%)	421.4 MHz

After place and route of the design on the hardware device the results presented in Table 7.5 was obtained. The table shows resource utilization as well as maximum frequency and the number of units that are mapped in parallel.

Table 7.5: Resource utilization for the final design after place and route.

Design	# Units	DSP-slices	LUTs	FFs	Maximum frequency
Floating point	3	828 (23%)	117011 (19%)	167219 (13%)	140.1 MHz
Fixed point Core	12	3456 (96%)	79220 (12%)	162416 (13%)	168.9 MHz
Fixed point NTD	12	3456 (96%)	60777 (9%)	195798 (15%)	189.3 MHz

7.5 System Performance

The following section present the result for the entire design according to performance measured in the amount of operations performed by one device with the specific design implemented. Table 7.6 displays the results consisting on the system frequency, number of $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}\boldsymbol{s}$ and $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}\boldsymbol{x}$ units mapped in parallel and the total amount of operations for the implementation. As mentioned in section 7.3, the number of operations per unit is 138.

Table 7.7 presents the performance in a more general way. The power consumption for the design is presented and the performance is measured in operations per second (OPS) for either floating point (FL) or fixed point (FI). The OPS gained for the design is then divided with the power consumed by the device to give a general result that can easily be compared to other implementations.

Table 7.6: Performance of final system implemented on Virtex-7.

Design	Maximum frequency	# Units	Tot. Operations
Floating point	140.1 MHz	3	58 GFLOPS
Fixed point Core	168.9 MHz	12	280 GFIOPS
Fixed point NTD	189.3 MHz	12	313 GFIOPS

Table 7.7: Power efficiency of final system implemented on Virtex-7.

Design	Power Consumption	Performance	Ops/W
Floating point	2.500 W	58 GFLOPS	23.2 GFLOPS/W
Fixed point Core	8.171 W	280 GFIOPS	34.3 GFIOPS/W
Fixed point NTD	7.981 W	313 GFIOPS	39.3 GFIOPS/W

Figure 7.1 shows the performance in OPS and OPS/W for the different designs.

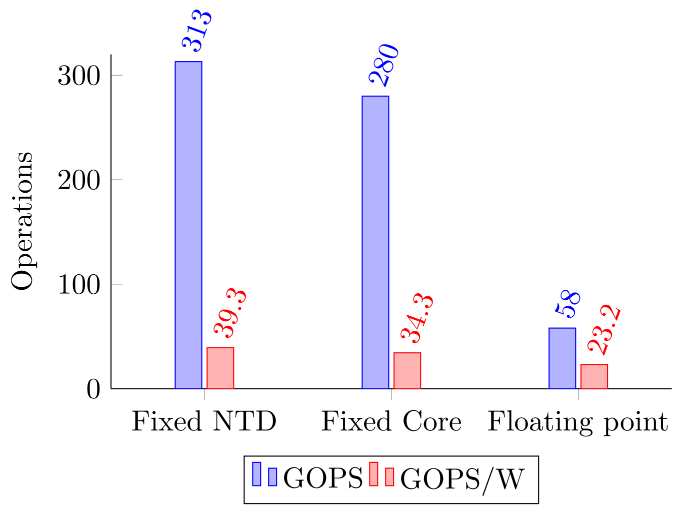


Figure 7.1: System perfomance for the three designs presented in GOPS and GOPS/W.

7.6 Synthesis tools

The final design is built from the components described in sections 7.1, 7.2 and 7.3. When using Synplify Pro as the tool for synthesis for those components and comparing the result to the one after synthesis with XST indicates an overall higher performance using XST. The resource utilization did not differ too much, but the estimated maximum frequency differed according to Table 7.8.

Table 7.8: Comparison of XST and Synplify Pro for the base components of the design.

	Frequency XST	**Frequency Synplify Pro**
Complex Multiplier Floating Point	526.6 MHz	507.5 MHz
Complex Multiplier Fixed Point Core	634.9 MHz	529.4 MHz
Complex Multiplier Fixed Point NTD	623.3 MHz	250.3 MHz
Parallel Multiplier Floating Point	526.6 MHz	507.5 MHz
Parallel Multiplier Fixed Point Core	634.9 MHz	381.1 MHz
Parallel Multiplier Fixed Point NTD	629.3 MHz	250.3 MHz

The result presented in the Table is from when synthesizing against the Virtex-7 device XC7VX980T. As seen in the Table, the biggest difference is when using the NTD design for fixed point implementation.

7.7 Scalability

The design is intended to be scalable when considering migrating from one technology generation to the next. The design is generic and scalable so that the same design is functional for different sizes of the data parameters specific for the application. To change the data parameters, the generics of the design are used.

When migrating from one technology to the next it is possible to map more parallel $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}\boldsymbol{s}$ and $\boldsymbol{s}^H\hat{\boldsymbol{R}}^{-1}\boldsymbol{x}$ modules according to the available resources. The NTD design is completely scalable between different devices. However, the design using IP cores is not scalable between different generations. In this thesis the designs was implemented on the Virtex-7 device, but an implementation on the Virtex-5 device also was done. Due to the smaller amount of hardware available in the Virtex-5 device, the number of units mapped in parallel was decreased. Table 7.9 shows the maximum frequency for the different designs when utilizing the available chip resources as much as possible on the two devices. Table 7.10 present the performance for the different designs implemented on the Virtex-5 device and Figure 7.2 displays the OPS and OPS/W for the fixed point NTD for the implementation on Virtex-5 compared to the Virtex-7 device.

Table 7.9: Maximum frequency for the different designs on Virtex-5 and Virtex-7.

Design	Virtex-5	Virtex-7
Floating point	66.3 MHz	140.1 MHz
Fixed point Core	102.6 MHz	168.9 MHz
Fixed point NTD	167.5 MHz	189.3 MHz

Table 7.10: Performance of final system implemented on Virtex-5.

Design	# Units	Power Consumption	Tot. Operations	OPS/W
Floating point	3	4.5 W	27.4 GOPS	6.0 GOPS/W
Fixed point Core	3	4.6 W	42.5 GOPS	9.2 GOPS/W
Fixed point NTD	3	5.5 W	69.3 GOPS	12.7 GOPS/W

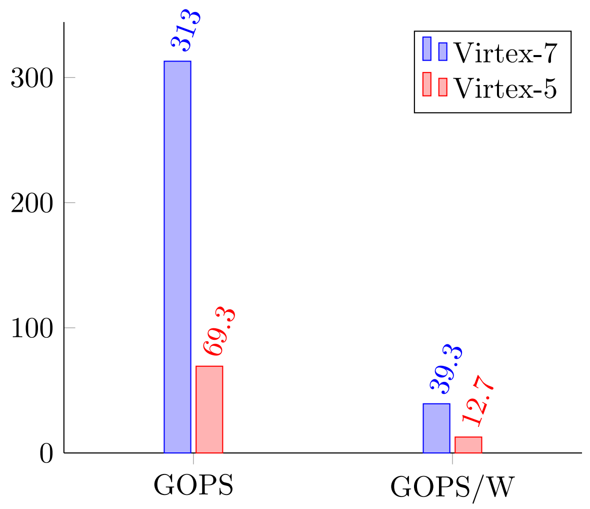


Figure 7.2: GFIOPS and GFIOPS/W for entire fixed point NTD design for Virtex-7 and Virtex-5.

7.8 Floating Point Implementation Designed for Specific Hardware

For the base components, a solution using number representation of 24 bits floating point was designed. Out of those 24 bits 17 was used for the mantissa, as described in section 6.1.1. For the complex multiplier and the parallel complex multiplier the results

Table 7.11: Synthesis results for complex multiplier and parallel multiplier using 24 bits floating point number representation.

Design	**DSP-slices**	**LUTs**	**FFs**	**Maximum frequency**
Complex Multiplier	4	1127	1240	561.0 MHz
Parallel Complex Multiplier	64	28442	31840	561.0 MHz

presented in Table 7.11 was obtained. The same example parameters were used and the Virtex-7 device was the target hardware platform. XST was used for synthesis.

8. Results from other Parallel Solutions

The theory behind STAP has been discussed frequently over the past, but the computational demand requires high performance processors or hardware to create an implementation that operates in real-time. This chapter present some existing result from solutions in Graphic Processor Units (GPU) and multi-core processors to put the result presented in chapter 7 in perspective. All of the results presented in this chapter are gathered from articles and reports on the subject. For more information about the implementations described see the references for this chapter.

8.1 Graphic Processor Units (GPU)

The parallel structure of GPUs make them suitable for implementing scientific computing and other parallel computations with high performance requirements, this is referred to as General Purpose Computation on Graphics Hardware. In 2010 a performance evaluation was performed as a master thesis evaluating the suitability for using GPUs for Radar Signal Processing. By utilizing CUDA[1] and OpenCL programming framework STAP among other algorithms were implemented on a NVIDIA GTX 260 GPU with the specifications in Table 8.1.

Table 8.1: Specifics for NVIDIA GeForce GTX 260.

Performance (FLOPS)	875 GFLOPS
Power (Watt)	171W
FLOPS/W	5.1 GFLOPS/W

In Table 8.2 the results from the implementation can be observed. Only the nominal power was studied in the thesis report so no actual power consumption of the GPU during the execution of the algorithm was mentioned.

Table 8.2: Implementation results of STAP on NVIDIA GPU.

	Maximum performance	Actual performance	Ratio
Without memory transfers	875 GFLOPS	240 GFLOPS	28%
With memory transfers	875 GFLOPS	130 GFLOPS	15%

As a comparison to newer GPU's, NVIDIA's Tesla GPU which implements their new Kepler GPU core has a peak performance of 4.58 TFLOPS single precision at 225W according to NVIDIA's specifications [18][19].

[1]CUDA$^{\text{TM}}$is a parallel platform and programming model/framework developed by NVIDIA.

8.2 Multi-core Processors

The company BittWare has a multi-core co-processor for FPGAs in their product range that in their case work together with an Altera Stratix FPGA. The co-processor is named Anemone and creates a hybrid solution for floating point operations. Anemone is a low power multi-core processor consisting of 16 floating point cores and is scalable for up to a maximum of 4096 cores providing a peak of 4.9 TFLOPS [20]. According to BittWare the Anemone is able to provide 19.2 GFLOPS/W in computational performance.

The multi-core processor has been used for implementation of a matrix multiplication algorithm called xGEMM, which suggests that the processor has been capable of up to 90% of the theoretical performance peak for that specific algorithm using 16 cores [21].

9. Discussion

The intention of this chapter is to discuss the results and the advantages and disadvantages of the different designs compared to one another. The focus is on what have impact on the results, why and how the bottlenecks might be solved. The hardware, tools and performance are discussed and analysed. In addition to discussing the design this chapter also intends to analyse and reason about the suitability for implementing STAP on an FPGA in the near future.

9.1 Hardware platform

The target platform in this thesis work is a Xilinx Virtex-7 device and it is of importance that this is not the only suitable platform on the market. There are a number of vendors on the market and the designs of the different FPGAs are not identical. When the device Stratix V from Altera was studied it was found that the DSP blocks used in this device are slightly different. The datasheet for the device says that the device supports up to 3,926 18×18 or 1,963 27×27 multipliers [22]. Even though this thesis work focused on hardware platforms from Xilinx, it might be interesting to investigate other vendors further to reach the best result.

Focusing on the Xilinx Virtex-7 and Virtex-5 devices, one aspect is that the slices change over time. Just by studying the two devices in this thesis work some differences are found. It is therefore likely that the development of future generations include more DSP slices and might contain even larger, more advanced DSP slices, with a possibility of improving the design even further.

9.2 Tools

During this thesis work the impact of the tools have been obvious, especially for the pipelined, non-technology dependent solution. In Table 7.8 one could clearly see results that differed depending on whether XST or Synplify Pro was used as synthesis tool. The designer needs knowledge about the tool to understand the result and to be able to optimize the design. The pipelined NTD design presented in this thesis is, as mentioned, non-technology dependent but still designed to be implemented as a tailored solution for this kind of DSP slice. The problem when using Synplify Pro for synthesis is that the tool is too general and can not handle the internal register in the DSP slices in the most efficient way. For this design solution to work and be fast enough, the internal registers in the DSP slices should be used for creating parts of the pipeline. Synplify Pro does not use those registers as intended, resulting in a very slow design. XST is a Xilinx tool for synthesis, and more aware of the DSP slice registers and how to pipeline those for a faster design.

Recently Xilinx provided the designers with a design suite called Vivado [23]. Vivado is a new design environment provided to accelerate designs on the Xilinx 7 series. If this design suite was used instead of ISE it could, according to the Vivado overview, improve frequency, power consumption, build-time and other parameters even further. Vivado was not in focus in this thesis work, but some minor tests were performed for the final design. Using Vivado, it was possible to implement the floating point design while utilizing the resources in the FPGA to a higher extent compared to when using ISE. However, the result from the tests with Vivado are not presented in this thesis due to the fact that the results were not verified. This had to do with lack of time and experience regarding the tool. But the tests did imply that Vivado might be of interest for implementing the floating point design and could possibly achieve better results for the design, both regarding speed and number of units possible to implement.

9.3 Number Representation

During the first phases of the design, results showed that the components for floating point could compete with the ones for fixed point. The estimated maximum frequency was slightly lower for the floating point design. However, the floating point design utilizes significantly more logic. During the development of the design it was not clear if it was the logic or the DSP slices that would be the limiting factor. In the final synthesis result it was found that for the floating point implementation it was in fact also the DSP slices that was the limiting factor. The intention was to use as much of the FPGA as possible and for the fixed point designs, with less logic, that logic is not used for anything else. It might be possible to implement other parts of the algorithm or the signal processing chain by using this logic, giving these solutions an even greater advantage compared to the floating point implementation.

One problem with the floating point design occurred when including the MAC functions. As described in section 6.5.1, the latency of the accumulation results in a rather slow solution. To solve this, the data is transformed to fixed point before performing the accumulation. The transformation is not optimal and might lead to loss of precision. It is possible to minimize the loss of precision, but it would most definitely require more resources. Also, this implementation requires additional hardware that could be used for other parts of the algorithm.

Finally, when trying to utilize the Virtex-7 resources as much as possible, ISE could not implement the floating point design. Hence, we only got synthesis results for that implementation. However, the design was possible to implement on the Virtex-5 device with high utilization, but the smaller FPGA resulted in fewer parallel modules. Using the same, lower number of modules in the Virtex-7 device implementation succeeded, utilizing the device to a smaller extent than intended. The reason for the problems with the Virtex-7 device were not solved, but it probably has to do with different optimization algorithms used depending on the device. When using the new tool Vivado, the synhtesized netlist for the high utilization could be implemented on the Virtex-7 device.

The use of floating point number representation is quite new for the vendors and it is not unlikely to believe that support will increase and as an example a better solution for floating point MAC will exist in a near future. It might also be possible to design the solution in a different way so that a single-cycle feedback loop no longer is needed.

When using only 24 bits for floating point number representation the solution only need one DSP slice to perform the multiplication. Compared to the single precision number representation, this resulted in 4 DSP slices instead of 12 for a complex multiplier. A design using fewer bits for number representation should also result in fewer wires to route. However, this solution utilizes almost the same amount of logic so it would require more optimization before the benefits of the reduced number of DSP slices can be used. If the system can tolerate a small loss of precision and dynamic range, it would be interesting to optimize this design and obtain a final result in terms of power consumption and performance.

9.4 Performance

Evaluating the design results from this thesis with the other parallel solutions and comparing both peak performance and power efficiency there is no obvious winner in the competition. The FPGA approach is able to utilize logic efficiently and also provides a dense operations/power ratio, this is also true for the GPU. However, when only utilizing the approximately 18-25% of the peak performance it is not known what the power consumption at that point is, hence, the actual operations/power ratio is not known. A limiting factor for the FPGA approach considering performance is the slow clock frequency, this is however due to a couple of bottlenecks in the design which should be possible to improve for increased performance.

In comparison to the multi-core Anemone processor, it seems that a multi-core processor approach could be beneficial for floating point operations. The performance of up to 90% of the theoretical maximum for matrix and vector operations indicates that the multi-core processor architecture is suitable for matrix type computation. The Anemone is a co-processor for FPGAs, a hybrid solution that is designed to use the FPGA for some parts of the signal processing but perform calculations that benefit more when run on the co-processor on the Anemone. This implies that if more work in these two areas are made, both the multi-core solution as well as the FPGA approach might have the capacity of being more energy efficient and have an overall higher performance.

The usual way of limiting power consumption in CPU and GPU architectures is done by switching off unused sections of the chips while computational load is low, but this can not be performed perfectly, and because of this, the operations/power ratio is probably significantly lower than the ratio of peak performance over peak power since the utilization of the STAP algorithm on CUDA cores reached 25% at most.

To be able to determine which solution would be most energy efficient, an analysis is needed to determine power consumption and not only performance evaluation for implementation of the STAP algorithm. More work in both the FPGA design as well as for the GPU and multi-core processor solutions is required to be able to make more relevant conclusions.

9.5 Implementation of STAP on an FPGA

Results from the fixed point implementation in this thesis implies that FPGA's are suitable for this type of complex multiplications and parallelism. The limiting factor is not the operations to be performed, but more the routing and transfer of data to and from the memory blocks. It is possible that a design with better routing would lead to a more energy efficient design. The design in this thesis work would require a huge amount of FPGAs to implement a complete STAP system of this magnitude and the intention is to use this in a airborne radar with a limited space for the hardware.

Computational demands of approximately 100 TOPS would result in over 300 FPGA chips. It would be possible to divide the algorithm and spread it out on a number of

FPGAs but a number of that size is not appropriate. The power consumption of this system is slightly above 2400W which exceeds the theoretical maximum of 400W by far.

9.5.1 Bottlenecks of the Design

One of the major bottlenecks in the design is the memory management and buffering of data. Due to large fanout and long paths the path delay limits the switching frequency of the design while all components are fast enough for significantly higher clock frequencies.

For the floating point implementation the MAC unit is also a major bottleneck, since the multiply and accumulate operation with the accumulated sum in a single-cycle feedback loop is difficult to implement. This is described previously in section 6.5.1 and section 9.3.

The bottlenecks in the design should be possible to solve by inserting buffer registers to limit fanout in conjunction with a MAC solution that is pipelined even further. Another possible solution would be to re-design the matrix calculations so that MAC operations with feedback are avoided and can be implemented using multipliers and adders without the feedback.

10. Conclusion

This chapter summarizes the conclusions made in this thesis work. The work is divided in the development of the design, performance and scaling. For specific results from the designs, see chapter 7.

10.1 Design and Number Representation

The final design was implemented on a Xilinx Virtex-7 device using ISE. Both the fixed point NTD solution as well as the fixed point solution using IP cores was implemented with as high utilization of the chip as possible. The final floating point design was not possible to place and route on the Virtex-7 with resource utilization higher than 25%. However, the floating point design was successfully implemented on the Virtex-5 device with as high utilization as possible but could not compare to the results for the fixed point designs.

It was found that the support for floating point arithmetics in the tools need further development to be able to provide high performance operations which is required for implementation of STAP and Kelly's GLRT.

10.2 Performance and Power Efficiency

The designs for fixed point had a performance in the interval 35-40 GOPS/W while the floating point implementation resulted in a slightly lower performance. The results are quite positive, but considering the computational demands and limited space for the radar system in an airborne platform they are not good enough. To achieve performance of 100 TOPS, this design would require over 300 Virtex-7 devices and result in a power consumption of over 2400 W. A system like that is not a realistic solution, why both performance and power efficiency needs improvement.

Compared to the GPU approach there is no clear benefit of using FPGAs in favor of GPUs when considering floating point implementation. However, considering the power efficiency the FPGAs are superior. The Anemone hybrid solution could compete with the FPGA approach but also requires an FPGA. Due to the lack of available results from actual STAP implementations, both considering power efficiency and performance no concrete conclusion can be drawn.

10.3 Scaling

The final design uses generics to control sizes of memory blocks, input vectors and number of multipliers in parallel. This makes it possible to adjust the design for a specific case and making it suitable for different sizes and parameters.

Due to the generic design, it is possible to change a few generics when a new, larger FPGA is on the market. Instead of a large number of FPGAs working together, the calculations can be divided on fewer devices of the new FPGA generation where one device handles more units in parallel and reducing the total number of FPGAs.

11. Future Work

All base components were designed to have a maximum frequency that are as high as possible. When inserting the RAMs for storing $\hat{\boldsymbol{R}}_{sp}^{-1}$ and $\boldsymbol{x}$, the estimated maximum frequency decreased significantly. Therefore, one continuation of this work could be to increase performance by inserting buffer registers to decrease the fan out from the RAMs. This would create a pipelined structure between the RAMs and the components performing the calculations and probably remove the largest bottleneck of this design.

A recommendation after this thesis work is to implement a floating point library instead of using the generated IP cores. This would remove some of the technology dependence. Also, it would give the designer the possibility to remove parts that are included in the cores, like different rounding choices not considered in this design. It should not be that hard to implement this library when having some knowledge of the floating point structure. The floating point library would also simplify the design into an even more generic design, where a generic easier could set the data type and not requiring different designs depending on the number representation used. With this thesis in mind it is also likely that the designer will gain more control over the synthesis and tools performing the synthesis when avoiding the core components, especially for the floating point design. When running the synthesis and place and route, the designs using core components took significantly longer time compared to the pipelined design. The problem with not being able to implement a similar design for floating point in an efficient way could maybe be solved by this floating point library. Also, to increase the possibilities of a successful floating point implementation even more, a suggestion is that the designer becomes more familiar with the new tool Vivado. This is simply because the minor tests performed during this thesis work indicated that Vivado could result in a better floating point implementation.

It could also be of interest to investigate other vendors further. It is a possibility that the DSP slices used in other vendors devices might suit this design so that more modules in parallel can be mapped in a single FPGA. Other vendors also claim their support for floating point implementations is very good compared to their competitors, and this could boost the floating point design discussed in this thesis work.

Finally, other parallel solutions should be investigated further. Information and result for this kind of implementation is hard to find and to make a fair comparison, more knowledge is needed. One suggestion for a future thesis work could focus on implementing the same algorithm on a GPU or multi-core processor and compare the results to the ones achieved in this thesis work for a more genuine evaluation.

Bibliography

[1] M. Richards, *Fundamentals of Radar Signal Processing.* McGraw-Hill Electronic Engineering Series, McGraw-Hill, 2005.

[2] M. Skolnik, *Introduction to radar systems.* McGraw-Hill, 1962.

[3] A. Åhlander, H. Forsberg, M. Jonsson, and B. Svensson, "Radar signal processing using pipelined optical hypercube interconnects," tech. rep., Department of Computer Engineering, Chalmers University of Technology, Sweden and Computer and Electrical Engineering, Halmstad University, Sweden and Radar Division, Ericsson Microwave Systems AB, Mölndal, Sweden, 2001.

[4] A. Choudhary, W.-K. Liao, D. Weiner, and P. Varshney, "Design, implementation and evaluation of parallel pipelined stap on parallel computers," in *IEEE IPPS/SPDP*, vol. 36, pp. 528–548, April 2000.

[5] J. Toomay and P. Hannen, *Radar principles for the non-specialist.* SciTech radar and defense series, SciTech Pub., 2004.

[6] S. Montebugnoli, G. Bianchi, A. Cattani, F. Ghelfi, A. Maccaferri, and F. Perini, "The medicina ira-ska engineering group, some notes on beamforming," 2004. http://www.ira.inaf.it/Library/rapp-int-2004/353-04.pdf.

[7] K. Ounissi, "Advanced signal processing techniques for pulse-doppler radar," Master's thesis, Chalmers University of Technology, November 2006.

[8] R. Klemm and I. of Electrical Engineers, *Principles of Space-Time Adaptive Processing.* Iee Conference Publication, Institution of Electrical Engineers, 2002.

[9] C. D. Richmond, "The theoretical performance of a class of spacetime adaptive detection and training strategies for airborne radar," in *Conference Record of the Thirty-Second Asilomar Conference on Signals, Systems and Computers*, vol. 2, pp. 1327–1331, November 1998.

[10] A. Åhlander and M. Taveniku, "Instruction statistics in array signal processing," tech. rep., Department of Computer Engineering, Chalmers University of Technology, Sweden and Centre for Computer Systems Architecture, Halmstad University, Sweden and Ericsson Microwave Systems AB, Mölndal, Sweden.

[11] A. Devices, "Fixed-point vs floating point digital signal processing." http://www.analog.com/en/content/fixed-point_vs_floating-point_dsp/fca.html.

[12] Xilinx, *XST Synthesis Overview*, 2008. http://www.xilinx.com/itp/xilinx10/isehelp/ise_c_using_xst_for_synthesis.htm.

[13] Xilinx, *Synthesis and Simulation Design Guide*, 13.1 ed., March 2011. http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/sim.pdf.

[14] Xilinx, *XST User Guide*, 10.1 ed., 2008. http://www.xilinx.com/itp/xilinx10/books/docs/xst/xst.pdf.

[15] Synopsys, *Synopsys FPGA Synthesis Reference Manual*, September 2011.

[16] Xilinx, *Virtex-5 FPGA XtremeDSP Design Considerations User Guide*, 3.5 ed., January 2012. http://www.xilinx.com/itp/xilinx10/books/docs/xst/xst.pdf.

[17] Altera, *Quartus II Handbook Version 12.0, Volume 1: Design and Synthesis*, 12.0 ed., June 2012. http://www.altera.com/literature/hb/qts/qts_qii5v1.pdf.

[18] NVIDIA, *TESLA K10 GPU Accelerator*, 5.0 ed., June 2012. http://www.nvidia.com/content/PDF/kepler/Tesla_K10_BD-06280-001_v05.pdf.

[19] NVIDIA, *NVIDIA® TESLA® KEPLER GPU Computing Accelerators*, 2012. http://www.nvidia.com/content/tesla/pdf/NV_DS_TeslaK_Family_May_2012_LR.pdf.

[20] BittWare, *Anemone104^{TM}Co-Processor for FPGAs*, 3.0 ed., October 2012. ftp://ftp.bittware.com/documents/data_sheets/anemone_ovrvw.pdf.

[21] Parlant Ltd, *Matrix multiplication of the multi-core Anemone processor*, April 2011. http://www.paralant.com/docs/Insight-whitepaper-sgemm.pdf.

[22] Altera, *Stratix V Device Overview*, 3.0 ed., 2012. http://www.altera.com/literature/hb/stratix-v/stx5_51001.pdf.

[23] Xilinx, *Vivado Design Suite*, 1.1 ed., 2012. http://www.xilinx.com/support/documentation/white_papers/wp416-Vivado-Design-Suite.pdf.

A. Final Design Structure - Kelly's GLRT

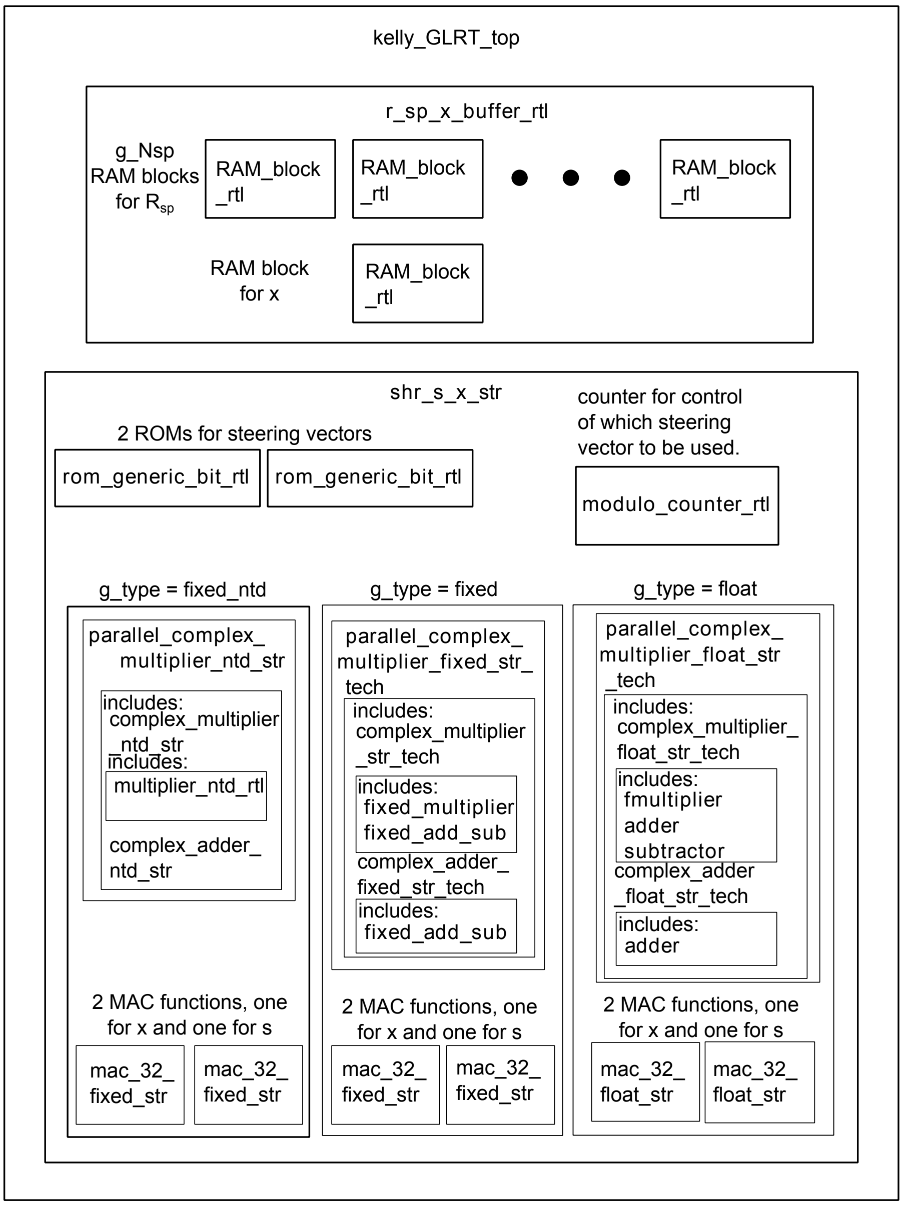

kelly_GLRT_top
r_sp_x_buffer_rtl
g_Nsp RAM blocks for R_{sp}
RAM_block _rtl
RAM_block _rtl
RAM_block _rtl
RAM block for x
RAM_block _rtl
shr_s_x_str
2 ROMs for steering vectors
rom_generic_bit_rtl
rom_generic_bit_rtl
counter for control of which steering vector to be used.
modulo_counter_rtl
g_type = fixed_ntd
parallel_complex_ multiplier_ntd_str
includes: complex_multiplier _ntd_str
includes: multiplier_ntd_rtl
complex_adder_ ntd_str
2 MAC functions, one for x and one for s
mac_32_ fixed_str
mac_32_ fixed_str
g_type = fixed
parallel_complex_ multiplier_fixed_str_ tech
includes: complex_multiplier _str_tech
includes: fixed_multiplier fixed_add_sub
complex_adder_ fixed_str_tech
includes: fixed_add_sub
2 MAC functions, one for x and one for s
mac_32_ fixed_str
mac_32_ fixed_str
g_type = float
parallel_complex_ multiplier_float_str _tech
includes: complex_multiplier_ float_str_tech
includes: fmultiplier adder subtractor
complex_adder _float_str_tech
includes: adder
2 MAC functions, one for x and one for s
mac_32_ float_str
mac_32_ float_str

Figure A.1: Final design structure

Figure A.1 illustrates the structure of the final design. Internal signals, ports and nets are not shown. The figure aims to illustrate the structure of source files and components written in VHDL or generated by ISE CoreGEN. The generic g_type controls which of the three instances that are generated. The MAC units consist of smaller components. The structure of the two different MAC units are described in Figure A.2 and A.3.

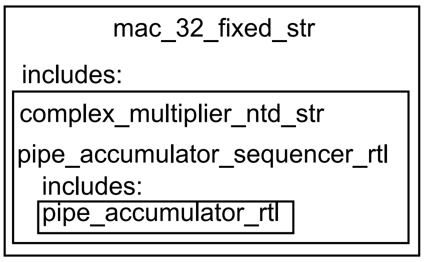


Figure A.2: Final design structure

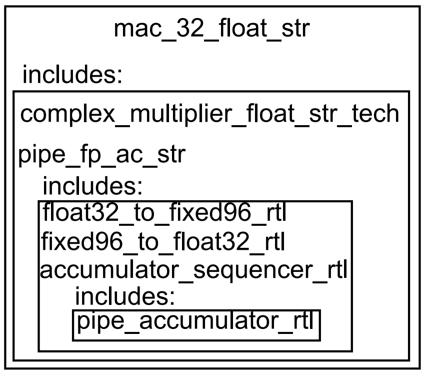


Figure A.3: Final design structure

B. Testbench

The matlab script generating stimuli and reference data is written to calculate the results from the different parts of the algorithm, starting with a single complex multiplication to continue with the parallel complex multiplication with an adder tree and then adding the final steps of the algorithm. This make it possible to test every part of the algorithm by using the corresponding result from the matlab script as a reference for the components result.

For the fixed point calculations the fixed point toolbox that is included in matlab is used. This gives the designer the possibility to set where the fixed point is placed, how many bits used to represent the number as well as additional settings like rounding mode. For the floting point calculations, the single precision format is used.

Due to rounding settings, the result from the testbench and the value in the text file might differ slightly. This is considered in the testbench and a small rounding error is allowed. The rounding error allowed has to be small so that the actual data used in the real system is accurate enough. The testbench is designed to be able to verify all different instances of a design at the same time, that means all fixed point components at the same time as the floating point implementations. Figure B.1 roughly describes the testbench, where the different designs under test (DUT) are the different component designs.

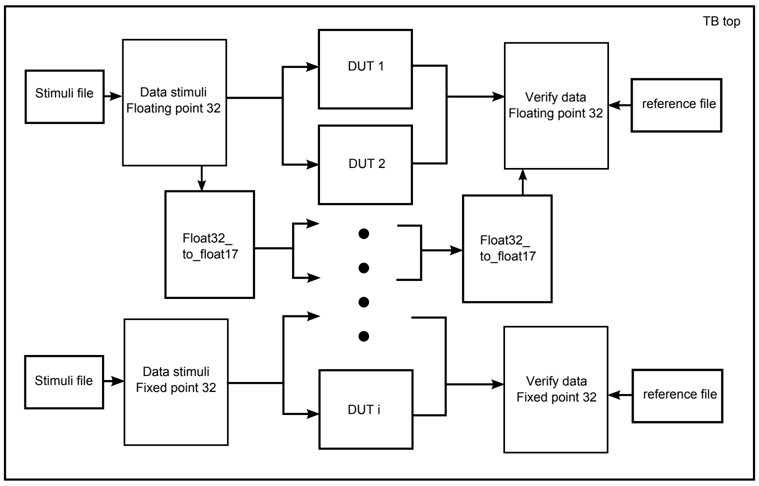

TB top
Stimuli file
Data stimuli Floating point 32
DUT 1
DUT 2
Verify data Floating point 32
reference file
Float32_ to_float17
Float32_ to_float17
Stimuli file
Data stimuli Fixed point 32
DUT i
Verify data Fixed point 32
reference file

Figure B.1: Block schedule of the testbench structure.

C. Additional Results

C.1 Synplify Pro against Virtex-7

When using Synplify Pro for synthesis, the result did not get as good as when XST was used instead. However, the synthesis was performed with Synplify Pro for almost all components of the design and the result are presented in this section.

In table C.1, the result from synthesis with Synplify Pro against the Virtex-7 device is presented.

Table C.1: Resource utilization for complex multiplication after synthesis with Synplify Pro against Virtex-7.

Design	**DSP-slices**	**LUTs**	**FFs**	**max freq. (MHz)**
Floating point	12	1078	1296	507.5
Fixed point Core	16	261	364	529.4
Fixed point NTD	16	194	388	250.3

Table C.2 presents the corresponding results for the parallel multiplier.

Table C.2: Resource utilization for parallel complex multiplication after synthesis with Synplify Pro against Virtex-7.

Design	**DSP-slices**	**LUTs**	**FFs**	**max freq. (MHz)**
Floating point	252	26044	30576	507.5
Fixed point Core	256	6141	5284	381.1
Fixed point NTD	256	4049	7939	250.3

C.2 Results for Virtex-5

The following results presented in the tables below is from synthesis and place and route against the Virtex-5 device XC5VSX240T. The designs tested are both fixed point implementations, the 32-bit floating point implementation and the design with 24-bit floating point data. Table C.3 shows the result after synthesis with both XST and Synplify Pro for the complex multiplier component.

Table C.3: Resource utilization for complex multiplication, Virtex-5.

Design	Synthesis Tool	DSP-slices	LUTs	FFs	max freq. (MHz)
Floating point	Synplify Pro	12	908	1412	376.8
Fixed point Core	Synplify Pro	16	194	296	384.2
Fixed point NTD	Synplify Pro	16	66	550	306.6
Floating point	XST	12	920	1412	403.7
Fixed point Core	XST	16	194	299	427.8
Fixed point NTD	XST	16	165	323	443.2

The parallel multiplier has too many I/O pins to place and route, but table C.4 presents the result after synthesis. Both Synplify Pro and XST are used.

Table C.4: Resource utilization for parallel complex multiplication against Virtex-5.

Design	Synthesis Tool	DSP-slices	LUTs	FFs	max freq. (MHz)
Floating point	Synplify Pro	252	22298	32432	343.9
Fixed point Core	Synplify Bro[1]	256	4994	6212	354.1
Fixed point NTD	Synplify Pro	256	4049	9261	212.4
Floating point	XST	252	22490	32432	403.7
Fixed point Core	XST	256	5024	6243	427.8
Fixed point NTD	XST	256	3645	7979	443.2

When synthesising the $\boldsymbol{s}^H \hat{\boldsymbol{R}}_{sp}^{-1} \boldsymbol{s}$ and $\boldsymbol{s}^H \hat{\boldsymbol{R}}_{sp}^{-1} \boldsymbol{x}$ component, Synplify Pro did not manage to perform the synthesis due to problems with reading the files containing the steering vectors for this design. The result after synthesis with XST is presented in table C.5.

Table C.5: Resource utilization for shrsx-top after synthesis with XST against Virtex-5.

Design	Target Platform	DSP-slices	LUTs	FFs	max freq. (MHz)
Floating point	XST	276	36597	60188	142.6
Fixed point Core	XST	288	5669	12262	427.8
Fixed point NTD	XST	288	5290	12073	371.3

[1] Barney Stinson