Implementation of Parametric Haar-like Transformations on FPGA

Mikko Koverola

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 14.5.2018

Supervisor

Prof. Jussi Ryynänen

Advisor

Ph.D. David Guevorkian

Aalto University
School of Electrical
Engineering

Aalto University
School of Electrical
Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Mikko Koverola

Title Implementation of Parametric Haar-like Transformations on FPGA

Degree programme Master's Programme in Nano and Radio Sciences

Major Micro- and Nanoelectronic Circuit Design **Code of major** ELEC3036

Supervisor Prof. Jussi Ryynänen

Advisor Ph.D. David Guevorkian

Date 14.5.2018 **Number of pages** 72+7 **Language** English

Abstract

Linear transformations are used in many algorithms and applications that are implemented in hardware. In this master's thesis a parametrized representation, called a parametric Haar-like transformation, is presented for a family of Haar-like linear transformations. An advantage of this parametric representation is that it can be implemented using a unified hardware architecture. The aim of this thesis is to study how the hardware architectures for parametric Haar-like transformations could be efficiently implemented as a part of a larger FPGA based system.

First hardware architectures for the transformations were investigated in VHDL and the final synthesizable RTL models were implemented with HLS. With HLS complex and real number implementations for flat hierarchy and class based hierarchy style descriptions were created for a variety of transformation sizes and synthesized to an Intel Stratix 10 FPGA. The synthesized implementations were characterized with respect to speed, latency, throughput and resource usage. In general, the class based hierarchies were found to be more suitable for FPGA implementations when increased throughput and faster clock rates are wanted. The flat hierarchies had a more algorithmic implementation style and were found to have slower clock rates and decreased throughput. Although, the flat hierarchies did consume less resources on the FPGA than the class based implementations.

Keywords Linear Transformations, FPGA, Haar-Transformation, VHDL, High-Level Synthesis, Catapult HLS

Aalto University
School of Electrical Engineering

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Mikko Koverola

Työn nimi Parametrisoitujen Haar-kaltaisten muunnosten implementointi FPGA:lle

Koulutusohjelma Nano- ja radiotieteiden maisteriohjelma

Pääaine Mikro- ja nanoelektroniikkasuunnittelu **Pääaineen koodi** ELEC3036

Työn valvoja Prof. Jussi Ryynänen

Työn ohjaaja Ph.D. David Guevorkian

Päivämäärä 14.5.2018 **Sivumäärä** 72+7 **Kieli** Englanti

Tiivistelmä

Lineaarimuunnoksia tavataan monissa algoritmeissa ja sovelluksissa, jotka ovat implementoitu laitteistoilla. Tässä diplomityössä esitetään parametrisoitu esitystapa, jota kutsutaan parametrisoiduksi Haar-kaltaiseksi muunnokseksi, perheelle Haar-kaltaisia lineaarimuunnoksia. Etuna parametrisoidussa esitystavassa on, että Haar-kaltainen muunnos pystytään implementoimaan yhtenäisellä laitteistoarkkitehtuurilla. Tämän työn tavoite on selvittää miten parametrisoituja Haar-kaltaisia muunnoksia voisi implementoida tehokkaasti osana isompaa FPGA-pohjaista järjestelmää.

Aluksi laitearkkitehtuureja muunnoksille tutkittiin VHDL:llä, jonka jälkeen syntetisoitavat RTL-mallit implementoitiin HLS:ää käyttäen. HLS:llä luotiin sekä kompleksi- että reaaliluku implementaatiot tasaistetulle ja luokkahierarkia tyyliselle kuvauksille, jotka syntetisoitiin Intel Stratix 10 FPGA:lle. Syntetisoidut implementaatiot karakterisoitiin nopeuden, latenssin, läpisyötön ja resurssien kulutuksen kannalta. Yleisesti ottaen, luokkahierarkia tyyliset implementaatiot ovat sopivampia FPGA implementoinnille, kun halutaan suurempia läpisyöttöjä ja kellotaajuuksia. Tasaistetuissa hierarkioissa oli algoritmisempi implementointi tyyli ja ne saavuttivat hitaampia kellotaajuuksia ja läpisyöttöjä. Kuitenkin, tasaistetut hierarkiat käyttivät yleisesti ottaen vähemmän resursseja FPGA:lla kuin luokkahierarkia implementaatiot.

Avainsanat Lineaarimuunnokset, FPGA, Haar-muunnos, VHDL, High-Level Synthesis, Catapult HLS

Preface

The work for this master's thesis was done as a part of a larger project investigating hardware acceleration of beamforming algorithms in Nokia and the thesis was written for the Department of Electronics and Nanoengineering in Aalto University School of Electrical Engineering.

First I would like to thank my advisor David Guevorkian for giving me the freedom to work on this interesting topic and providing excellent guidance when needed. Then I would like to thank my thesis supervisor Jussi Ryynänen for advice and guidance throughout the whole master's thesis procedure. I would also like to express my gratitude to my line manager Jaako Maunuksela for understanding and providing flexibility to balance the working time between this master's thesis and other work. I would also like to acknowledge all my colleagues at Nokia for providing direct and indirect support during the master's thesis. Lastly, a special thank you is in order for Richard Toone from Mentor Graphics. Without his excellent support this master's thesis could not have been done.

Otaniemi, 14.5.2018

Mikko Koverola

Contents

Symbols and abbreviations

Symbols

T, S	linear transformations
U, V	vector spaces
N	size of an n by n matrix or length n vector
N_j	number of spectral kernels in j-th stage sparse block diagonal matrix
N_{PEtot}	total number of processing elements in a parametric Haar-like transformation
N_{inputs}	number of inputs to a parametric Haar-like transformation
N_{Stage}	number of stages in a parametric Haar-like transformation
I_{Nmod2}	identity matrix if $Nmod2 = 1$, empty matrix if $Nmod2 = 0$
$\mathbf{C}, \mathbf{K}, \mathbf{A}$	transformation matrices
$\mathbf{G}$	Givens rotation matrix
$\mathbf{Q}$	orthogonal matrix
$\mathbf{R}$	right upper triangular matrix
$\mathbf{X}$	input matrix
$\mathbf{Y}$	output matrix
$\mathbf{W}_n$	classical n by n Haar-Wavelet transformation matrix
$\mathbf{H}_n$	transformation matrix of a parametric Haar-like transformation
$\mathbf{H}_j$	j-th stage sparse block diagonal matrix
$\mathbf{P}_j$	j-th stage permutation matrix
$\mathbf{V}_{js}$	j-th stage's s-th spectral kernel
$\mathbf{h}$	generating vector
$\mathbf{x}$	input vector
$\mathbf{y}$	output vector
$\mathbf{x}_j$	j-th stage output vector
$\mathbf{x}_{isub}$	spectral kernel's two-element input sub-vector
$\mathbf{x}_{osub}$	spectral kernel's two-element output sub-vector
$\mathbf{a}_n, \mathbf{b}_n$	basis vectors
$\mathbf{u}, \mathbf{v}$	complex vectors
u_{js}	spectral kernel element u of j-th stage's s-th spectral kernel
v_{js}	spectral kernel element v of j-th stage's s-th spectral kernel
c_j, a, b	complex coefficients
c_{ij}, a_{ij}	complex matrix elements
$x_{j_{nm}}$	j-th stage output matrix elements
x_{osub0}, x_{osub1}	two-element output sub-vector's vector elements
x_{isub0}, x_{isub1}	two-element input sub-vector's vector elements
n, i, j, s, p, m, k	integer parameters
s_{ji}, c_{ji}	Givens rotation matrix elements
ϕ, θ	spectal kernel basis parameters

Operators

$\prod_j^1$	product over from index j to index 1
$\lceil\ \rceil$	rounding up, ceil
$\lfloor\ \rfloor$	rounding down, floor
$\|\ \|$	absolute value
$\|\|\ \|\|$	norm
$(\ )^*$	complex conjugation
$[\ ]^T$	transpose
$\oplus$	direct sum
$\oplus_s^k$	direct sum from index s to index k
$\otimes$	Kronecker product
$O()$	Ordo notation

Abbreviations

AC	Algorithmic C
ALM	Adaptive Logic Module
ANSI	American National Standards Institute
ASIC	Application Specific Integrated Circuit
CORDIC	Coordinate Rotation Digital Computer
DCT	Discrete Cosine Transformation
DFT	Discrete Fourier Transformation
DUT	Device Under Test
DSP	Digital Signal Processing
FFT	Fast Fourier Transformation
FIFO	First In First Out, A buffer component
FPGA	Field Programmable Gate Array
FSM	Finite-Stage Machine
GUI	Graphical User Interface
HDL	Hardware Description Language
HLS	High-Level Synthesis
IP	Intellectual Property
IEEE	Institute of Electrical and Electronics Engineers
LUT	Look-up Table
M20K	20Kbit Embedded memory element
MLAB	Memory Adaptive Logic Module
MSB	Most Significant Bit
N-R	Newton-Raphson iteration
PE	Processing Element
PWL	Piecewise Linear
RAM	Random Access Memory
RTL	Register-Transfer Level
VHDL	Very High Speed Integrated Circuit Hardware Description Language

1 Introduction

Linear transformations are widely used in a variety of applications. Every linear transformation can be expressed as a multiplication between a transformation matrix and its input vectors. These transformation matrices can be applied sequentially and iteratively to input vectors in more complex algorithms. Often, the algorithms utilize unitary transformation matrices due to their special mathematical properties.

Unitary transformations are used in many applications in multimedia and wireless communication. One of the most known unitary transformation is the Discrete Fourier Transformation (DFT) [1], which transfers signals to spectral domain. The invention of the well-known Cooley-Tuckey Fast Fourier transformation (FFT) [2] algorithm for DFT computation accelerated the development of signal processing and communication systems. Since the invention of FFT, many other transformations such as the Discrete Cosine Transform (DCT), Walsh-Hadamard transformation, or Haar transformation have also found numerous applications, partially due to the possibility of computing them using fast algorithms similar to FFT.

The transformations can be designed to have various desired properties such as light computational complexity, higher data compression ability, higher spectral efficiency or higher decorrelation. In fact, one of the major applications of linear algebra is to construct linear transformations with desired properties. In the case of this master's thesis, one of the main properties is a parametric representation of a transformation matrix that can be efficiently mapped to a hardware implementation.

In this master's thesis linear transformations, called Haar-like transformations, are implemented in a parametric way on a Field Programmable Gate Array (FPGA). The Haar-like transformations are specified from a generalized parametric equation that can describe many different families of linear transformations. By defining the parameters from the generalized equation correctly, the desired Haar- like transformation can be synthesized. Thus, the linear transformations are named parametric Haar-like transformations [3][4][5].

1.1 Application

An advantage of having a general parametrized expression for a large set of linear transformations is the ability to implement a variety of linear transformations with a unified software and hardware architecture. Moreover, the linear transformation matrix can be fine-tuned to changing inputs to give desired outputs by adjusting the parameters. [6]

The generalized equation can be described as a flow graph that can be easily mapped to a hardware implementation. The generalized equation and flowgraph can be further specified to describe a specific Haar-like transformation by selecting a specific set of parameters. The parametric Haar-like transformations can be implemented with a unified hardware architecture that uses generating vectors to define the Haar-like transformations. By re-generating a given Haar-like transformation with another generating vector the transformation can be fine-tuned and adjusted to changing inputs. [3][4][5]

It has been demonstrated that this method of describing parametric Haar-like transformations could be applied in multimedia systems to implement image compression [6] and denoising [7]. Recently, it has been found that parametric Haar-like transformations can be efficiently used also in telecommunication systems, particularly in beamforming applications. Since the requirements for the beamforming algorithms are demanding, hardware acceleration is needed for the algorithms. This master's thesis was done as a part of a project studying hardware acceleration for these beamforming algorithms done at Nokia.

The aim of this work is to study how parametric Haar-like transformations could be efficiently implemented as a part of a more complex telecommunications or multimedia system on an FPGA platform. FPGA implementations can reduce the time-to-market of a product by faster prototyping. Although, the programmable logic on an FPGA can be at least 10 times less efficient than an Application Specific Integrated circuit (ASIC) in power consumption and performance [8]. After the FPGA implementation is mature enough, the product can be migrated to ASICs for improved performance.

1.2 Field Programmable Gate Arrays

FPGAs are a part of the *»gate-array»* design paradigm that aims to lower the nonrecurring costs by avoiding a full fabrication process for a digital circuit. ASICs are hardware components specifically manufactured to implement a desired functionality. Thus, they need to go through a specific and full manufacturing process. As opposed to ASICs, gate-arrays are pre-manufactured generic arrays of logic gates that can be configured to implement the desired functionality by fabricating only the interconnections between the logic. [8]

FPGAs extend this idea by making the interconnections and logic programmable. The programmable interconnection network can be implemented in a variety of ways. FPGA types include write-once, non-volatile and volatile FPGAs. In the write-once FPGAs the interconnection network can be programmed only once by for example running currents through the connection elements. Depending on the connection element type the connection element is either permanently opened or closed by the currents. In non-volatile FPGAs the interconnection programming is stored into non-volatile memories and in volatile FPGAs the interconnection is stored into volatile memories. The stored interconnection programming is then used to configure a transistor based interconnection network. [8]

The programmable logic inside the FPGAs can be implemented as array-based or cell-based logic. Only cell-based FPGAs are considered in this work. The logic cells inside the FPGA fabric may consist of multiple gates and components that can be configured to implement logic functions. Depending on the vendor, these programmable logic cells have different names. [8]

The FPGA device chosen for this master's thesis was Intel's (formerly Altera) Stratix 10 SX series 1SX280LN3F43I1VG with a speed grade 1 (fastest grade). This FPGA type is non-volatile and cell-based. In Intel FPGAs the logic cells are called Logic Array Blocks (LABs) and each LAB includes multiple Adaptive Logic Modules

(ALMs) that can be configured to implement the logic functions. The ALMs consist of a Look-Up Table (LUT), multiplexers, adders and registers. The Stratix 10 SX series FPGAs can include several hundreds to millions of ALMs [9]. In intel FPGAs the LABs may be also configured as memory elements called Memory Logic Array Blocks (MLABs) that are LUT based 640 bit Random Access Memories (RAMs). Also, larger 20 Kbit embedded memory elements called M20Ks are included in the FPGA. The MLABs are more numerous and distributed with a finer granularity in the FPGA fabric than the M20Ks. By utilizing these memory elements the ALM and register usage in the FPGA can be reduced. Although, this comes with a cost of increased latency. [8][10][11]

1.3 Implementation Methods

Architectural design exploration for the hardware architectures of the parametric Haar-like transformations was performed with a Hardware Description Language (HDL). HDLs provide a method to raise the abstraction level in the design process. By raising the abstraction level irrelevant details are hidden from the designer and thus making the design tasks faster. This is usually facilitated by automating the tasks on the previous lower levels. The levels of abstraction describing a digital circuit can be divided as transistor-level, gate-level, register-transfer-level (RTL) and processor-memory-switch- level [12]. As the design techniques have become more mature the abstraction level has been raised steadily from the transistor-level in the 1970s to the gate-level in 1980s and to RTL in 1990s [13].

HDLs operate on the RTL abstracting the transistor- and gate-level design tasks by describing the digital circuit in code. The lower level tasks are automated by synthesis tools that translate the HDL descriptions of the digital circuit into a netlist. The netlist can be then used in transistor-level or gate-level design and implementation. For FPGAs the netlist is used to generate the programming file that configures the interconnection and logic inside the FPGA fabric. The HDL chosen was the VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL).

During the VHDL exploration it became apparent that a faster design method was needed. Thus, the final FPGA implementations were to be done with High-Level Synthesis (HLS) tools. The HLS approach provides yet another way to raise the abstraction level in the design process. Instead of describing the system with traditional HDLs, the behavior and functionality of the system is described with a higher-level programming language. The RTL models are then extracted from the higher-level functional description by the HLS tools and the RTL model can then be used to generate the netlist.

The HLS tool chosen was Mentor Graphics' Catapult HLS. With Catapult HLS the C++ programming language can be used to describe the system functionality by adhering to the Catapult HLS C++ design rules. From the C++ descriptions the Catapult HLS tool is able to extract the RTL models in VHDL. The RTL models were in turn synthesized for the 1SX280LN3F43I1VG FPGA device with Intel's Quartus Prime Pro 18.0 IR2 synthesis tool.

The performance of the synthesized implementations were characterized by performance metrics for speed, latency, throughput, and resource usage. The maximum achieved clock rate was used to characterize the speed of the transformation hardware and the resource usage on the FPGA was characterized by the total ALM and register utilization. The latency and throughput were reported in clock cycles. All the metrics were automatically generated by the Quartus Prime Pro and Catapult HLS tools.

1.4 Thesis Structure

In Chapter 2 the parametric Haar-like transformations are described. First, a brief theoretical introduction to linear transformations is given in Section 2.1. Then in Section 2.2 the parametric Haar-like transformations are described in detail and in Section 2.3 a brief overview of inverse square root implementations on FPGAs is given. The inverse square root calculation plays a central role in the hardware implementation of the parametric Haar-like transformations.

In Chapter 3 the implementation methods are covered in detail. In Section 3.1 the HDL models are described and in Section 3.2 the methods used to extract the final RTL models using Catapult HLS are presented. In Chapter 4 the results are presented. First the processing elements are characterized in Section 4.1 and then the results for the class based hierarchy designs are presented in Section 4.2. In Section 4.3 the results for the flat hierarchy designs are presented. Finally, answers to the design questions presented in Section 3.1.4 are formulated in in Section 4.4. In Chapter 5 the conclusions for this thesis are presented.

2 Parametric Haar-like Transformations

In this chapter, mathematical and theoretical considerations of the parametric Haar-like transformations are presented. To understand how linear transformations can be applied in different applications, a brief introduction to linear transformations is done in Section 2.1. In Section 2.2 the actual parametric Haar- like transformations are presented and the methods used in mapping the transformations into a hardware architecture are described in detail.

Last, a short review on inverse square root calculations on FPGAs is presented in Section 2.3. The inverse square root calculation should be given special attention when mapping the parametric Haar-like transformations to a hardware architecture.

2.1 Linear Transformations

A transformation maps an input of the transformation to an output, for example a function is a transformation. A linear transformation does this linearly. In other words, for transformation T to be linear it must meet two requirements.

$$T(\mathbf{u}+\mathbf{v}) = T(\mathbf{u}) + T(\mathbf{v}) \tag{1}$$

$$T(a\mathbf{u}) = aT(\mathbf{u}) \tag{2}$$

These requirements are also known as additivity (1) and homogeneity (2) requirements. Equations (1) and (2) can be combined into a single requirement.

$$T(a\mathbf{u}+b\mathbf{v}) = aT(\mathbf{u}) + bT(\mathbf{v}) \tag{3}$$

A linear transformation matrix can be constructed with a special set of vectors called the basis vectors. In short, the basis vectors are a set of linearly independent vectors that span a vector space. Every vector in the space, that the basis will span, can be expressed as a linear combination of the basis vectors. If the linear transformation for the basis is known then linear transformations for every vector inside the vector space are known. Thus, the basis vectors are of special interest. [14]

Consider a set of basis vectors $\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3, \ldots, \mathbf{b}_j$ that span a vector space. An input vector in the vector space is a unique linear combination of the basis vectors and can be written as

$$\mathbf{u} = c_1\mathbf{b}_1 + c_2\mathbf{b}_2 + c_3\mathbf{b}_3 + \cdots + c_j\mathbf{b}_j \tag{4}$$

If we solve the linear transformation T for the basis vectors, the linear transformation must produce the output for the input vector $\mathbf{u}$ according to the condition presented in equation (4) and we get the output vector of the transformation as

$$T(\mathbf{u}) = c_1T(\mathbf{b}_1) + c_2T(\mathbf{b}_2) + c_3T(\mathbf{b}_3) + \cdots + c_jT(\mathbf{b}_j) \tag{5}$$

Let a linear transformation T map an input from an j-dimensional vector space U with a basis $\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3, \ldots, \mathbf{b}_j$ to an output in an i-dimensional vector space V with

a basis $\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3, \ldots, \mathbf{a}_i$. Thus, we can express the basis of the input space U in terms of the basis of the output space V according to equation (5) as

$$\begin{cases} T(\mathbf{b}_1) = c_{11}\mathbf{a}_1 + c_{21}\mathbf{a}_2 + c_{31}\mathbf{a}_3 + \cdots + c_{i1}\mathbf{a}_i \\ T(\mathbf{b}_2) = c_{12}\mathbf{a}_1 + c_{22}\mathbf{a}_2 + c_{32}\mathbf{a}_3 + \cdots + c_{i2}\mathbf{a}_i \\ T(\mathbf{b}_3) = c_{13}\mathbf{a}_1 + c_{23}\mathbf{a}_2 + c_{33}\mathbf{a}_3 + \cdots + c_{i3}\mathbf{a}_i \\ \quad \vdots \\ T(\mathbf{b}_j) = c_{1j}\mathbf{a}_1 + c_{2j}\mathbf{a}_2 + c_{3j}\mathbf{a}_3 + \cdots + c_{ij}\mathbf{a}_i \end{cases} \tag{6}$$

Thus, we can express the linear transformation T as a matrix $\mathbf{C}$ constructed from the coefficients c_{ij}.

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & c_{13} & \ldots & c_{1j} \\ c_{21} & c_{22} & c_{23} & \ldots & c_{2j} \\ c_{31} & c_{32} & c_{33} & \ldots & c_{3j} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{i1} & c_{i2} & c_{i3} & \ldots & c_{ij} \end{pmatrix} \tag{7}$$

When the matrix of the linear transformation represents the basis correctly, linearity will guarantee the correct output vector according to the transformation. This illustrates the fact that every linear transformation can be expressed as a multiplication between a transformation matrix and its input vectors. [14]

To illustrate how linear transformations can be applied, a simple and contrived example of a QR-decomposition using Givens rotations is presented in appendix A. The QR-decomposition algorithm used in the example is presented in [1]. Moreover, a similar entity to the 2 by 2 Givens rotation, called a spectral kernel, is defined later in section 2.2 when the parametric Haar-like transformations are described. This example is intended to give the reader more insight to the functionality of the parametric Haar-like transformations.

The subject of this master's thesis are the implementations of linear transformations called parametric Haar-like transformations [3][4][5]. The transformations from this family are unitary and can be computed using fast algorithms of linear complexity. An important property of a parametric Haar-like transformation is the possibility to adapt the transformation basis to include a predefined set of basis vectors. [3]

The Haar-like transformations, can be applied in various algorithms that have applications in telecommunications or multimedia systems. For example, image compression [6] and denoising [7] can improved using parametric Haar-like transformations. Recently, it has been shown that these transformations can also be used in telecommunications systems, specifically in baseband beamforming algorithms.

In a larger context, the work performed in the scope of this master's thesis was a part of a project for hardware acceleration of baseband beamforming algorithms in telecommunications systems. The aim of this thesis is to map the parametric Haar-like transformations into efficient hardware implementations that can be utilized as a part of a larger FPGA based telecommunications or multimedia systems.

2.2 Parametric Haar-like Transformations

In this section parametric Haar-like transformations introduced in [3][4][5] are described in detail. First a generic representation for parametric unitary transformations is discussed in Section 2.2.1. In Section 2.2.2 the generic representation presented in Section 2.2.1 is specified to describe Haar-like transformations. Last, a brief overview on how the Haar-like transformations could be mapped onto hardware is given in Section 2.2.3

2.2.1 Parametric Representation of Unitary Transformations

A discrete unitary transformation is represented by the matrix equation

$$\mathbf{Y} = \mathbf{H}_n\mathbf{X} \tag{8}$$

where $\mathbf{X}$ is an n by m input matrix and $\mathbf{Y}$ is an n by m output matrix for the transformation. The matrix $\mathbf{H}_n$ is the unitary transformation matrix. In general, the matrix equation (8) requires $O(mn^2)$ operations for computation. However, using so called fast transformations the matrix equation can be computed using fewer operations. As was shown in [3], many classes of fast transformations can be presented using a unified equation

$$\mathbf{H}_n = \mathbf{P}_{m+1}\prod_{j=m}^{1}(\mathbf{H}_j\mathbf{P}_j) = \mathbf{P}_{m+1}(\mathbf{H}_j\mathbf{P}_j)(\mathbf{H}_{j-1}\mathbf{P}_{j-1})\dots(\mathbf{H}_2\mathbf{P}_2)(\mathbf{H}_1\mathbf{P}_1) \tag{9}$$

where $\mathbf{H}_j$ are sparse block diagonal matrices and $\mathbf{P}_j,\ \ j = 1,\dots,m+1$ are permutation matrices. Equation (9) can be used to describe multiple families of fast orthogonal transformations such as FFT, Fast Cosine, Walsh-Hadamard, Vilenkin-Krestenson and Haar transformations. The sparse block diagonal matrices $\mathbf{H}_j$ have smaller 2 by 2 orthogonal matrices $\mathbf{V}_{js}$, called spectral kernels, as their block diagonal entries. With the equations presented in [4] the block diagonal sparse matrices can be represented by the equation

$$\mathbf{H}_j = (\oplus_{s=0}^{k-1}(\mathbf{V}_{js})) \oplus I_{Nmod2} \oplus (\oplus_{s=k}^{\lfloor\frac{N}{2}-1\rfloor}(\mathbf{V}_{js})),\ \ k \in \{0,1,...,\lfloor N/2\rfloor\} \tag{10}$$

where the operator $\oplus$ is the direct sum, N is the dimension of the transformation matrix and I_{Nmod2} is an identity matrix of order 1 when $N\ mod\ 2 = 1$ or an empty 0 by 0 matrix when $N\ mod\ 2 = 0$. The 2 by 2 spectral kernels are defined as

$$\mathbf{V}_{js} = \begin{pmatrix} u_{js} & v_{js} \\ v_{js} & -u_{js} \end{pmatrix} = \begin{pmatrix} \cos(\phi) & e^{i\theta}\sin(\phi) \\ \sin(\phi) & -e^{i\theta}\cos(\phi) \end{pmatrix} \tag{11}$$

where θ and ϕ are parameters that define the basis of the transformation. When the transformation matrix $\mathbf{H}_n$ is represented with equations (9) and (10), the discrete unitary transformation in by equation (8) can be computed with a fast transform algorithm that consists of m stages. The algorithm can be described by the set of equations

$$\begin{cases} \mathbf{x}_0 = \mathbf{x} \\ \mathbf{x}_j = \mathbf{H}_j\mathbf{P}_j\mathbf{x}_{j-1}, \quad where \quad j = 1,\dots,m \\ \mathbf{y} = \mathbf{P}_{m+1}\mathbf{x}_m \end{cases} \tag{12}$$

where $\mathbf{x}$ is the original input vector to the transformation, $\mathbf{x}_j$ is the j-th stage output vector and $\mathbf{y}$ is the output vector. [4]

Below an example of an arbitrary j-th stage matrix equation is given showing the block diagonal j-th stage transformation matrix $\mathbf{H}_j$, the permutation matrix $\mathbf{P}_j$, the output and input vectors $\mathbf{x}_j$ and $\mathbf{x}_{j-1}$.

Example 2.2.1, The fast transform algorithm written explicitly

This example explicitly shows the j-th stage of an arbitrary transformation of size $N = 4$ to give the reader clearer understanding of the algorithm described by the set of equations (12). In the example the permutation matrix is chosen to be the identity matrix.

$$\mathbf{x}_j = \mathbf{H}_j\mathbf{P}_j\mathbf{x}_{j-1} \Leftrightarrow \begin{pmatrix} x_{j_1} \\ x_{j_2} \\ x_{j_3} \\ x_{j_4} \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} u_{j0} & v_{j0} \\ v_{j0} & \text{-}u_{j0} \end{pmatrix} & 0 \\ 0 & \begin{pmatrix} u_{j1} & v_{j1} \\ v_{j1} & \text{-}u_{j1} \end{pmatrix} \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_{(j\text{-}1)_1} \\ x_{(j\text{-}1)_2} \\ x_{(j\text{-}1)_3} \\ x_{(j\text{-}1)_4} \end{pmatrix} \tag{13}$$

The algorithm described by the set of equations (12) can be further expanded into a generalized algorithmic expression that implements fast matrix-to-matrix multiplications. The main idea is to process the input matrix $\mathbf{X}$ column by column through all the $j = 1, 2, 3, \ldots, m$ stages of the algorithm. The input matrix could also be processed also row by row as long as the input vector usage is consistent throughout the algorithm execution. The generalized algorithm is described below.

Algorithm 2.2.1, The Generalized Fast Matrix Transformation Algorithm

1. Take the first column (or row) vector $\mathbf{x}_0$ from the input matrix $\mathbf{X}$ as the input vector

2. Permutate the components of the input vector $\mathbf{x}_0$ according to the first stage permutation matrix $\mathbf{P}_1$ and set the stage index j to $j = 1$

3. Partition the permutated vector into two-element sub-vectors $\mathbf{x}_{sub(0)}, \mathbf{x}_{sub(1)}, \ldots, \mathbf{x}_{sub(N_j)}$, where N_j is the number of spectral kernels in the current j-th stage transformation matrix $\mathbf{H}_j$ and $\mathbf{x}_{sub(s)}$ is a two-element sub-vector. If the length of the stage input vector is odd, bypass one of its vector elements to the next stage.

4. Multiply each sub-vector $\mathbf{x}_{sub(0)}, \mathbf{x}_{sub(1)}, \ldots, \mathbf{x}_{sub(N_j)}$ with the corresponding spectral kernels $\mathbf{V}_{js}$ in the current j-th stage transformation matrix $\mathbf{H}_j$.

5. Concatenate the output sub-vectors from each multiplication with the corresponding spectral kernels to form the output vector $\mathbf{x}_j$ of the j-th stage

6. Set the stage index j to $j = j + 1$ and permutate the output vector $\mathbf{x}_j$ according to the permutation matrix $\mathbf{P}_j$

7. Repeat the steps from 3 to 6 for all the m stages of the transformation untill the stage index j is set to $j = m$

8. After all the m stages have processed their vectors, the final output vector $\mathbf{x}_m$ is obtained

9. The first column (or row) vector $\mathbf{y}_0$ of the final output matrix $\mathbf{Y}$ is then given by the equation $\mathbf{y}_0 = \mathbf{P}_{m+1}\mathbf{x}_m$

10. Repeat steps from 1 to 9 for all the other column (or row) vectors of the input matrix $\mathbf{X}_0 = \mathbf{X}$

11. The final output matrix $\mathbf{Y}$ is formed after all the column (or row) vectors of the input matrix $\mathbf{X}_0 = \mathbf{X}$ have been processed through all the $j = 1, 2, 3, \ldots, m$ stages

The algorithm 2.2.1 is represented as a generalized flow graph in figure 1.

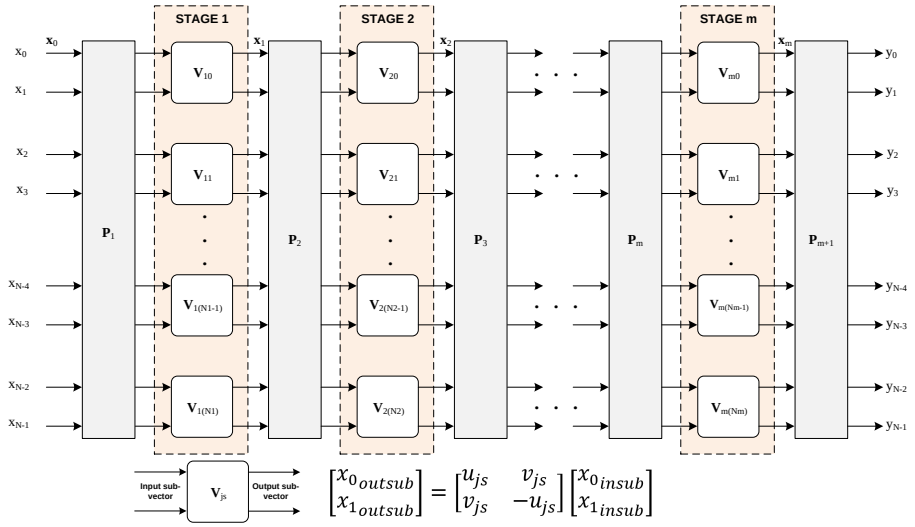

Figure 1: A generalized flow graph of the algorithm 2.2.1 [5].

The equations (9), (10) and (11) can be used to describe multiple orthogonal transformation matrices that can be mapped to a flow graph representation. By varying the integer parameters m and k, the permutation matrices and the spectral kernels different families of transformation matrices (FFT, Fast Cosine, Walsh-Hadamard, Haar...) can be described. Thus, a parametric representation of a unitary transformation has as parameters *the choice of spectral kernels*, *the choice of permutation matrices*, integer parameters m and k. [5][4][3]

The central argument of this master's thesis is that the algorithm 2.2.1 can be easily mapped to hardware due to its flow graph representation of a regular structure, similar to the FFT (figure 1). The parametric nature of the linear transformations described by the methods introduced in this section will make the synthetization of families of transformation matrices with different basis and sizes on hardware

easy. The equations (9), (10) and (11) can be used to define a parametric Haar-like transformation matrix by specifying the permutation matrices, the spectral kernels and the integer parameters m and k. This is discussed in the next section.

2.2.2 Parametric Representation of Haar-like Transformations

The classical Haar-wavelet transformation matrix presented in [1] is described recursively by the equation (14).

$$\mathbf{W}_n = \begin{cases} \left(\mathbf{W}_p \otimes \begin{pmatrix} 1 \\ 1 \end{pmatrix} \middle| \quad \mathbf{I}_p \otimes \begin{pmatrix} 1 \\ -1 \end{pmatrix} \right), & if \quad n = 2p \\ \\ \begin{pmatrix} 1 \end{pmatrix}, & if \quad n = 1 \end{cases} \tag{14}$$

where $\otimes$ is the Kronecker product and $\mathbf{W}_n$ is the Haar-wavelet transformation matrix of size n, where n is a power of two integer. The Haar-transformation is often defined by convention as the transpose of the Haar-wavelet transformation matrix defined by equation (14). For example, the transpose of $n = 8$ Haar-wavelet transformation matrix $\mathbf{W}_8$ is defined as

$$[\mathbf{W}_8]^T = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{pmatrix} \tag{15}$$

As described in [5], the family of parametric Haar-like transformations is defined by the following three conditions when the transformations can be expressed using equation (9).

1. The elements of the spectral kernels are non-zero for $s = 0, 1, 2, \ldots, (N_j - 1)$, where $N_j = \lfloor N/2^j \rfloor$ is the number of spectral kernels of the j-th stage, where $j = 0, 1, 2, \ldots, m$.

2. The elements of the spectral kernels define the 2 by 2 identity matrix $\mathbf{V}_{js} = I_2$ for the j-th stage, when $s = N_j, \ldots, \lceil N/2^j \rceil, \quad j = 0, 1, 2, \ldots, m$

3. The permutation matrix of the j-th stage is defined as $\mathbf{P}_j = \mathbf{P}_{j1} \oplus I_{N-N_j}$, where $\mathbf{P}_{j1}$ is a permutation matrix of order N_j and the permutation type chosen is the inverse perfect shuffle.

For example, the classical Haar-wavelet transformation matrix, described by the equation (14), is defined by the three conditions when $m = \log_2(N)$, $k = 0$ and $\mathbf{V}_{js} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$, when $j = 0, \ldots, m$ and $s = 0, \ldots, (N_j - 1)$.

In this master's thesis a method described in [5] is considered to construct parametric Haar-like transformations in such a way that they have a predefined vector as their first-row vector. The equation (16) describes a Haar- like transformation matrix $\mathbf{H}_n$ that has as its first-row vector the vector $\mathbf{h}$.

$$\mathbf{y} = \mathbf{H}_n \mathbf{h}^T = [\|\mathbf{h}\|, 0, 0, \ldots, 0]^T, \quad where \quad \mathbf{h} = [h_0, h_1, \ldots, h_{N-1}] \tag{16}$$

In other words, for a given input vector $\mathbf{h}$ a transformation matrix $\mathbf{H}_n$ that satisfies the equation (16) has to be found. Thus, the vector $\mathbf{h}$ is called a *generating vector*.

To repeat, the spectral kernels are defined by the equation (17). When equation (17) is compared to the equation (A2) of Givens rotations presented in appendix A one can immediately see the similarities. In fact, the spectral kernels can be used the same way as Givens rotations to introduce zeros to the output vector by defining the spectral kernel elements in a similar way as was done in the example presented in appendix A.

$$\mathbf{V}_{js} = \begin{pmatrix} u_{js} & v_{js} \\ v_{js} & -u_{js} \end{pmatrix} = \begin{pmatrix} \cos(\phi_{js}) & e^{i\theta}\sin(\phi_{js}) \\ \sin(\phi_{js}) & -e^{i\theta}\cos(\phi_{js}) \end{pmatrix} \tag{17}$$

When the spectral kernel is multiplied with the two-element input sub-vector, as described in the algorithm 2.2.1, we get the equation

$$\mathbf{x}_{osub} = \mathbf{V}_{js}\mathbf{x}_{isub} \Leftrightarrow \begin{pmatrix} x_{osub_0} \\ x_{osub_1} \end{pmatrix} = \begin{pmatrix} u_{js} & v_{js} \\ v_{js} & -u_{js} \end{pmatrix} \begin{pmatrix} x_{isub_0} \\ x_{isub_1} \end{pmatrix} \tag{18}$$

To map the lower element of the output sub-vector to zero, we find the solutions for coefficients u_{js} and v_{js}

$$\begin{pmatrix} x_{osub_0} \\ 0 \end{pmatrix} = \begin{pmatrix} u_{js} & v_{js} \\ v_{js} & -u_{js} \end{pmatrix} \begin{pmatrix} x_{isub_0} \\ x_{isub_1} \end{pmatrix} \Rightarrow$$
$$u_{js} = \frac{x_{isub_0}}{\sqrt{(x_{isub_0})^2 + (x_{isub_1})^2}} \quad \& \quad v_{js} = \frac{x_{isub_1}}{\sqrt{(x_{isub_0})^2 + (x_{isub_1})^2}} \tag{19}$$

For complex numbers, the solutions for spectral kernel coefficients are found to be

$$\mathbf{V}_{js} = \begin{pmatrix} (u_{js})^* & (v_{js})^* \\ v_{js} & -u_{js} \end{pmatrix}, \quad where$$
$$u_{js} = \frac{x_{isub_0}}{\sqrt{|x_{isub_0}|^2 + |x_{isub_1}|^2}} \quad \& \quad v_{js} = \frac{x_{isub_1}}{\sqrt{|x_{isub_0}|^2 + |x_{isub_1}|^2}} \tag{20}$$

where $(v_{js})^*$ and $(u_{js})^*$ are the complex conjugates of the spectral kernel elements and $|x_{isub_0}|$ and $|x_{isub_1}|$ are the absolute values of the complex vector elements. [3][4][5]

As can be seen from equation (19), every spectral kernel can be defined in such a way that the lower element of the output sub-vector is mapped to zero. This means that for any given stage the output vector will have $N_j = \lfloor N/2^j \rfloor$ more zeros than its own input vector. When the permutation matrix is defined to be an inverse perfect shuffle, the zero outputs are shuffled to the bottom half and the non-zero outputs

are shuffled to the top half of the next stage input vector. Clearly, the equation (16) is satisfied after all the spectral kernels in each stage are defined according to the relationships described in equation (19). [4][5]

In other words, the generating vector $\mathbf{h}$ in equation (16) is used to generate all the spectral kernel elements according to the relationships described in equation (19) for real number implementations and according to equation (20) for complex number implementations. When the spectral kernels are generated correctly it follows that equation (16) is satisfied.

By applying all the conditions presented in this section, a left upper triangular flow graph is formed for the parametric Haar-like transformations. Each transformation consists of at least $m = \lceil \log_2(N) \rceil$ stages and the number of spectral kernels that are not defined as identity matrices is approximately halved in each stage. The final permutation matrix $\mathbf{P}_{m+1}$ is defined to be an identity matrix. All the identity matrices (in permutations and spectral kernels) can be replaced with straight interconnects between graph nodes, achieving the left upper triangular form of the flow graph. [4][5]

In figure 2 the left upper triangular flow-graph for $N = 8$ Haar-like transformations and the usage of the generating vector is explicitly is shown.

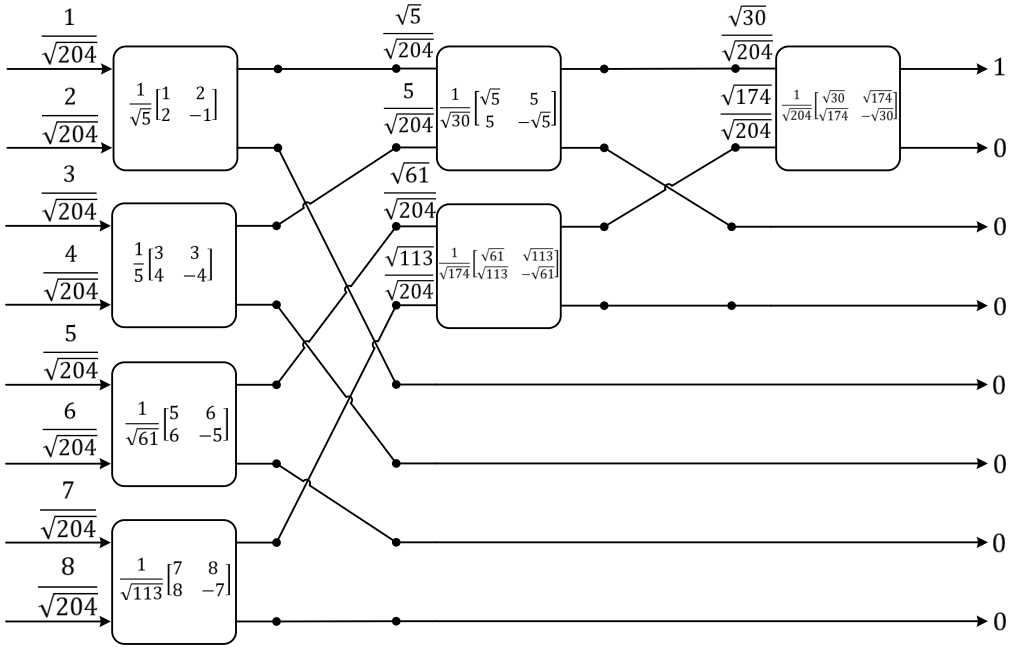

Figure 2: $N = 8$ Haar-transform flow graph showing explicitly the inputs to spectral kernels and the spectral kernels defined from the input sub-vectors according to equation (19)

From the figure 2 it can also confirmed that the output vector satisfies the condition presented in equation (16) and thus the transformation matrix is correctly represented by the spectral kernels.

When the spectral kernels from figure 2 are placed in sparse block diagonal matrices $\mathbf{H}_j$ and the permutation matrices $\mathbf{P}_j$ are represented as inverse perfect shuffles, the Haar-like transformation matrix can be formed according to the equation (9). The Haar-like transformation matrix represented by the spectral kernels in figure is of the form

$$\mathbf{H}_8 = \frac{1}{\sqrt{204}} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 2.4 & 4.8 & 7.2 & 9.6 & -2.1 & -2.5 & -2.9 & -3.3 \\ 5.8 & 11.7 & -3.5 & -4.7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 7.4 & 8.8 & -5.6 & -6.4 \\ 12.8 & -6.4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -8.6 & -8.6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 11 & -9.1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 10.7 & -9.4 \end{pmatrix} \tag{21}$$

When the Haar-like transformation matrix (21) is compared to the transpose of the classical Haar-wavelet transformation matrix presented in equation (15), one can immediately see the similarities and the *»likeness»* of the Haar-like transformations. The described generation procedure for the spectral kernels can also be described as an algortihm. The generation algortihm is presented in detail below.

Algorithm 2.2.2, The Generation Algorithm for Haar-like Transformations

1. Take the generating vector $\mathbf{h}$ as the input vector to the transformation as the generating vector $\mathbf{h}$
2. For parametric Haar-like transformations the permutation matrix $\mathbf{P}_1$ is defined to be an identity matrix and the generating vector $\mathbf{h}$ is passed to the first stage and the stage index j is set to $j = 1$
3. Partition the permutated vector into two-element sub-vectors $\mathbf{h}_{sub(0)}, \mathbf{h}_{sub(1)}, \ldots, \mathbf{h}_{sub(N_j)}$, where $N_j = \lfloor N/2^j \rfloor$ is the number of spectral kernels in the current j-th stage transformation matrix $\mathbf{H}_j$
4. Generate the spectral kernel elements for each spectral kernel $\mathbf{V}_{js}$ in the current j-th stage transformation matrix $\mathbf{H}_j$ using the corresponding generating sub-vector $\mathbf{h}_{sub(0)}, \mathbf{h}_{sub(1)}, \ldots, \mathbf{h}_{sub(N_j)}$ by defining the spectral kernel elements according to equation (19) for real number implementations and according to equation (20) for complex number implementations
5. Concatenate the output sub-vectors from each corresponding spectral kernel as defined in equation 19 to form the output vector $\mathbf{h}_j$ of the j-th stage
6. Set the stage index j to $j = j + 1$, define the permutation matrix $\mathbf{P}_j = \mathbf{P}_{j1} \oplus I_{N-N_j}$ and permutate the output vector $\mathbf{h}_j$ according to $\mathbf{P}_j$
7. Repeat the steps from 3 to 6 for all the m stages of the transformation untill the stage index j is set to $j = m$

8. After all the m stages have processed the generating vector, the transformation matrix $\mathbf{H}_n$ is defined by the spectral kernels and the permutation matrices according to the equation (9) and the final output vector $\mathbf{h}_m = [\|\mathbf{h}\|, 0, 0, \ldots, 0]$ is obtained

A parametric representation of Haar-like transformations that can be mapped onto hardware has been described above. The generic expression for unitary transformations described by equations (9), (10) and (11) was specified to describe Haar-like transformations by fixing the choice of permutation matrices and the spectral kernel elements.

Based on this generic expression a flow graph representation of the parametric Haar-like transformations can be conceived. By utilizing this flow graph representation, a unified hardware architecture can be created for the parametric Haar-like transformations. This unified hardware architecture can be used to implement both algorithms; the algorithm for spectral kernel generation (algorithm 2.2.2) and the algorithm for fast matrix multiplication (algorithm 2.2.1). Thus, the hardware architecture for the parametric Haar-like transformations works in two modes; in a generating mode and a multiplication mode. In the next section a brief overview is given about how the parametric Haar-like transformations can be mapped onto hardware.

2.2.3 Mapping Parametric Haar-like Transformations to Hardware

When the algorithm 2.2.2 is mapped to hardware, it is obvious that the permutation matrices can be implemented as interconnects. The functionality of the parametric Haar-like transformations can be achieved with Processing Elements (PEs). The PEs can operate in two modes, a *generation mode* or a *multiplication mode*.

In generation mode the spectral kerenel elements are generated from the two-element input sub-vectors as described in algorithm 2.2.2. In multiplication mode the PE computes the product of the generated spectral kernel and a two-element input sub-vector as described in algortihm 2.2.1. A series of conceptual schematics illustrating the PE operation, as proposed in [5], are presented below (see figures 3 - 6). In each figure, the signal path is shown in green.

The PE consists of two input branches and a single output branch. A multiplier, an adder-subtractor, a pair of shift registers to store the values of the spectral kernels, a transmission gate that controls the input to the shift register, a logic block that performs an inverse square root operation and a pair of multiplexers that are controlled to pass the correct inputs to the multipliers.

The operation of the PE is as follows. In the generation mode the input sub-vector elements are extracted from the generating vector. These elements are directed to the inputs of the PE at the input ports 1 and 2. First the inverse norm $\frac{1}{\sqrt{(x_{isub_0})^2+(x_{isub_0})^2}}$ needs to be generated. The input values are routed to the multipliers from the input ports and again through the multiplexers, effectively performing the squaring operations $(x_{isub_0})^2$ and $(x_{isub_0})^2$. The squared values are then passed to an adder-subtractor block for addition and the output is finally passed to the inverse square

root logic block that generates the value for the inverse norm. The operations are presented in figure 3.

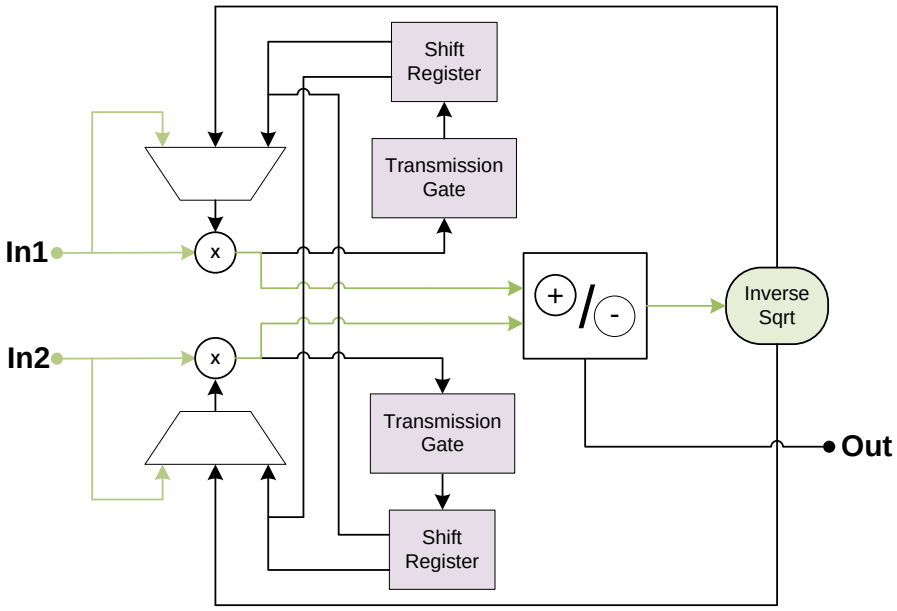

Figure 3: The first step in the PE generation mode is to generate the inverse norm from the input values. The signal path is shown in green.

When the inverse norm is generated, the value is passed to both of the multiplexers and multiplied with the input values. This generates the values for the spectral kernel elements v_{js} and u_{js} that are stored to the shift register of the corresponding input branch. For complex operations additional conjugation and absolute value generation needs to be implemented. The operations are presented in figure 4.

In the multiplication mode, the PE performs the following operations with the two-element input subvectors and the stored spectral kernel elements.

$$x_{osub_0} = u_{js}x_{isub_0} + v_{js}x_{isub_1} \tag{22}$$

$$x_{osub_1} = v_{js}x_{isub_0} - u_{js}x_{isub_1} \tag{23}$$

The first step in the multiplication mode is to multiply the first row of the spectral kernel with the two-element input sub-vector. (equation (22)). The spectral kernel element values are passed from the shift registers to the multiplexers to be multiplied with the inputs. After multiplication the values are added in the adder-subtractor block and passed to the output of the PE. Thus, completing the first row operation. This operation is illustrated in the figure 5.

The second step in the multiplication mode is to multiply the second row of the spectral kernel with the two-element input sub-vector (equation (23)). The values from the shift registers are crossed with the input branches and after multiplication

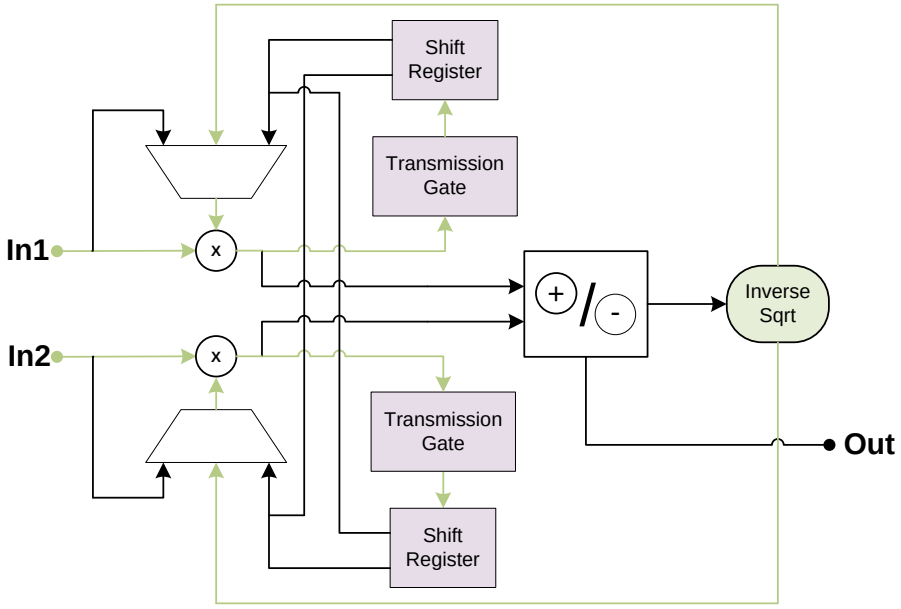

Figure 4: The second step in the PE generation mode is to generate values for the spectral kernels by multiplying the inputs with the inverse norm. The values for the spectral kernel elements are stored in the shift registers. The signal path is shown in green.

the values are passed to the subtractor in the adder-subtractor block and the output is routed to the output port. This operation is illustrated in the figure 6.

By replacing the spectral kernels with processing element blocks in the flow graph, a unified hardware architecture that can be used to implement both the generation algorithm (algorithm 2.2.2) and the fast matrix transformation algorithm (algorithm 2.2.1) for Haar-like transformations. Thus, by utilizing the parametrization introduced in the equations (9), (10) and (11) Haar-like transformations with different sizes can be conveniently described with their flow graph representaitons.

The inverse square root logic block plays an important role when designing the hardware implementations for the PEs. The behavior of the inverse square root logic block will reflect on the behavior of the whole transformation hardware implementation. It is important to design the inverse square root logic block to fit the requirements for the transformation while the requirements for the transformation largely depend on how the transformation will be used as a part of a larger system.

To get a better idea how inverse square root calculations can be implemented on FPGAs, a brief review is presented in the next section. This review is not intended to be an exhaustive literature review on the subject but to give the reader a better idea how inverse square root logic can be implemented on FPGAs.

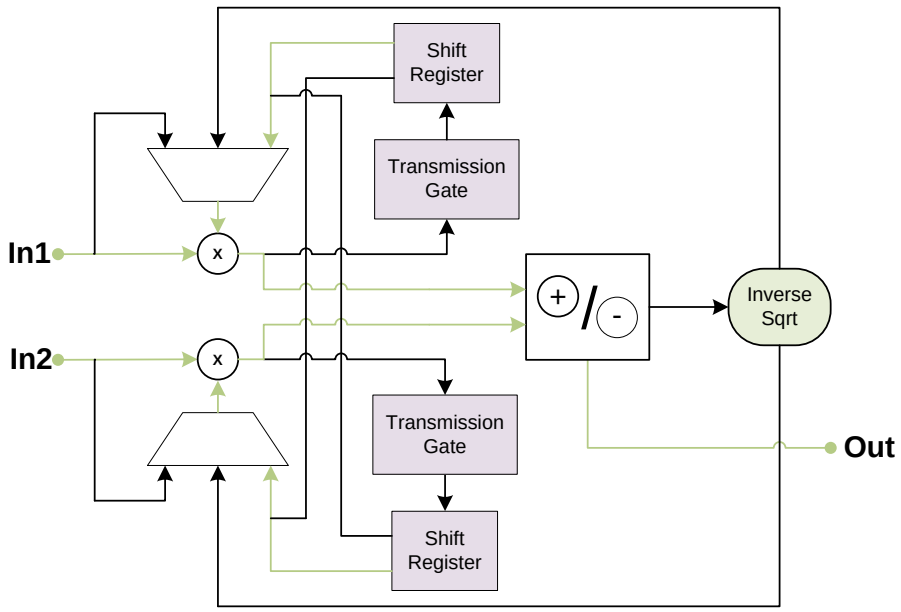

Figure 5: The first step in the PE multiplication mode is to do the first row operation. This is done by utilizing the adder in the adder-subtractor block. The signal path is shown in green.

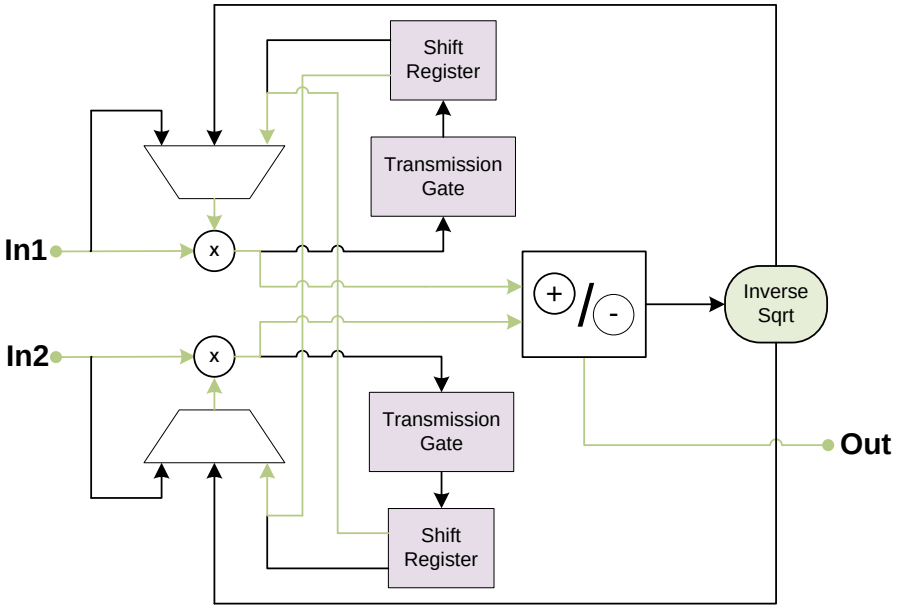

Figure 6: The second step in the PE multiplication mode is to do the second row operation. This is done by utilizing the subtractor in the adder-subtractor block. The signal path shown in green.

2.3 Inverse Square Root Calculation on FPGAs

Choosing the right method for the inverse square root calculation is of utmost importance when implementing the square root logic block for the PEs. The choice can be a complicated selection process between different criteria such as *bitwidth*, *input range* and *approach* [15]. More specified criteria can also be defined depending on the overall design.

Determining the bitwidth for the computations is an optimization task for achieving the needed accuracy with the minimal bitwidth. By carefully considering the bitwidths for computations, efficient usage of the FPGA resources can be ensured. If information about the expected input range for the inverse square root calculation is available a more specialized implementation can be considered. Usually floating-point methods for the inverse square root calculation require the input range to be in the interval $[1, 2)$. This means that scaling logic is needed for the input values to accommodate a broader range of input values. [15]

Fixed-point methods can be usually divided in to reduced range and full range methods. For reduced range methods the input needs to be in the interval $[1, 2)$ and scaling is required. For full range methods no scaling logic is needed and the input value can be used directly for the computation. Common approaches for the inverse square root calculations include such methods as look-up table (LUT) based methods, polynomial approximations and iterative methods. [15]

2.3.1 Common Inverse Square Root Calculation Methods

The LUT based methods are among the simplest and fastest implementations of inverse square root logic. Since LUTs are widely used in FPGAs these methods become attractive for inverse square root calculations on FPGAs. In the most straightforward LUT based methods the results are stored directly into into the LUTs. The drawback of LUT based methods is that for wide bitwidths the size of the LUTs becomes large. The resource usage grows exponentially with the number of address bits needed for the LUTs. For higher precision, the LUTs are often used together with iterative methods. In iterative methods usually the initial value is stored into the LUTs. [15][16]

Series expansions and polynomial approximations are often used in evaluating nonlinear functions. These methods are non-iterative and often based on the Taylor series expansion. For high accuracies in the inverse square root calculation, multiple terms of the Taylor series need to be expanded which translates to an increased number of multipliers and storage elements for the coefficients in hardware. [15][16]

In polynomial approximations a highly nonlinear function, such as the inverse square root, can be approximated with a less nonlinear function. For example, such a method was proposed for ASICs in [18] and a similar method was implemented for FPGAs in [19]. Approximation by piecewise linear functions is also a popular method for inverse square root calculations [15][16][19][20].

Alternatives for series expansions and polynomial approximations are the iterative methods. Perhaps the most widely used iterative method for evaluating the inverse square root is the Newton-Raphson iteration and it is given special attention in this

section. The Newton-Raphson iteration is a root calculation method based on the Taylor series expansion using the first two terms of the series. The attraction to this method is due to its rapid quadratic convergence where the precision of the result roughly doubles each iteration. The method calculates a first initial value for the solution and then iterates the solution to achieve better accuracy. [15][16]

Often, LUT based methods are used to obtain the first initial value for the solution and, after the initial value is formed, adders and multipliers can be used to implement the iteration. Thus, the better the initial value the fewer iterations are needed for a given accuracy and therefore the resource usage can be decreased. To achieve even higher precisions more terms of the Taylor series can be used. The general n-order method is known as the Householder method [21] where the first order Householder method is the Newton- Raphson iteration and the second order Householder method is called Halley's method. [15][16][21]

The inverse square root algorithms applying the Newton-Raphson iteration are also known to use *»magic numbers»* for increased accuracy and speed. One of the first fast inverse square root algorithms that used a magic number has somewhat ambiguous origins in the late 1990s gaming industry. The method is sometimes referred to as simply the *»0x5f3759df method»*, where the hexadecimal number is the actual magic number used. [22][23][24][25]

In effort to explain why some magic numbers work very well in the Newton-Raphson iteration mathematical derivations have been done [24][25]. Floating-points implementations of the Newton-Raphson iteration utilizing magic numbers on FPGAs have also been investigated [22].

Another popular iterative method to implement various nonlinear computations such as the trigonometric functions, the square root and the inverse square root is the Volder's algorithm [26], also known as the Coordinate Rotation Digital Computer (CORDIC). IP blocks for CORDIC are nowadays widely available from different FPGA vendors, which makes this approach attractive for FPGA use [16]. CORDIC can be derived from the general Givens rotation transformations and it is used to rotate input vectors by a specific angle or rotating the input vector onto x-axis while computing the required angle of rotation in a Cartesian plane [26][27][28].

An FPGA implementation of the QR-decomposition algorithm using Givens rotations used the Xilinx CORDIC IP cores to calculate the square root and a division to obtain the inverse square root [29]. Although, lower latency and resource usage was achieved for a QR-decomposition when inverse square root calculation was done using LUT based Newton-Raphson iteration [30]. CORIDC and LUT based Newton-Raphson iteration are also compared in [31].

As the spectral kernels are similar entities to Givens rotations a CORDIC implementation might be a suitable implementation method. The PEs of the Haar-like transformations will most likely have a multicycle inverse square root operation in the generation mode. The multicycle operation of the PEs should be taken into account when designing the hardware architectures for the parametric Haar-like transformations.

3 Implementation

In this Chapter the hardware implementation methods are presented. In Section 3.1 hardware description language models are created for Haar-like transformations of sizes $N = 8, 7, 6, 5, 4, 3, 2$ and the implications of various design choices are discussed. Based on the findings done in Section 3.1 it was decided to implement the final synthesizeable designs using high-level synthesis tools. The implementation of the final hardware architecture with high-level synthesis is discussed in Section 3.2.

3.1 Hardware Description Language Model

The starting point of this master's thesis was a larger Matlab algorithm that applied the Haar-like transformations in matrix form and the transformations needed to be accelerated. First, the Haar-like transformations needed to be described in Matlab so that they could take advantage of the vector processing properties in algorithms 2.2.1 and 2.2.2. After the desired behavior of the algorithms was achieved in the Matlab model, the first steps toward a hardware implementation could be taken.

Since Matlab's m-language executes sequentially (like C or Fortran), the first step in mapping the Matlab description of the Haar-like transformations into hardware was to introduce parallel execution and synchronous behavior under a clock signal to the transformations. This was done by describing the Matlab model in a Hardware Description Language (HDL). The HDL language chosen, was the VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL). Often, the VHDL models are partitioned into two groups. Behavioral models and structural models [12].

Behavioral models are at the most abstract level of VHDL models. Behavioral models describe the desired functionality of the system and they can include code structures that are not synthesizable. This implies that some of the code structures cannot be translated into hardware by a synthesis tool. Even though, the behavior of the system is correct in the simulation tools. [12]

Structural models describe accurately how the system is composed of other subsystems. Structural VHDL models describe the operation of the system in terms of data storage and transfer. A widely used structural model is the Register-Transfer Level (RTL) model. In the RTL model the operation of the system is often divided into a data-path and a control-path. In the data path, data is being transferred between storage elements (registers and/or memories) and only combinational logic is allowed between the storage elements. Thus, only the storage elements may store values in RTL models. The control-path controls and sequences how data is moved through the data-path. If the RTL model is written correctly in VHDL most synthesis tools are able to translate the VHDL code into a netlist. Thus, for RTL models the system's behavior is correct in the simulation tools as well as in the netlist translated from the RTL model by the synthesis tools. [12]

Sometimes, it is beneficial to describe a system in terms of both structural and behavioral models. These are called *mixed models*. In mixed models some aspects of the system are described as an RTL model and others as a behavioral model. [12]

3.1.1 Haar-like transformation VHDL mixed model

To get a clearer view how the algorithms 2.2.1 and 2.2.2 would be behave as hardware implementations and to investigate the implications of various design choices, VHDL mixed models for real-number Haar-like transforms of sizes $N = 8, 7, 6, 5, 4, 3, 2$ were created. As stated in section 3.1, in mixed models some aspects of the design can be modelled as in RTL models and other aspects as in behavioral models.

The behavioral modelling aspect was done using an IEEE VHDL Math Real Package [34] to implement the PE behavior with floating-point real numbers. Also, the behavior of the inverse square root block could be modelled by using the functions provided in the package and artificially inserting delay to the PEs to emulate the multicycle operation.

The VHDL models of the Haar-like transforms were composed of stages, PEs and permutations in between the stages as described in Chapter 2. The inverse perfect shuffle permutations between the stages were implemented as interconnects. The stages included output registers to which the output sub-vectors from the PEs were stored. The correct functionality of the VHDL model was verified against the Matlab model.

A possibility for hardware reuse was discovered when the architectures of the Haar-like transformations were explored. This method is applicable to subsequent pairs of transformation sizes with even ($N = 2k$) and odd ($N = 2k - 1$) numbers of inputs. As in, transformation pairs $N = 8$ & 7, $N = 6$ & 5 and $N = 4$ & 3 can each be implemented with the same hardware architecture. It is possible to implement a pair of Haar-like transformations with the same hardware when the input stage's first PE has the capability to simply exchange and bypass the elements of its input sub-vector to the output of the PE. The exchange-bypass capability is needed when the odd-input-transformation ($N = 2k - 1$) of the pair is being calculated. The overall architecture is of the same form as the even number transformation's regular form. This hardware reuse method is illustrated in figure 7.

For example, in the case of transformation pair $N = 8$ & 7 the invalid value (the value that is not part of the $N = 7$ transformation's input data vector) would be routed to the top element of the $N = 8$ input vector when implementing the $N = 7$ transformation. The first PE of the input stage exchanges and bypasses its sub-vector to the output in reverse order so that the invalid value is shuffled to the bottom half of the $N = 8$ transformation's output vector. When all the stages in the transformation have processed their input vectors, and the output vector is formed, an additional re-arranging could be done to the obtain the final output vector. After the re-arranging, the order of final output vector elements would correspond to the order of the elements in the input vector.

The possibility of hardware reuse gave rise to two types of PEs. One that has the basic operation of a spectral kernel and another that in addtion to the basic operation of a spectral kernel has the capability to define the elements of the spectral kernel according to the equation (24) for exchange-bypassing.

$$\begin{pmatrix} x_{osub_1} \\ x_{osub_0} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x_{isub_0} \\ x_{isub_1} \end{pmatrix}, \quad where \;\; u_{10} = 0 \;\; and \;\; v_{10} = 1 \tag{24}$$

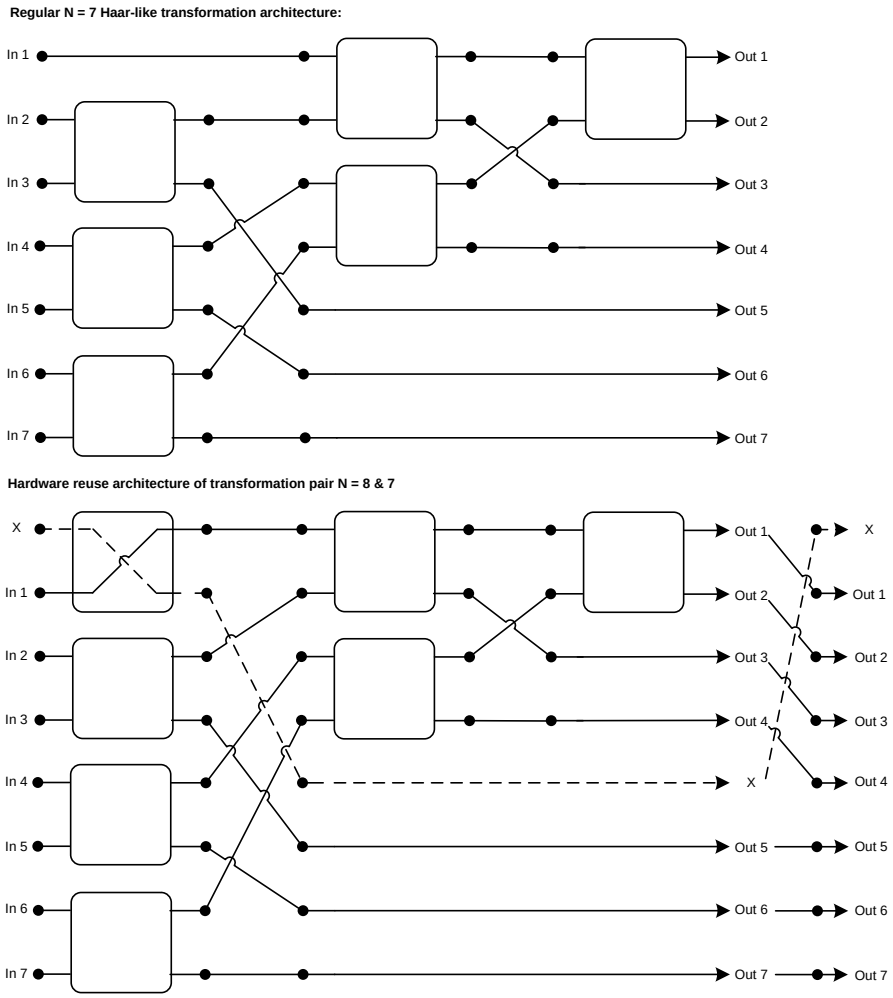

Figure 7: A schematic illustrating the hardware reuse method for Haar-like transformation pair $N = 8$ & 7. The regular architecture of $N = 7$ Haar-like transformation is shown on top and the hardware reuse architecture is shown at the bottom.

This capability was implemented with an additional control signal that, when active in generation mode, would force values of the spectral kernels to be defined according to equation (24) and then the spectral kernel values would be stored to the kernel registers inside the PE. In figure 8, the input stage of the $N = 8$ & 7 transformation pair is shown.

The PEs were designed to be controlled by a Finite State Machine (FSM). When the control signal from the FSM is set high the PEs operate in generation mode and when it is low the PEs operate in multiplication mode. The FSM control signal is drawn in green and the additional exchange-bypass control signal is drawn in blue. White PEs represent the type of PEs with only the basic operation and the first gray PE of the stage represents the type of PEs with also the exchange-bypass capability.

When the stages are connected together and a Haar-like transformation is configured, the FSM control signal could be routed to all the stages. Thus, the FSM

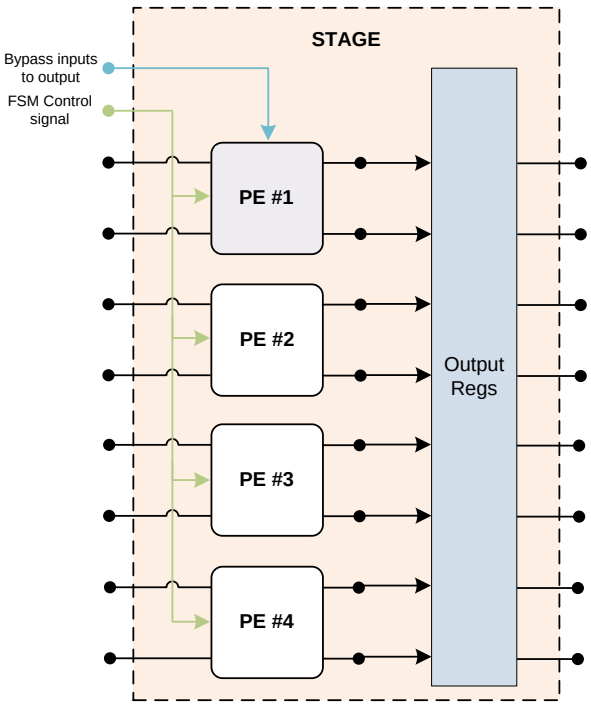

Figure 8: A schematic of an input stage for a $N = 8\&7$ Haar-like transformation pair with the exchange-bypass capability to enable hardware reuse.

would be able to control the generation and the multiplication modes of the whole transformation hardware architecture as described by the algorithms 2.2.1 and 2.2.2. Also, additional pipeline register stages were implemented to sychronize the data flow through the transformation. The *»Generation done»* signal is routed to the FSM to indicate that the generation mode is finished and to trigger a state change to multiplication mode. An example of the Haar-like transformation of the transformation pair $N = 8\ \&\ 7$ is presented in figure 9.

To test the functionality of the VHDL models a VHDL test bench was created. The test bench contained the FSM and additional control logic that facilitated the test procedure. Input test vectors were created from the Matlab model and they were saved into a separate input test vector file that was read from the test bench. The read values were passed to an instantiation of the Haar-like transformation VHDL top level entity.

Usually the top-level instantiation, instantiated in the test bench, is referred to as the Device Under Test (DUT). A schematic of the test bench and the DUT is presented in figure 10. All the control signals are not drawn separately for clarity of the figure. The control signals are represented simply by purple connections. The simulations to verify the functionality of the designs were done with Mentor Graphics' QuastaSim 10.3d simulation environment.

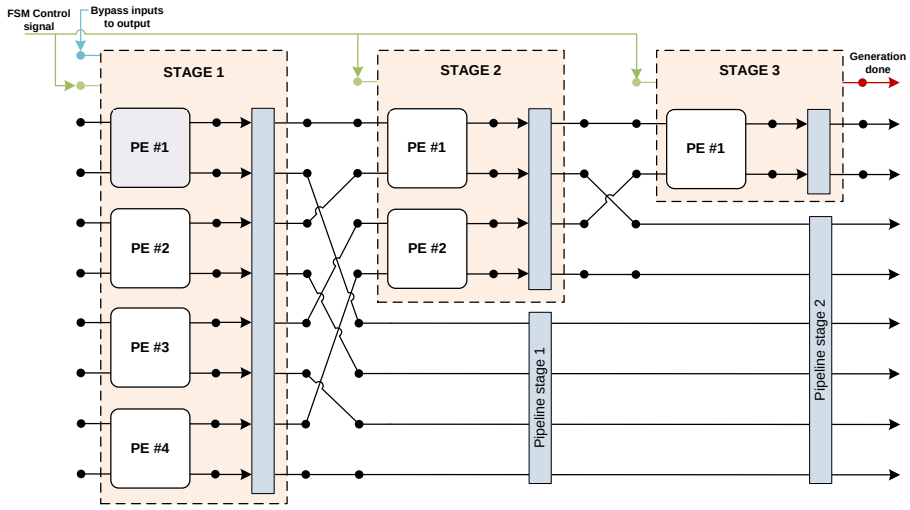

Figure 9: A schematic illustrating the hardware architecture implementing the transformation pair $N = 8$ & 7. Detailed routing of the signals inside the stages is omitted for clarity.

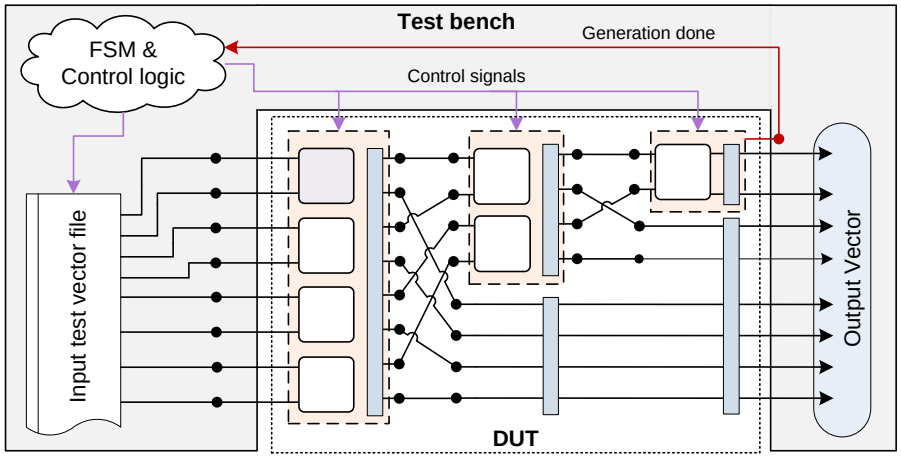

Figure 10: A schematic of the VHDL test bench setup. All the control signals are represented by purple connections and the red connection represents the *»Generation done»* signal from the last stage.

Controlling the generation mode of the whole hardware architecture by routing the FSM control signal to all the stages at once induces latency to the output when changing state from the generation mode to the multiplication mode. This would limit the throughput of the system. As discussed in section 2.3, the PEs will most likely have multicycle operation at least in the generation mode due to the inverse square root logic. To illustrate how the FSM is controlling the transformation and how it relates to the input and output data flow it is assumed that each stage can process its input vector in one clock cycle, even though this is not a realistic assumption

A timing diagram representing the simulation waveforms is presented in figure 11.

The simulation start pulse is initiated at *T0*. On the next rising edge of the clock, the FSM transitions to the generation mode setting the control signal to the PEs high (enabling the generation mode in the PEs) and the generating vector *V0* is read from the input test vector file.

After three clock cycles all the stages have processed the generating vector *V0* and the transformation output vector *O0* is available in the transformation output registers at time *T1*. At the same time the last stage sets the *»Generation done»* signal high which triggers a state change in the FSM and the control signal to the PEs is set low (enabling the multiplication mode in the PEs). The FSM also sets the *»Read input vectors»* signal high and starts reading the input vectors $V0, V1, V2, V3, \ldots$ from the file to apply the generated Haar-like transformaiton matrix to the input vectors. An input vector is read from the file at each rising edge of the clock.

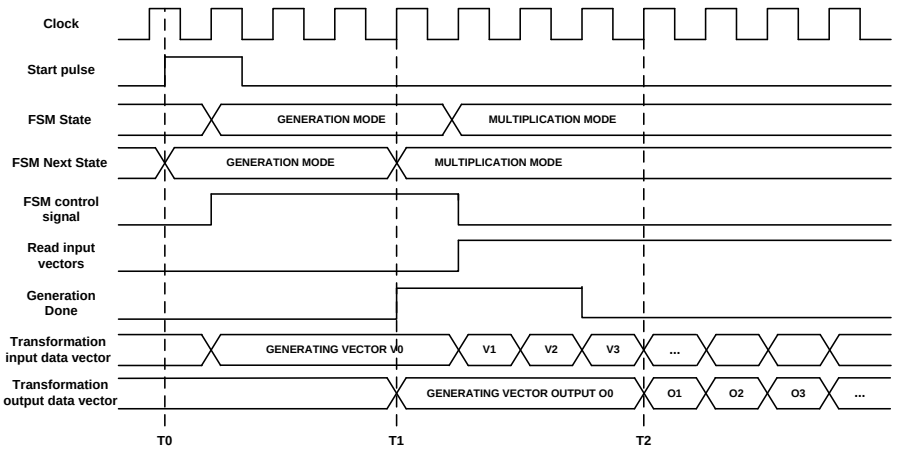

Figure 11: A Timing diagram illustrating the induced latency when the FSM control signal is routed to all stages at once.

The key point to consider here is that when the control signal is routed to all the stages at once the stages are not able to process input vectors during the generation mode as this would result into re-defining the spectral kernel elements according to the new input vector. Only after all the stages have processed the generating vector and the FSM changes state to multiplication phase, the vector *V1* can start to propagate through the transformation. Thus, latency (from *T1 + 1 Clock Cycle* to *T2*) is induced to the output of the transformation.

The assumption that each stage can process the input data vectors in one clock cycle is not realistic if high clock rates are desired. Depending on the implementation of the inverse square root logic, differences between processing times of the PEs inside a stage may vary. Also, if the exchange-bypass capability of the first stage is used, the PEs will have different processing times inside the stage. Moreover, the multiplication and addition operations in the PEs might also need multiple cycles and pipelining to achieve higher clock rates. For increased throughput, a more pipelined approach of controlling the generation and multiplication modes of the stages would be desirable. This would reduce the latency in the transformation when changing from generation phase to multiplication phase.

3.1.2 Pipelined Haar-like transformation VHDL mixed model

The multicycle operations in the PEs needed to be taken into account in the VHDL models. One way to deal with varying data rates is to introduce FIFO (First-In First-Out) buffer components to buffer the data flow where needed. By replacing the output registers with output FIFOs, it was possible to achieve synchronous data flow in the transformation hardware architecture. The last stage could be implemented with an output register as the need for an output FIFO would depend on the component the last stage output is connected to.

In figure 12 the first stage of the $N = 8$ Haar-like transformation hardware architecture is presented when the output register is replaced with an output FIFO. To ensure that the data vector stored in the output FIFO of is valid and all the PEs have finished their (possibly multicycle) operations additional control signals needed to be introduced to the design.

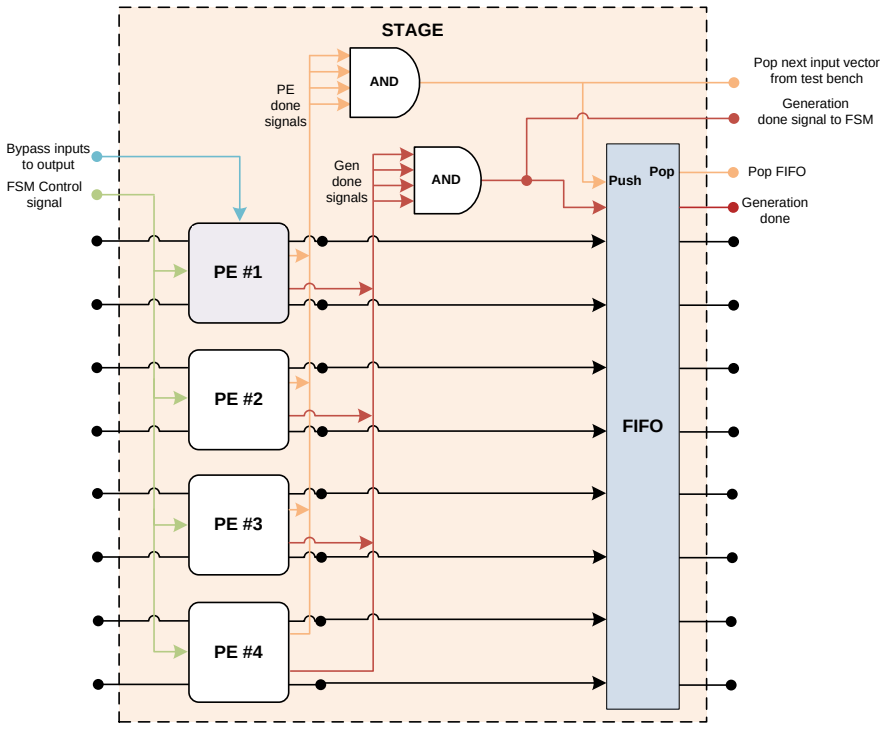

Figure 12: The input stage's architecture when the output register is replaced with an output FIFO.

First, a *»PE Done»* signal needed to be added to indicate that a given PE has finished its operation. The »PE done» signal is set high each time the processing element finishes its generation or multiplication operation. The individual *»PE Done»* signals originating from each PE were routed through an AND gate whose output controls the push port of the output FIFO. Thus, every time when all the PEs have finished with their processing the resulting output vector is pushed into the output FIFO.

The output *»PE Done»* from the AND gate is also routed to the previous stage's FIFO pop port to make the next input vector for the stage available. This control scheme enables each stage to push its output vector to the output FIFO and pop the next input vector from the previous stage's FIFO simultaneously.

In effort to achieve more pipelined behavior, each PE also outputs a separate *»Gen Done»* signal that is set high when the generation operation finishes and is set low when the multiplication mode is set for the PEs. Similarly, all the *»Gen Done»* signals are routed through an AND gate to the output FIFO and stored as the *»Geneneration Done»* output signal. The *»Geneneration Done»* signal would be then passed to the next stage's FSM control signal input port. The first stage passes its *»Geneneration Done»* signal also to the controlling top-level FSM to trigger a state change. Thus, the first stage can start processing the next input vector in multiplication mode while the second stage starts processing the generating vector and a more pipelined approach is achieved for the whole transformation hardware.

When the stages are connected together the following hardware architecture is formed. The *»PE Done»* signals are connected always to the previous stage's FIFO pop input port and the *»Geneneration Done»* signal is connected to the next stage's FSM control signal input port and the bottom half of the input vector elements are always connected directly to the to the output FIFO data inputs. The architecture is designed for FIFOs that are able to operate in »show ahead» [35] mode. Meaning, that the first input pushed into the FIFO is available at the output of the FIFO before a pop is signaled.

When the FIFO output is popped, the next value stored in the FIFO is made available to the FIFO output. The last stage in the transformation would have an output register instead of an output FIFO. The *»PE Done»* signal in the last stage would be connected to the output register enable to control when a valid output vector from the PEs is registered to the transformation output. The resulting hardware architecture is presented in figure 13.

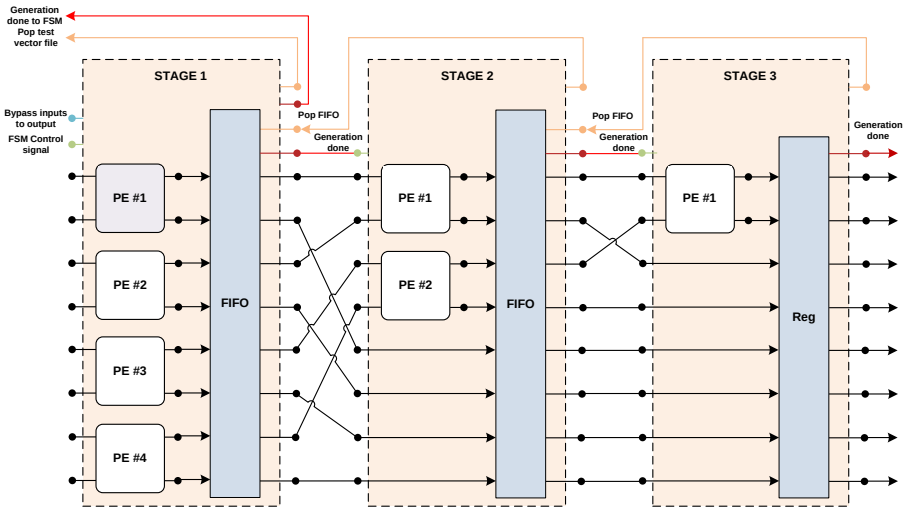

Figure 13: The Haar-like transformation hardware architecture implemented with output FIFOs

To test how the architecture would behave when the PEs have multicycle operations, delay was artificially induced to the PE VHDL models. The number of clock cycles it took for a PE to produce an output sub-vector in generation and multiplication mode could be adjusted with a purely behavioral delay logic inside the PEs. For example, a timing diagram of a simple case where a PE can produce an output sub-vector in four clock cycles in generation mode and in two clock cycles in multiplication mode is presented in figure 14.

Same as previously, the start pulse of the simulation starts the simulation at time T0 and on the next rising edge of the clock the first vector (the generating vector) is read from the file and the FSM changes state to generation phase. The FSM control signal to the first stage is set high enabling the generation mode in the PEs of the first stage. After four clock cycles the first stage has processed the generating vector and the *»Genereration Done»* signal is set high. The *»PE Done»* signal is also set high indicating that all PEs have finished their operations in the first stage. This controls the push signal to the stage's output FIFO and the *»Pop»* signal to the test bench input test vector file.

As the first stage's *»PE Done»* signal is fed back to the top-level FSM, a state change is triggered in the FSM. The FSM control signal is set low and the multiplication mode is enabled for the first stage PEs and the first stage starts to process the next input vector V1 in multiplication mode at time *T1*. At time *T1* also the generating vector *V0* becomes available at the first stage's FIFO output because it is operating in *»show ahead»* mode and the second stage starts to process its generating vector.

Finally, at time *T2* the third and final stage has processed the generating vector and the *»PE done»* signal of the third stage is raised high to indicate that all the PEs have finished their operations. This signal is also connected to the output register enable. Thus, the output vector is stored to the output register of the final stage and the whole transformation's output vector *O0* is available at the output. Simultaneously, the output FIFO of the second stage is popped and the next input vector *V1* is available to the third stage and the stage is set to work in multiplication mode on the next clock cycle from *T2*. The first and second stage have already been processing vectors in the multiplication mode independently and a more pipelined operation is achieved for the whole transformation architecture.

3.1.3 Processing Element VHDL mixed model

The structure of the PEs that was proposed in [5] was modified to fit the behavioral modeling of the whole system better. The overall functionality of the PEs was kept more in line with the original Matlab model than the proposed architecture and two output branches were added instead of one output branch. The PEs were mainly implemented as behavioral VHDL descriptions and therefore were non-synthesizable. For example, the inverse square root functionality was implemented using the square root function from the math package. A schematic of a PE with an exchange-bypass capability is presented in figure 15. The schematic shows an approximation of how the synthesis tool might implement the PE logic if the whole implementation was

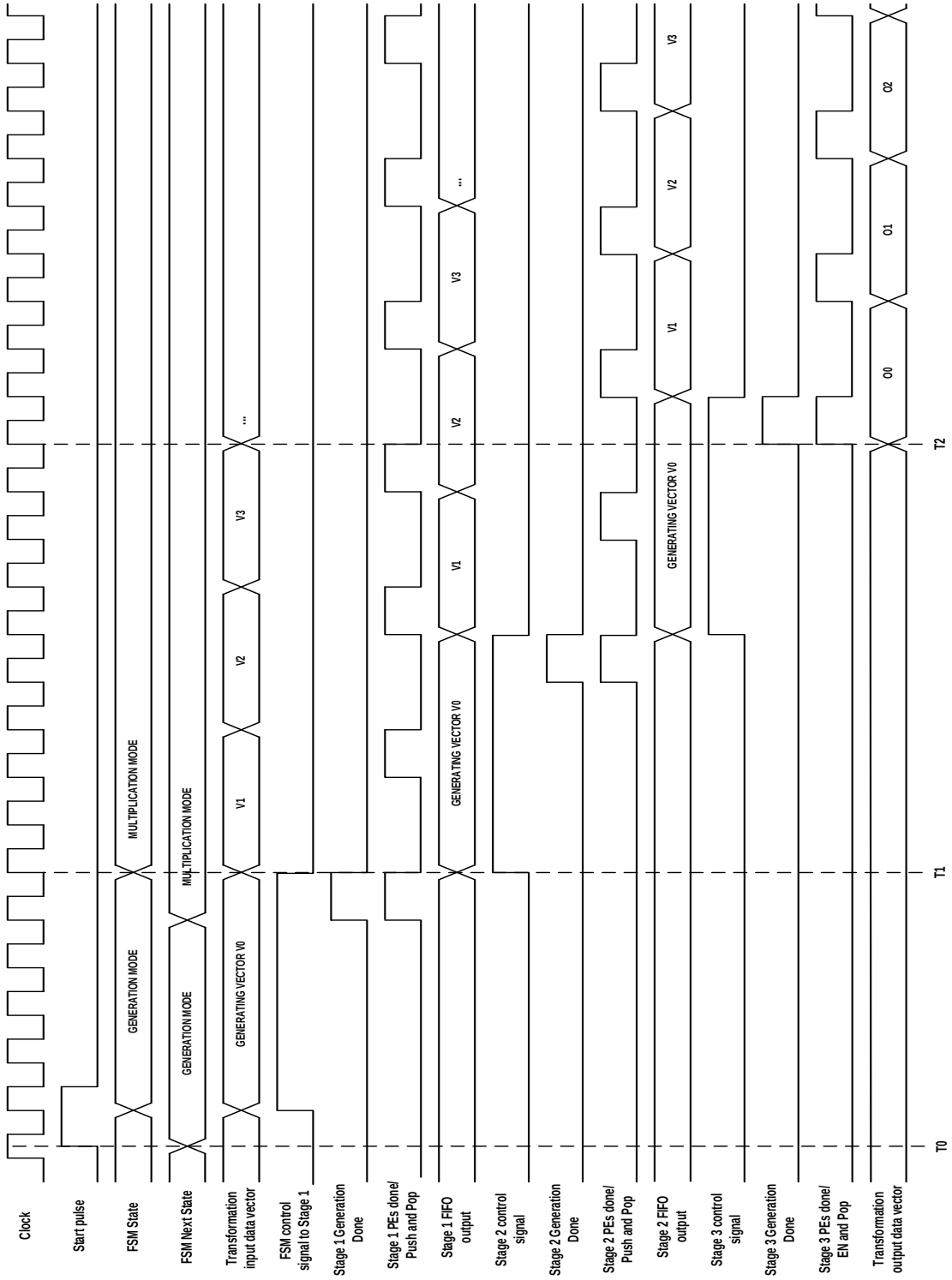

Figure 14: A timing diagram of the operation of the Haar-like transformation's FIFO implementation hardware architecture.

synthesizeable. It shows the overall functionality of the PE, although, some of the details are omitted for clarity. The non-synthesizeable delay logic is represented by a *»logic cloud»* in the figure.

The PEs were designed to be controlled by a control signal from a FSM. When the control signal is set high the PE is operating on generation mode and when the control signal is set low the PE is operating on multiplication mode. Each PE has two main computational logic blocks that are used to facilitate the PE operation so that it matches the operation of the spectral kernels. The two logic blocks are named *kernel generation logic* and *multiplication logic* in figure 15.

When the PE is operating in multiplication mode (control signal is low) the PE performs the row operations according to equations (22) and (23). The row operations are implemented by the multiplication logic with multiply-add and multiply-subtract operations with the PE inputs and the spectral kernel elements stored in the kernel registers.

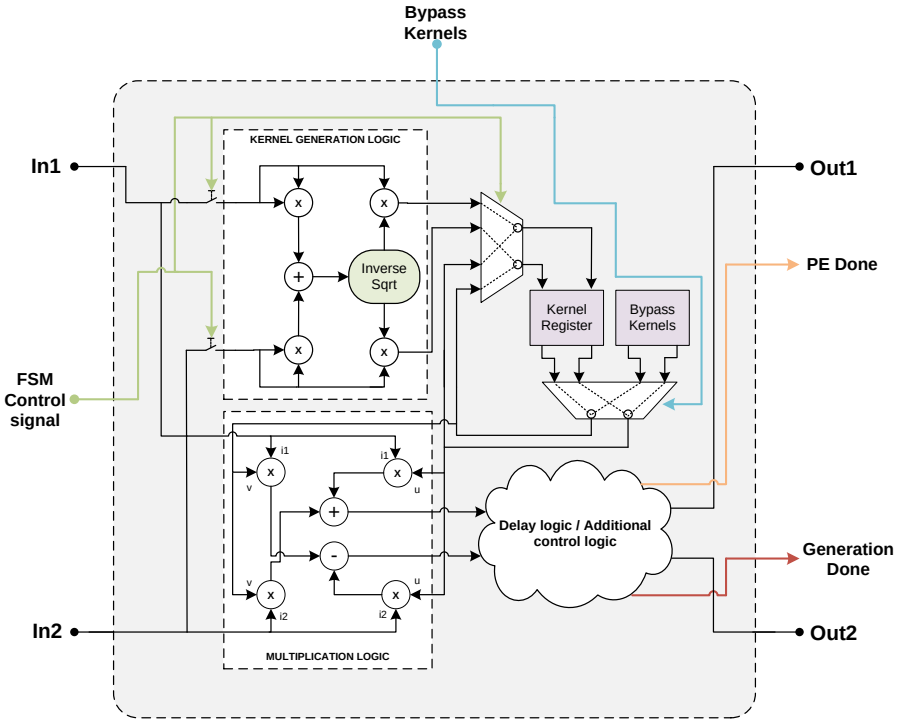

Figure 15: An approximation the PE architecture based on the VHDL mixed model.

The kernel generation logic calculates the inverse norm from the input values, multiplies the inputs with the inverse norm to form the spectral kernel elements and stores the elements into the kernel register. The control signal from the FSM enables the inputs to be routed to the kernel generation logic when the PE is operating in generation mode (control signal is high). The multiplication logic is also used when the PE is operating in generation mode to define the generating vector for the next stage. Thus, when the kernel elements are generated for the first time the butterfly operation is done to the input values (from which the kernels were generated) by the

multiplication logic. This defines the output sub-vector that is used to create the next stage generating vector.

To enable hardware reuse, as described in section 3.1.1, exchange-bypass capability is needed for the first PE of the input stage. The PE must be able to define the spectral kernel elements according to equation (24) to achieve this exchage-bypass capability. This can be implemented with a simple look-up table represented by the *»Bypass Kernels»* block in the figure 15. When the bypass control signal is set high the values for the look-up table are used to override the spectral kernel values stored in the kernel register.

To emulate multicycle operations of the inverse square root logic, and other possible multicycle operations inside the PE, additional delay logic was introduced to the PE. In VHDL it is possible to delay signal assignments using the *»after»* statement in signal assignments. This feature of the VHDL was used to define the number of clock cycles after which the outputs and other signals were made available to the PE output. The delay logic implemented is purely behavioral and not meant to be synthesized.

To reduce the number of multipliers and adders hardware reuse could be incorporated more to the PE design in a similar fashion as proposed in [5]. This would require more complex control logic to be implemented for the PE. For example, using more than two states for the top-level FSM or using a dedicated FSM inside the PE to control the operation. Also, if high clock speeds are desired it might become necessary to pipeline the kernel generation and multiplication logic blocks. The main findings from the mixed model designs are summarized in the next section and the next steps towards a fully synthesizable RTL model are discussed.

3.1.4 Findings and summary

The purpose of the investigations with the VHDL mixed model was to provide more insight to the implications and details of the hardware implementation of the algorithms 2.2.1 and 2.2.2. Moreover, by describing the behavioral parts of the mixed model with RTL descriptions the mixed model could be converted to a fully synthesizable RTL model in VHDL. During the exploration three main design questions were discovered that needed investigation.

First, the implementation of the inverse square root logic and its effects on the latency and the overall system behavior would need to be investigated. As discussed in section 2.3, the performance of the inverse square root logic reflects to the behavior of the whole system.

Second, an investigation for the accuracy of a fixed-point implementation would be needed. Furthermore, there might be a need to implement the Haar-transformations in floating-point arithmetic or if only fixed- point arithmetic was used, the bitwidth of the numbers might have to be very large. Perhaps, floating-point arithmetic would need to be only applied in some critical points of the systems. For example, the inverse square root calculation.

The third design question was, how to finally extend the real number implementation of the Haar-like transformations into a complex number implementation. The

design questions are summarized as

- How to implement the inverse square root logic and to study its effects on latency and overall system behavior?
- What kind of accuracies and bitwidths can be achieved with fixed-point arithmetic and what are the possibilities for floating-point arithmetic to increase accuracy?
- How to extend the real number implementation to a complex number implementation?

If hand coded VHDL was to be used to create the synthesizable RTL models and to do the design exploration, floating-point and fixed-point inverse square root logic blocks would need to be implemented. Implementing a hand coded VHDL inverse square root logic block would be far from trivial and using VHDL to describe floating-point arithmetic can be very time consuming. To do the design exploration in a timely manner, the synthesizeable RTL models were decided to be implemented with high-level synthesis tools. The implementation of the parametric Haar-like transformations' RTL models using high-level synthesis is discussed in more detail in the next section.

3.2 High-Level Synthesis Implementation

In High-Level Synthesis (HLS) RTL models are extracted from behavioral and functional descriptions by the HLS tools. Instead of describing the system with tradition HDLs, the behavior and functionality of the system is described with a higher-level programming language. Currently two main languages are used to make the behavioral and functional descriptions for HLS tools, namely the SystemC from Accellera and Algorithmic C from Mentor Graphics [36].

The HLS tool chosen was Mentor Graphics' Catapult HLS design environment version 10.2/754530 (beta release). Using Catapult HLS the implementation of the inverse square root logic would be simplified as the HLS tool is able to synthesize RTL code from the built-in square root and division functions. Also, a built-in inverse square root function included in the tool's libraries was investigated. The floating-point and complex number investigations could also, be done faster using the Algorithmic C data types provided by the tool.

With Catapult HLS the behavioral and functional hardware descriptions can be written in the ANSI C++ programming language with the Algorithmic C bit-accurate data types in use. However, when using C++ to describe the functionality of the system the designer must keep in mind that he is still using C++ to describe hardware *not software*. There are still many rules that have to be followed so that the tool understands what the designer wants to implement. Moreover, some C++ constructs may not be supported at all. [36] Full description of the Catapult HLS recommended coding style for C++ is outside the scope of this master's thesis. The reader is referred to the Catapult HLS Blue Book [36] and the Catapult Synthesis User and Reference Manual [37] for a more detailed description.

As discussed in Chapter 2, the parametric Haar-like transformations can be generically described by defining a set of parameters. This generic principle was also applied to the C++ models used to generate the RTL models for the Haar-like transformations' hardware architectures. A natural way to describe generic functionality in C++ is to utilize classes. Therefore, the design strategy was to utilize class based C++ design as much as possible.

It is crucial to have at least some idea how the Algorithmic C Datatypes can be used to model hardware with C++ descriptions. Thus, the Algorithmic C Datatypes are introduced in Section 3.2.1. In Section 3.2.2 the C++ description of the PEs are discussed and two methods to implement generic Haar-like transformation architectures using C++ in HLS are presented in Sections 3.2.3 and 3.2.4.

3.2.1 Algorithmic C Datatypes

Native C++ data types are not bit-accurate and flexible enough for hardware modeling. Catapult HLS provides a package called Algorithmic C (AC) Datatypes that allows bit-accurate modeling of arbitrary length fixed-point and floating-point numbers. The AC Datatypes package provides also a way to easily model complex numbers. The usage of different data types is enabled by including a header to the C++ file. The headers files define C++ templatized classes for different data types to model the bit-accurate hardware functionality. For example, when fixed-point numbers are used the *»ac_fixed.h»* file must be included to the C++ file. [38]

Fixed-point numbers are defined by the templatized class *ac_fixed<W,I,S,Q,O>*, where W is the width of the fixed-point number, I is the integer part width (or radix point placement) from the most significant bit (MSB). S is a boolean defining whether the number is signed (true) or unsigned (false). Fixed point numbers are defined as two's complement when signed numbers are used. Optional parameters Q and O define the quantization (rounding) and overflow methodology used. (In this thesis trunctaion (AC_TRN) and rounding (AC_RND) are always used). The numerical range for the fixed-point numbers are defined as from 0 to $(1 - 2^{-W}) \times 2^I$ for unsigned numbers and as $(-1/2) \times 2^I$ to $(1/2 - 2^{-W}) \times 2^I$ for signed numbers. The minimum increment for the fractional part is defined as 2^{I-W}. [38]

To illustrate how the fixed-point numbers are used, consider the fixed-point number definition ac_fixed<5,3,true> x. The integer part of x is defined to be 3-bits wide and signed numbers are used. Therefore, the numerical range for the integer part is from -4 to 3 when using two's complement. This leaves the fractional part with 2-bits to represent the range from 0 to 1. Therefore, the minimum increment is 1/4, which can also be calculated from the equation 2^{3-5}. Thus, the maximum range for the variable x is from -4 to 3.75. The numerical range can be confirmed form the equations $(-1/2) \times 2^3$ and $(1/2 - 2^{-5}) \times 2^3$.

Interestingly enough, a fixed-point number can be defined as ac_fixed<2,3,true> y. This would imply that the radix point is located beyond the width of the fixed-point number. The definition translates to a fixed-point number y that does not have a fractional part and when the equations are calculated for the number ranges the minimum increment is 2 and the numerical range is from -4 to 2. This means

that y is defined to cover the 3-bit two's complement numbers with a 2-bit number. Thus, the minimum increment is 2 and numbers -4,-2,0 and 2 are covered. These examples are illustrated in figure 16.

Floating-point numbers are defined by the templatized class *ac_float<W,I,E,Q>*. The significand of the floating-point number is defined as a signed fixed-point number ac_fixed<W,I,true> and the exponent is defined as an integer ac_int<E,true> (E denotes the width, true denotes the signedness). The optional parameter Q defines the rounding methodology. The numerical range is calculated from the equations $(-1/2) \times 2^{I+max\_exp}$ to $(1/2 - 2^{-W}) \times 2^{I+max\_exp}$, where $max\_exp = 2^{E-1} - 1$. The smallest increment is defined as $2^{I-W+min\_exp}$, where $min\_exp = -2^{E-1}$. For example, the IEEE 754 [39] single precision floating point number is defined as ac_float<25,1,8>. Notice that the significand is defined to be ac_fixed<25,1,true> which includes the 23-bit significand, the sign bit and the *»hidden bit»*. There is no *»implied 1»* in the ac_float. Similarly, the IEEE 754 double precision floating point number is defined as ac_float<54,1,11>. [38]

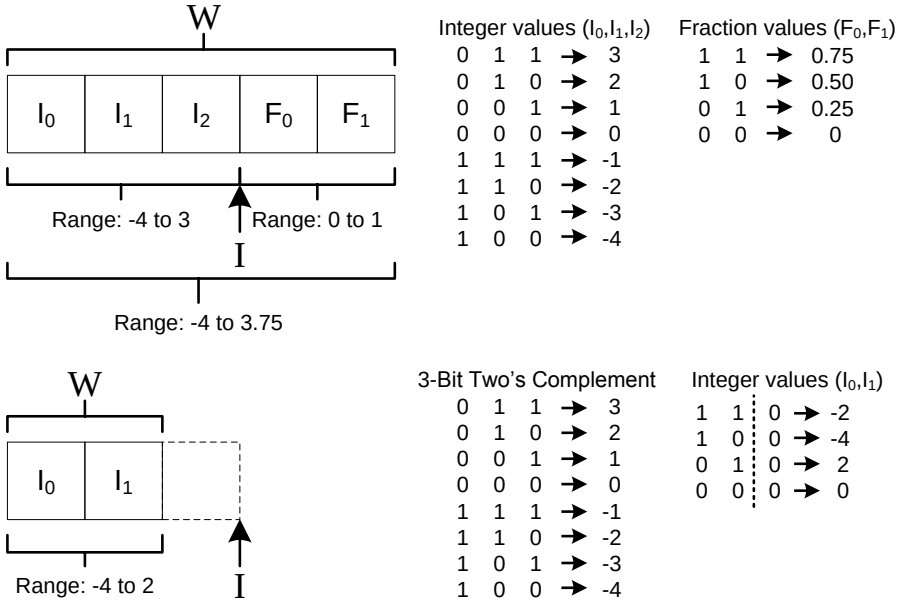

Figure 16: Examples of the AC fixed-point data type.

Complex numbers are defined by the templatized class *ac_complex<T>* where T is the type of the real and imaginary numbers. The type T parameter can be given as an AC float or integer type. Native C++ types *bool*, *signed and unsigned char*, *short*, *int*, *long*, *long long*, *float* and *double* are also valid types for the AC complex numbers. The numerical ranges for the real and imaginary parts are defined according to the set type. [38]

The Catapult HLS tool is used to extract an RLT model of a functional description written in C++. Functions that execute sequentially in the C++ code are converted to processes that can execute in parallel in the RTL model. To ease the RTL extraction procedure for hierarchical models and to ensure that the functionality

of the C++ code is translated correctly to the RTL model, a channel class can be used in the Catapult HLS. The *ac_channel<T>* class defines FIFO data transfer between the RTL model's processes that are generated from the C++ functions. The parameter T defines the data type that is passed through the channel. For example, ac_channel<ac_fixed(4,2,false)> defines a channel that passes unsigned fixed-point values through the channel. [38]

3.2.2 Processing Elements

A high-performance PE is the enabling factor for a high-performance Haar-like transformation and the inverse square root logic block lies at the heart of an efficient PE. As discussed in Chapter 2, it is of utmost importance to choose a well suited calculation method for the inverse square root logic. The Catapult HLS 10.2/754530 provides two methods for implementing the inverse square root. One method is to implement the inverse square root functionality with the standard AC square root and division functions and the other is a piecewise linear (PWL) inverse square root function included in the AC math libraries.

The PWL inverse square root function is a polynomial approximation method that can process fixed- and floating-point inputs. The square root function on the other hand is implemented as an iterative calculation and it can only have fixed-point values as an input. For fixed-point PE implementation the results suggest that the PWL inverse square root function uses less resources than the square root and division functions but it also produces more inaccurate outputs than the square root and division functions combined. The results are discussed in section 4.1.

The accuracy constraints for the PEs depend largely on how the Haar-like transformation is used. If information about the expected input range is available this information can be taken advantage of in the design process. For example, if it is known that the transformations will process input vectors whose elements' magnitudes will vary only from 0 to 1, the PEs can be designed to accommodate this range of inputs with a given level of accuracy while minimizing the bitwidths. For the investigation done in this master's thesis, the input test vectors were restricted to have only integer values between 1 and 32.

The fixed-point implementation of the PEs was done similarly as in the VHDL models. Conceptually the same architecture can be considered as in the figure 15, although without the delay logic. An example code of the C++ implementation of the real-number fixed-point PEs is presented below. The name of the type definitions for each operation output type are included in the example code comments.

Each output type for the operations is defined to be an AC fixed-point number. By adjusting the lengths of the integer and fractional parts of the output types the accuray-level for the whole PE could be defined. Also, defining the output types as AC complex types easy migration to a complex number implementation could be achieved. The bitwidths used for each output type in every transformation size are presented in appendix B.

```
void PE(
        input_type in1,
        input_type in2,
        int K_ind1,
        int K_ind2,
        ctrl_type gen,
        ctrl_type sel,
        output_type &out1,
        output_type &out2
       ){
    if (gen == 1) { // Kernel generation logic
        if (!( (in1 == 0) && (in2 == 0) )){
            sqr1 = (in1*in1);        // sqr1 = sqr_type
            sqr2 = (in2*in2);        // sqr2 = sqr_type
            Temp = sqr1 + sqr2;      // Temp = sum_type
            sqrt(Temp,sqroot);       // sqroot = sqroot_type
            div(One,sqroot,invNorm); // invNorm = norm_type
        }
        if((in1 == 0) && (in2 == 0)){ // If norm is zero identity kernels
            K[K_ind1] = 1; //K = kernel_type
            K[K_ind2] = 1; //K = kernel_type }
        else if (sel == 1){ // Generate bypass kernels
            K[K_ind1] = 0; //K = kernel_type
            K[K_ind2] = 1; //K = kernel_type }
        else { // Generate kernels
            K[K_ind1] = in1*invNorm; //K = kernel_type
            K[K_ind2] = in2*invNorm; //K = kernel_type
        }
    }
    // Multiplication logic
    out1 = (K[K_ind1]*in1) + (K[K_ind2]*in2);
    out2 = (K[K_ind2]*in1) - (K[K_ind1]*in2);
}
```

In the PEs the most complicated calculations are done in the kernel generation logic. Floating-point Matlab reference calculations were generated for the operations in the kernel generation logic and the reference output values were compared with the output values of the fixed-point operations. The accuracy of the C++ fixed-point values was set by first determining the needed integer part widths and then adjusting the bitwidth of the fractional part to achieve the wanted accuracy-level (See figure 16). The accuracy and bitwidth optimizations results are discussed in section 4.1.1 for the real number and in section 4.1.2 for the complex number implementations.

The Catapult HLS tool is able to unroll loops to parallelize the loop iterations. This is done either by using a pragma or setting the unrolling option from the tool GUI. To investigate latency reductions, the iterative square root and division

functions were fully unrolled. When using a single PE, the Catapult HLS tool was unable to introduce parallelism by unrolling but when multiple PEs are using the functions the tool is able to reduce latency of the transformations as a whole. The results are discussed in Chapter 4. Generally, with Catapult HLS the fixed-point implementations can achieve wanted accuracies with fewer hardware resources than floating-point implementations. Although, it might come into question to implement the PE calculations, or parts of them, using floating-point arithmetic.

In Catapult HLS 10.2/754530 the PWL inverse square root function can be directly used with the AC floating-point data types. However, no considerable improvement could be achieved to resource usage, clock rate or accuracy with the floating-point PWL function. Another possibility to increase the PE accuracy would be to convert the fixed-point output value from the square root function to floating-point and do the division in floating-point. As there is no standard way to do this conversion in Catapult HLS, the conversion would need to be implemented with additional logic. Due to limitations for the time frame of this thesis this method was not investigated further. The floating-point PEs are discussed in more detail in section 4.1.3.

3.2.3 Class Based Hierarchy

To create a generic description of the parametric Haar-like transformations, templatized C++ classes were used to implement a class based hierarchy. One of the main reasons to use hierarchical C++ designs in Catapult HLS is to enable pipelining between loops by describing the C++ loops in different hierarchical blocks [37].

In Catapult HLS it is possible to implement a hierarchical design using C++ classes and instantiations of those classes. The *»bottom-up»* hierarchical design method was found to be suitable for the Haar-like transformations. In the bottom-up method instantiations of lower level sub-classes are instantiated in top-level classes to form a hierarchy. [37]

In Catapult HLS C++ is used to describe hardware and but the tool needs to understand what is being described. Thus, design rules need to be followed with the C++ coding style when implementing a class based hierarchy. Some of the main rules that must be followed when designing a class based hierarchy include such rules as

- Only one public member-function is allowed per class and it must use *hls_design interface* pragma
- All data to the public member function must be passed through the defined interface
- To form hierarchies, hierarchical blocks should be connected with channels

By adhering to the rules presented in [37], a bottom-up class based hierarchy was implemented.

To investigate the parametric Haar-like transformations on a broad range, transformations of sizes $N = 32, 26, 16, 8, 7, 6, 5, 4, 3, 2$ are considerd. As discussed in

section 3.1.1 transformation pairs 7 & 8,5 & 6 and 3 & 4 could be implemented with the same hardware architecture. Thus, in total of seven C++ hardware implementations were made. The implementatios were composed of a top-level instantiation and multiple generic stage sub-class instantiations. Both of the classes were templatized with parameters to form a generic Haar-like transformation that could be instantiated with a wanted size and structure by defining the template parameters.

The stage sub-class structure has two main components. One main public function named *»run»* that defines the public interface and a private function named *»PE»*. The private *»PE»* function implements the processing element functionality as described in section 3.2.2. The *»PE»* function must be defined as an inline function to be compatible with the hierarchical design. This means that the function calls will be replaced with the logic inside the function by the HLS tool. In the main public function *»run»* the *»PE»* function is called within a loop that is fully unrolled with the *hls_unroll* pragma to generate parallel logic for the PEs. Since the *»PE»* function is defined as an inline function, the parallel calls from the loop are not generated as separate hierarchical PE blocks. Instead, each unrolled parallel function call generates non-hierarchical combinational logic inside the hierarchical stage instantiation.

An alternative, and possibly a better, way would be to implement the processing element functionality as another PE sub-class. The PEs would be instantiated inside the stage sub-class in a bottom-up manner as an array of instances and the size of the array of PE instances would be controlled by a template parameter. This way, the PEs would be instantiated as hierarchical blocks inside the stage sub-class and not as *»flat»* combinational logic. This method would require also an array of channel instances that connect the array of PE instances correctly to other logic inside the stage. In Catapult HLS 10.2/754530 the support for arrays of instances was not mature enough for this method to be utilized. Although, in future version this method might become possible to utilize.

The data type transferred through a channel between stages is a data vector, as in a C++ array. If arrays are passed through channels the arrays must be defined as a C++ struct. The data vector structs contained an array for the data elements and an array for the control elements. The control element array contained values to control the mode of the PE and the exchange-bypass operation the same way as discussed in section 3.1. For example, if the generation variable in the control array is set to 1 the stage enables the kernel generation logic in the PEs.

The number of processing elements generated inside a hierarchical stage block is controlled by the number of iterations in the loop where the *»PE»* function is called. The number of iterations in the loop is in turn controlled by a template parameter. According to the Catapult HLS design rules for the C++ coding style, data should be transferred to the class instances through the one defined interface. An example C++ code illustrating the stage sub-class structure is presented below.

```
template<typename ChType, int N_PEs, int N_inputs, bool Skip>
class Stage_class{
 private:
   // Private data arrays
   input_type  Stage_in[N_inputs];
   output_type Stage_out[N_inputs];
   output_type Reg[N_inputs];
   kernel_type Kernel_array[2*N_PEs];

   // Private PE function
   #pragma hls_design inline
   void PE(/*Arguments omitted*/){
       // PE Functionality omitted for clarity
   }
   // Other private data members omitted for clarity

 public:
   // Public Constructor
   Stage_class(){
      // Initializations omitted for clarity
   };

   // Main public function
   #pragma hls_design interface
   void CCS_BLOCK(run)(
                       ac_channel<ChType > &StageIn_ch,
                       ac_channel<ChType > &StageOut_ch
                      ){

    // Channel read logic omitted for clarity

         // PE loop generates PE logic
         // Iterations according to template parameter N_PEs
         #pragma hls_unroll yes
         PE_loop : for (int i = 0; i < N_PEs; ++i){

                 // PE function call
                 PE(/*Arguments omitted*/);

             // Indexing omitted for clarity
         }

    // Shuffle and channel write logic omitted for clarity
 }
};
```

In figure 17 a conceptual block diagram is presented where the stage operation is illustrated. The operation is as follows. When arrays are passed through a channel, the arrays are first read to a temporary array and the data elements from the temporary array are passed inside a *»Read loop»* to a *»Stage in»* array. The *»PE»* function calls are done in a *»PE loop»* that loops as many times as the template variable dictates. The kernel elements generated in the generation phase for all the PEs must but stored to a private array for persistent data storage between function calls. When the kernel array is implemented as a private data member of the stage sub-class the data stored in the array is persistent as long as the stage instance exists.

Inside the *»PE»* function calls the two-element output sub-vectors are stored to a private *»Reg»* array in the stage class. After the *»PE loop»* has finished, the data elements form the *»Reg»* array are shuffled inside a *»Shuffle loop»* and stored to another private array *»Stage out»*. After the shuffle the data elements are read from the private *»Stage out»* array to a temporary array and the temporary array is written to the output channel.

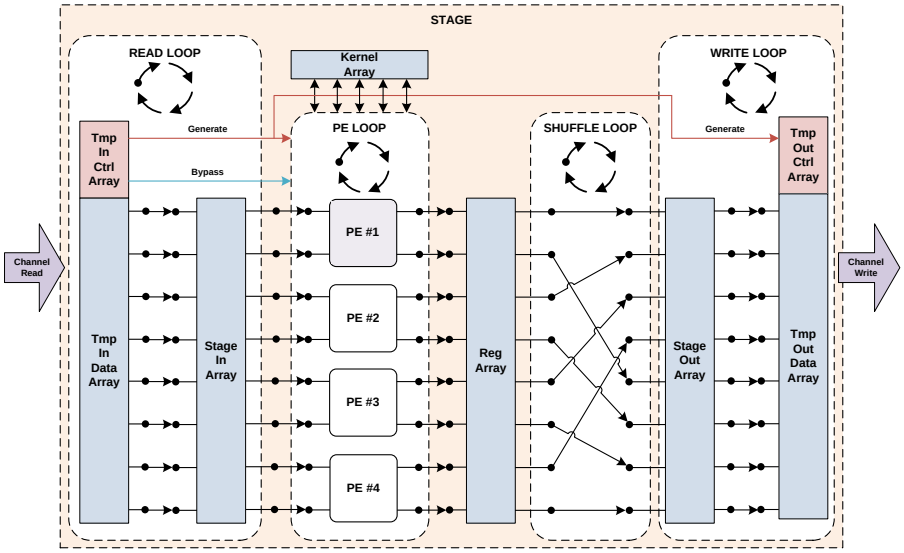

Figure 17: Conceptual block diagram of the stage operation. All loops are unrolled.

For the biggest transformation size $N = 32$, five stages are needed to form the upper triangular Haar-like transformation flow graph. Thus, all the transformations would need anywhere between 5 to 1 stages to be implemented. Therefore, five generic top-level classes that could implement a 5, 4, 3, 2 and 1 stage transformations were created. To form hierarchies, the stage instances should be connected with AC channels. The detailed implementation of the transformation architecture is parametrized and controlled with template parameters. An example C++ code of a five-stage-transformation top- level class is presented below.

The stages of the transformations, and the channels connecting them, were instantiated as private instantiations inside the top-level transformation class to form a bottom-up hierarchical design. The exchange-bypass capability discussed in section 3.1.1 is needed in the first PE of the first stage of the transformation.

Thus, two types of stage sub-classes were created. The regular *»Stage_class»* and the *»Stage_class_ex»* with the exchange-bypass capability.

```
template<typename ChType, int N1, int N2, bool S2, int N3, bool S3,
                     int N4, bool S4, int N5, bool S5, int N_inputs>
class TF_Stage5{
 private:
   // Private channels to connect stages
   ac_channel<ChType> Top_12_ch;
   ac_channel<ChType> Top_23_ch;
   ac_channel<ChType> Top_34_ch;
   ac_channel<ChType> Top_45_ch;

   // First stage has bypass capability => No "S1" needed
   Stage_class_ex<ChType, N1, N_inputs> Stage1;

   // Other stages
   Stage_class<ChType, N2,  N_inputs, S2> Stage2;
   Stage_class<ChType, N3,  N_inputs, S3> Stage3;
   Stage_class<ChType, N4,  N_inputs, S4> Stage4;
   Stage_class<ChType, N5,  N_inputs, S5> Stage5;

 public:
   // Public constructor
   TF_Stage5(){}

   #pragma hls_design interface
   void CCS_BLOCK(run)(
                       ac_channel<ChType >  &TF_In_ch,
                       ac_channel<ChType >  &TF_Out_ch
                      ){
          // Connect the stages using the private channels
          Stage1.run(TF_In_ch,Top_12_ch);
          Stage2.run(Top_12_ch,Top_23_ch);
          Stage3.run(Top_23_ch,Top_34_ch);
          Stage4.run(Top_34_ch,Top_45_ch);
          Stage5.run(Top_45_ch,TF_Out_ch);
   }
};
```

The N parameters in the template represent the number of PEs in each corresponding stage. The S parameters represent a boolean value that enables the stage to skip the first data element in its input data vector and pass straight to the *»Stage out»* array. The *N_inputs* parameter denotes the number of inputs to the transformation and it is used to initialize indices in the lower level stage sub-class instances. The *ChType* parameter defines the vector type that is being passed through

the channels. For example, the transformation $N = 6$ presented in figure 18 would be defined in a two-stage top-level class instantiation with the template parameters as *<vector6,3,1,true,1,false,6>*, where the *vector6* template parameter is the struct type passed through the channels.

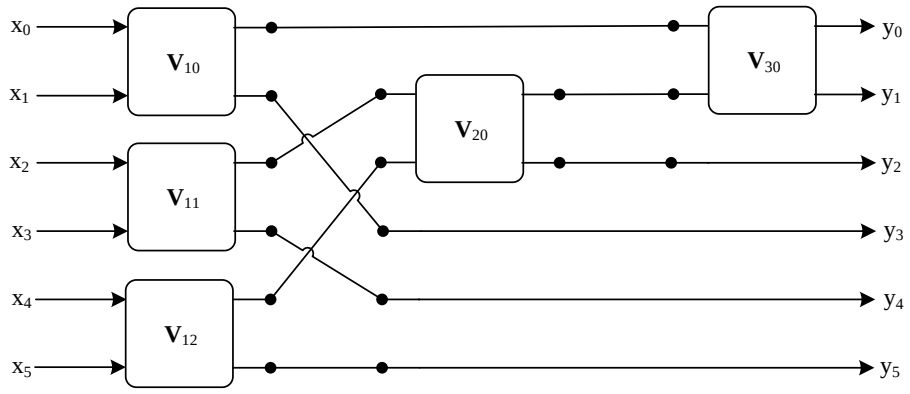

Figure 18: A flow graph of $N = 6$ Haar-like transformation

As the transformations were implemented with the square root and division functions, four-decimal accuracy levels were maintained for 80% of the transformation output values while the bitwidths were minimized. The accuracies are summarized in section 4.1 (see tables 8 and 9). The bitwidths used are presented in appendix B.

Further possibilities for resource optimizations were investigated by mapping the kernel array to different memory elements. No conclusive evidence was obtained to which memory element the kenrel array should be mapped to minimize the overall resource usage for all the transformation sizes. The results are discussed in section 4.2.1 for the real and in section 4.2.2 for the complex number implementations.

In effort to reduce the transformation latency, loop unrolling of the iterative division and square root functions was investigated. The main result was that latency could be significantly reduced by unrolling the functions but this comes at a high cost in terms of increased resource usage and reduced clock rates. In section 4.2.1 and 4.2.2 the results are discussed in more detail.

Since the top-level classes needed to be classified and implemented according to the number of stages in the transformation, the proposed method is not fully compliant with the generic principle of parametric Haar-like transformations. For a more generic description using the bottom-up class based hierarchy, support for generically connecting arrays of channel instances would be needed. Therefore, investigation on a more generic transformaiton type was performed. To implement a more generic transformation all the hierarchies needed to be flattened. Thus, a *»flat hierarchy»* was conceived. The implementation of parametric Haar-like transformations using a flat hierarchy is discussed in the next section.

3.2.4 Flat Hierarchy

The flat hierarchy design consisted of a single templatized top-level transformation class. An example C++ code illustrating the flat hierarchy structure is presented below.

```
template<ChType, int N_inputs>
class TF_class{
 private:
   // Index and other private member variables omitted for clarity
   input_type Stage_in[N_inputs];
   output_type Stage_out[N_inputs];
   output_type Reg[N_inputs];
   kernel_type Kernel_array[2*(N_inputs-1)];
 public:
   TF_class(){
     // Initializations omitted for clarity
   }
   #pragma hls_design interface
   void CCS_BLOCK(run)(
                       ctrl_type gen,
                       ctrl_type inv,
                       ac_channel<ChType > &StageIn_ch,
                       ac_channel<ChType > &StageOut_ch
                      ) {
   // Channel read and write logic are omitted for clarity

       #pragma hls_unroll yes
       STAGE_LOOP: for (int i_stages = 0; i_stages < N_stages;
                         ++i_stages) {

            #pragma hls_unroll yes
            PE_LOOP: for (int i = 0; i < PE_loop_cond; ++i){
              // PE Logic omitted for clarity
            }
            #pragma hls_unroll yes
            SHUFFLE_LOOP: for (int i = 0; i < N_inputs; i +=2){
              // Shuffle Logic omitted for clarity
            }
       }
};
```

The main idea behind the flat hierarchy was to implement the Haar-like transformation computation in the main public member function with three loops. A *»Stage loop»*, a *»PE loop»* and a *»Shuffle loop»*. The *»Stage loop»* would iterate through the number of stages needed in the transformation and the *»PE loop»* would iterate through the

number of PEs inside the stage that is currently being iterated in the *»Stage loop»*. Everytime the *»PE loop»* finishes, inverse perfect shuffle is performed to the output by the *»Shuffle loop»*. After the *»Stage out»* array is formed,the values are passed back to the *»Stage in»* array for another iteration of the *»Stage loop»*. All the loops are again full unrolled to enable parallelism. A conceptual schematic of the overall architecture of the flat hierarchy structure is presented in figure 19.

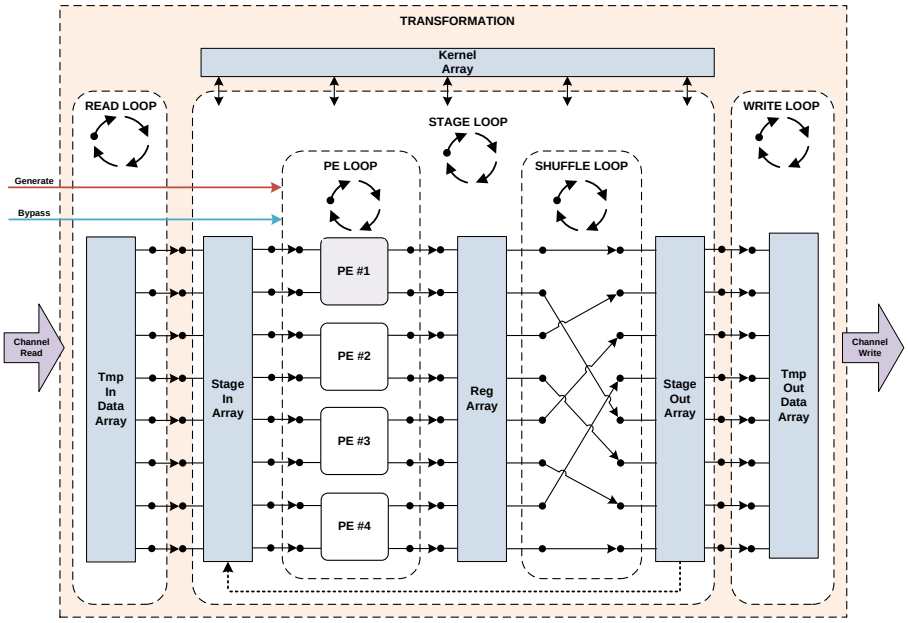

Figure 19: A conceptual image of the transformation operation

In the flat hierarchy only information about the channel type and the number of inputs to the transformation need to be set. The total number of PEs utilized by the transformation is needed to size the *»Kernel array»* correctly and to set the proper terminating conditions to the *»PE loop»*. It can be shown that the total number of PEs in a given transformation is one less than its number of inputs. As in, $N_{PE_{tot}} = (N_{inputs} - 1)$. The number of stages is calculated from the equation $N_{Stages} = \lceil \log_2(N_{inputs}) \rceil$. Thus, the number of stages can be derived from the *N_inputs* parameter by a *LOG2_CEIL »helper class»* that is provided by the Catapult HLS libraries.

Complex and real number implementations were created for the flat hierarchy designs. The same investigations were repeated for the flat hierarchies as for the class based hierarchies. Generally, the flat hierarchy implementaitons used less resources than the class based implementations but they also had lower clock rates. The results are discussed in section 4.3.1 for the real and in section 4.3.2 for the complex number implementations.

4 Design Optimization and Results

In this chapter, the design optimizations and synthesis results for the parametric Haar-like transformation hardware architectures are presented. The synthesizable RTL models were extracted with Catapult HLS 10.2/75453 and then the models were synthesized with Quartus Prime Pro 18.0 IR2. The FPGA device chosen for synthesis was Intel's (formerly Altera) Stratix 10 SX series 1SX280LN3F43I1VG with a speed grade 1 (fastest) and the target clock rate was set to 350 MHz.

The Stratix 10 devices have LUT based Adaptive Logic Modules (ALMs) that are used to implement most of the logic functions in the FPGA. Also, DSP blocks may be used to achieve high precision fixed and floating-point (IEEE single precision [39]) multiply-accumulate arithmetic. The Stratix 10 SX series include large embedded memory block, M20Ks, that are 20 Kbits in total size. Also, smaller LUT based RAM blocks that are 640 bits in total size, called Memory Logic Array Blocks (MLABs), may be employed. The FPGA included more MLABs than M20Ks and the MLABs are distributed with a finer granularity.

The resource usage for each transformation size was represented by the total number of ALMs and registers used in the device. Possibilities of utilizing the M20K and MLAB memory elements to reduce the ALM and register usage was investigated for both class based and flat hierarchies. The real number fixed-point implementations were extended to complex number implementations by changin the data types to AC complex. The complex AC data type enabled easy migration from a real number implementation to a complex number implementation.

As discussed in section 3.1.1, subsequent transformation pairs of sizes $N = 8$ & 7, $N = 6$ & 5 and $N = 4$ & 3 could be implemented with the same hardware architecture. Thus, transformation sizes $N = 32, 26, 16, 8$ & $7, 6$ & $5, 4$ & $3, 2$ are considered in this chapter. The functionality of the designs was verified with a C++ testbench and with the Catapult HLS SC Verify verification environment using QuestaSim 10.3d. Each design was verified using an integer generating vector in the generation phase and then with a second integer input vector in the multiplication phase. For example, the N = 8 transformation the generating vector was $V0 = [1, 2, 3, 4, 5, 6, 7, 8]^T$ and the input vector in the multiplication phase was $V1 = [3, 8, 2, 5, 7, 1, 4, 6]^T$. The resulting output vectors were compared to the reference Matlab model.

The chapter is organized as follows. First, in Section 4.1 the PE implementation methods are presented and the implementation of the square root logic and its effects are discussed. Also, the floating-point possibilites are covered. In Sections 4.2 and 4.3 fixed-point class based and flat hierarchies are covered. Both hierarchies were also extended to complex number implementations using the complex AC data types. Finally, answers to the design questions presented in Section 3.1.4 are formulated and a summary of the results is given in Section 4.4.

4.1 Processing Elements

In this section the results for the PE implementations described in Section 3.2.2 are discussed. The fixed-point real number implementation is discussed in Section 4.1.1 and the fixed-point complex number implementation is presented in Section 4.1.2. Finally, the possibilites for floating-point arithmetic in the PEs are covered in Section 4.1.3.

4.1.1 Fixed-point Real Number Implementation

The PE functionality was implemented as a private function in the class based hierarchy and the same functionality was used in the flat hierarchy. The private *»PE»* function was synthesized separately to investigate the differences between the PE implementations utilizing the PWL inverse square root function and the square root and division functions.

The PE using square root and division functions was optimized with respect to the bitwidths. The same bitwidths were used for the PE implementation with the PWL inverse square root function to compare the achieved accuracy level. The synthesis results are presented in table 1.

Table 1: The resource usage and maximum clock rate for the fixed-point real number PE implementations.

Resource	**PE, Sqrt&Div**	**PE, Inverse Sqrt**
ALMs	536	497
DSPs	1	1
Tot. Registers	485	714
Max. Clock Rate	424.63 MHz	336.81 MHz

The results suggest that the PE implemented with the PWL inverse square root function uses less ALMs but it may use more registers in total. Also, for the fixed-point real number implementations the square root and division function allowed a higher clock rate. Both implementation used only one fixed-point DSP block when synthesized. This might imply that Quartus is not able to fully infer DSP block usage from the RTL model extracted from the C++ description. Improving DSP block inference would need more in-depth investigation and was excluded from the scope of this master's thesis.

The PE implemented with the square root and division functions was optimized with respect to the bitwidths to give an accuracy level of at least four decimals when compared to the Matlab floating-point reference. The same bitwidths were used for the PE implemented with the PWL inverse square root function to compare the achieved accuracy. The output value comparisons for the operations used in the PEs are presented in appendix C table C1. From table C1 it can be seen that the PWL inverse square root function is able to produce output values that are only accurate

up to two decimal places with the same bitwidths as the square root and division functions.

Catapult HLS provides latency calculations automatically for the generated RTL. As the square root and division functions were implemented as iterative functions, the total latency for the PE was higher. The latency for the PE implemented with the square root and division functions was 71 clock cycles whereas the PE implemented with the PWL inverse square root function had a latency of 38 cycles. The latency results are presented in table 2.

Table 2: The latency results for the fixed-point real number PE implementations.

	PE, Sqrt&Div	**PE, Inverse Sqrt**
PE Latency	71 Cycles	38 Cycles
Square root Latency	15 Cycles	N/A
Division latency	28 Cycles	N/A

In effort to reduce the PE total latency the square root and division function were fully unrolled to introduce parallelism to the loop iterations. The unrolling of the division and square root functions provided worse results than the fully rolled versions. The results are shown in table 3.

Table 3: The resource usage, maximum clock rate and latency results for the unrolled fixed-point square root and division real number PE implementation.

Resource	**PE, Sqrt&Div Unrolled**
ALMs	1195
DSPs	1
Tot. Registers	853
Max. Clock Rate	287.03 MHz
PE Latency	80 Cycles

The increased latency can be explained by the fact that each division and square root iteration needs the result from the previous iteration. Thus, dependencies between iterations prevent further parallelization of the iterations. When the functions are fully unrolled the iterations form a sequential chain of logic and resource usage is increased and new critical paths are introduced that reduce the maximum clock rate. As will be shown in sections 4.2 and 4.3 when multiple PEs are used in a stage and the square root and division functions are unrolled the Catapult HLS tool is able to improve parallelism of the logic and reduce the latency for the transformations as a whole.

4.1.2 Fixed-point Complex Number Implementation

The fixed-point real number PE investigations were repeated for the complex number implementations. Again, the bitwidths were optimized for the PE version that used the square root and division functions and the same bitwidths were used in the PE that used the PWL inverse square root function for comparison. The synthesis results are presented in table 4.

Table 4: The resource usage and maximum clock rate for the fixed-point complex number PE implementations.

Resource	**PE, Sqrt&Div**	**PE, Inverse Sqrt**
ALMs	1408	1355
DSPs	1	1
Tot. Registers	1946	1795
Max. Clock Rate	342.47 MHz	332.23 MHz

From the table 4 it can be seen that for the complex number implementations the ALM usages have almost tripled and the total register usage has increased four times for the square root and division version and more than doubled for the PWL inverse square root version. As the complex number implementation requires real and imaginary part processing resource usage increase is expected. The clock rate is lowered for the square root and division function implementation but it stays nearly the same for the PWL version.

The bitwidths were optimized the same way as for the fixed-point real number PE implementations. The square root and division function PE implementation was able to give at least four decimal precision for the inverse norm, kernel and output sub-vector values. With the same bitwidths the PWL inverse square root version could only achieve a precision of one to three decimal places. The results are presented in appendix C table C2.

In real and complex number PEs the inverse norm calculation is done with real numbers. Therefore, the square root and division functionality remains the same and the latencies for the square root and division functions remain the same. Although, the overhead from the complex number processing introduces increased latency somewhere else inside the PEs and the total latency of the PEs is increased. The latency and throughput results are presented in table 5.

For the unrolling of the division and square root functions similar results were achieved as for the real number implementation. Dependencies between the iterations prevent parallelization of the logic and worse results are achieved. The results are shown in table 6.

To summarize, the inverse square root logic can be implemented with the functions provided by the AC libraries. The PWL inverse square root is able to reduce the resource usage and latency but is less accurate and has lower clock rate than the square root and division combined.

Table 5: The latency results for the fixed-point complex number PE implementations.

	PE, Sqrt&Div	**PE, Inverse Sqrt**
PE Latency	90 Cycles	57 Cycles
Square root Latency	15 Cycles	N/A
Division latency	28 Cycles	N/A

Table 6: The resource usage, maximum clock rate and latency results for the unrolled fixed-point square root and division complex number PE implementation.

Resource	**PE, Sqrt&Div Unrolled**
ALMs	2139
DSPs	1
Tot. Registers	1747
Max. Clock Rate	312.4 MHz
PE Latency	99 Cycles

4.1.3 Floating-point Possibilities

In section 3.2.2 two possibilities for PE floating-point operation were discussed. One possibility is directly using floating-point numbers and the PWL inverse square root function in the PE and the other requires fixed-point to floating point conversion after the square root operation for the division to be done in floating point.

The PWL inverse square root PE was implemented using real IEEE single precision [39] floating-point numbers. The resource usage is compared to the fixed-point real number PE implementation using the inverse square root function in table 7. The floating-point implementation was not able to produce any improvement compared to the fixed-point implementations and the floating-point implementation was able to produce similar results to the PWL inverse square root implementation in terms of accuracy. The accuracy level was two decimal places as can be seen from table C3 in appendix C.

The results presented in this section suggest that a floating-point implementation using the PWL inverse square root function would not bring considerable improvements when compared to the fixed-point versions. The loss in accuracy for the PWL inverse square root function may become a prohibitive factor when the Haar-like transformations are used in larger systems. Therefore, further investigation of the PWL inverse square root function was exclueded from the scope of this master's thesis.

Table 7: The resource usage and maximum clock rate for the fixed-point real number PE implementations.

Resource	PE, Fixed-point	PE, Floating-point
ALMs	497	2284
DSPs	1	1
Tot. Registers	714	1857
Max. Clock Rate	336.81 MHz	299.58 MHz

As discussed previously, implementing the division operation in floating-point arithmetic could improve the accuracy of the inverse norm output value. This has the potential to increase the accuracy of the whole PE operation. While this method remains feasible, it was not investigated further due to limitations in the master's thesis time frame. A third possibility is always to implement a completely new inverse square root function that is optimized for the Haar-like transformation hardware architecture.

4.2 Class Based Hierarchy

In this section, the design optimization and results are discussed for the class based hierarchy design presented in Section 3.2.3. The design optimization and results for the real number fixed-point implementaiton are discussed in Section 4.2.1 and the fixed-point complex number implementation is covered in Section 4.2.2.

4.2.1 Fixed-point Real Number Implementation

As a first step in the implementation, the bitwidths of the PE operations were optimized. As the expected input range was known and all the PEs were implemented with the square root and division functions, the bitwidths could be minimized while a four-decimal accuracy level was maintained for at least 80% of the output vector elements. The output vectors of the fixed-point C++ implementations where compared against the floating-point Matlab reference to verify the accuracy level. The bitwidths used in the implementations of all transformation sizes are presented in appendix B.

In table 8, the accuracy levels maintained for each transformation size are presented. In the first column the total number of all the non-zero the output vector elements from all the test vectors are presented. Notice that the output vector of the transformation for the generating vector has only one non-zero output vector element as discussed in section 2.2.2 (see figure 2).

Thus, the total number of output elements is the one non-zero element from output vector resulting form the the generating vector plus the full output vector resulting from the input test vector in multiplication mode. In the second column the total number of non-zero output elements that had above four-decimal accuracy

are shown and in the third column the total number of non-zero output elements below the four-decimal accuracy level are shown. In the fourth column total accuracy percentages are presented.

Table 8: The results for the real number implementation with at least 80% four-decimal accuracy level. The table is showing the total number of elements from all test vectors, elements above and below the level and the total percentage.

Size	**Total**	**Above**	**Below**	**Percentage**
N = 32	33	32	1	93.9394%
N = 26	27	24	3	88.8889%
N = 16	17	14	3	82.3529%
N = 8&7	17	14	3	82.3529%
N = 6&5	13	11	2	84.6154%
N = 4&3	9	8	1	88.8889%
N = 2	3	3	0	100.000%

After the sufficient bitwidths were found, the next optimization task was to further reduce the resource usage by mapping the kernel arrays to memory elements. In the Catapult HLS tool it is possible to choose how the arrays of the C++ descriptions are to be mapped in the FPGA. The kernel arrays were mapped to the dual-port MLAB and M20K memory elements as well as registers. The total ALM and register utilizations are shown in figure 20 when the kernel array is mapped to MLABs, M20Ks and registers.

From figure 20 it can be seen that the ALM usage could not be significantly reduced by mapping the kernel arrays to different memory elements. The total register usage was lowest for the transformation sizes below $N = 5\,\&\,6$ when the kernel array was mapped to registers. On the other hand, the total register usage was minimized when the kernel array was mapped to the memory elements for the largest transformation size $N = 32$.

The maximum achieved clock rates are presented in figure 21. The results in figure 21 did not provide conclusive evidence to which memory element the kernel array should be mapped to achieve the highest clock rate for all sizes. The results presented in this section suggest that each transformation hardware needs to be considered individually when choosing the memory element to which the kernel array is to be mapped.

As the iterative square root and division functions were used to implement the inverse square root logic, the latencies for the transformations were high. To reduce the latencies, the square root and division functions were fully unrolled with the Catapult HLS tool. All the arrays were mapped to registers to minimize the latencies since if arrays are mapped the memory elements the latency would only increase.

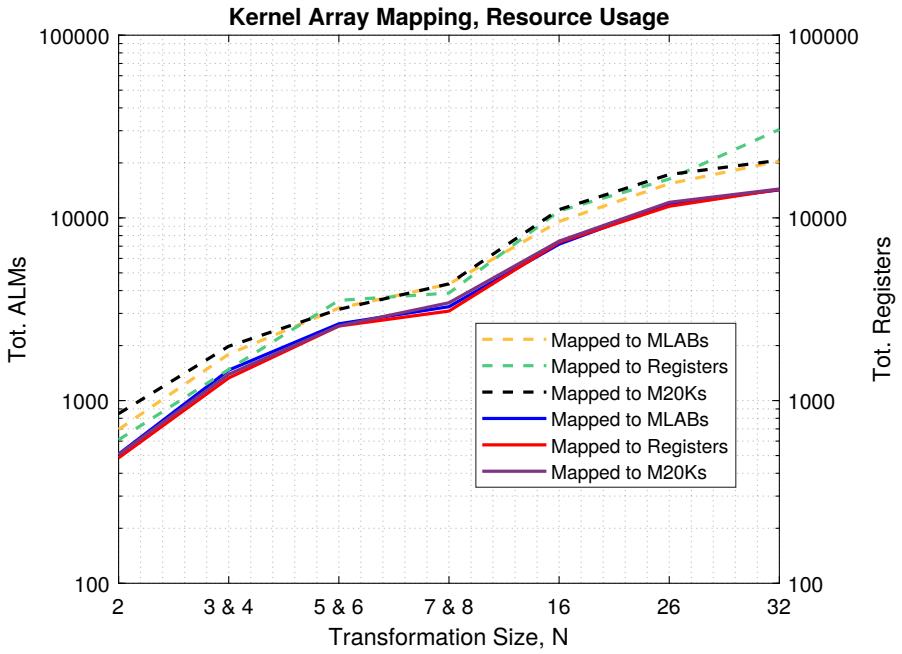

Figure 20: Resource usage when kernel arrays are mapped to memory elements. Dashed lines represent the total register usage and solid lines represent the total ALM usage.

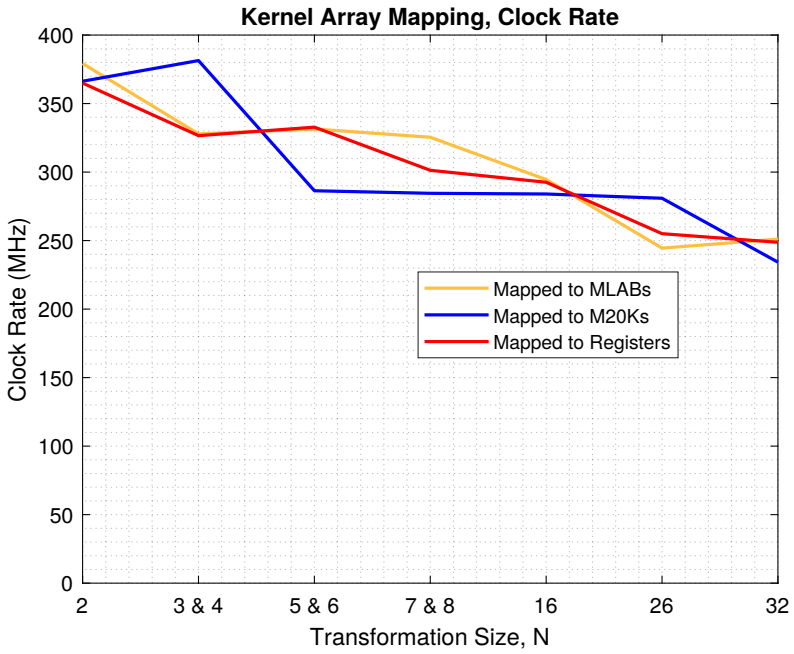

Figure 21: The maximum achieved clock rate when kernel arrays are mapped to memory elements.

In figure 22 the latencies and throuputs are compared when the transformations are utilizing fully unrolled or rolled square root and division functions. From the figure it can be seen that the latencies can be significantly reduced by unrolling the iterations in the division and square root functions. Although, this comes at a high cost with regards to increased resource usage and decreased clock rates as is seen from the figures 23 and 24.

When the throughputs from figure 22 were investigated with the Catapult HLS tool, it was discovered that the throughput always equals the latency of the first stage that has the highest latecy of all the stages. In other words, the throughput of the whole transformation is limited by the latency of the first stage. The throughput is lower than the latency for the transformation due to the FIFO data transfer between the stages (introduced by the AC channels). Thus, the design is more *»pipelined»* as discussed in section 3.1.2. Although, the control signals do not exactly match the VHDL design presented in 3.1.2.

The differencies in resource usage and clock rates between the fully rolled and unrolled versions are significant. For example, the unrolled $N = 32$ transformation had almost five times higher ALM usage than the rolled version. Also, the maximum achieved clock rate is approximately 50 MHz lower for the unrolled version. Perhaps, less extreme trade-offs could be achieved by partially unrolling the functions.

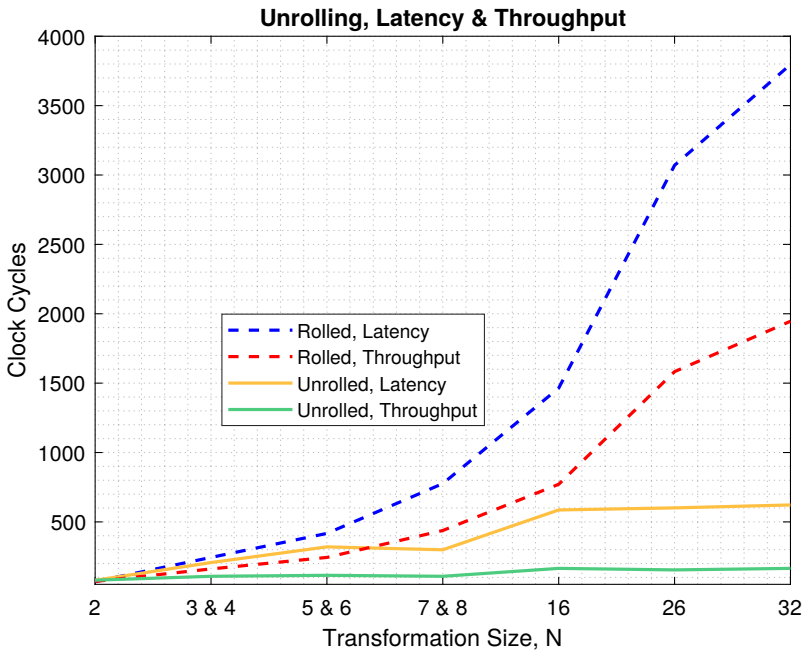

Figure 22: Latencies of the class based fixed-point real transformations when the square root and division functions are fully unrolled and rolled.

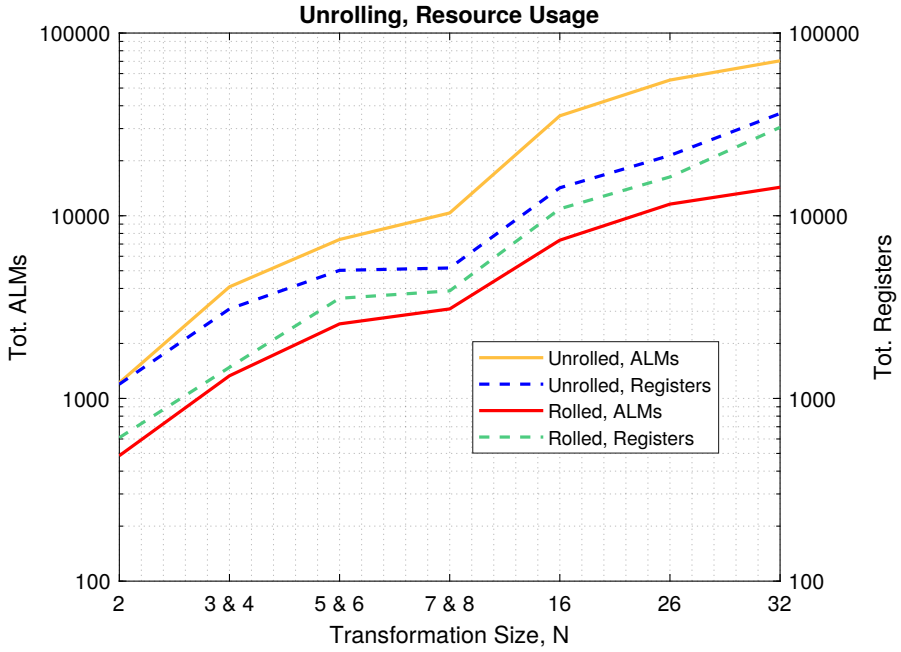

Figure 23: Resource usage of the class based fixed-point real transformations when the square root and division functions are fully unrolled and rolled

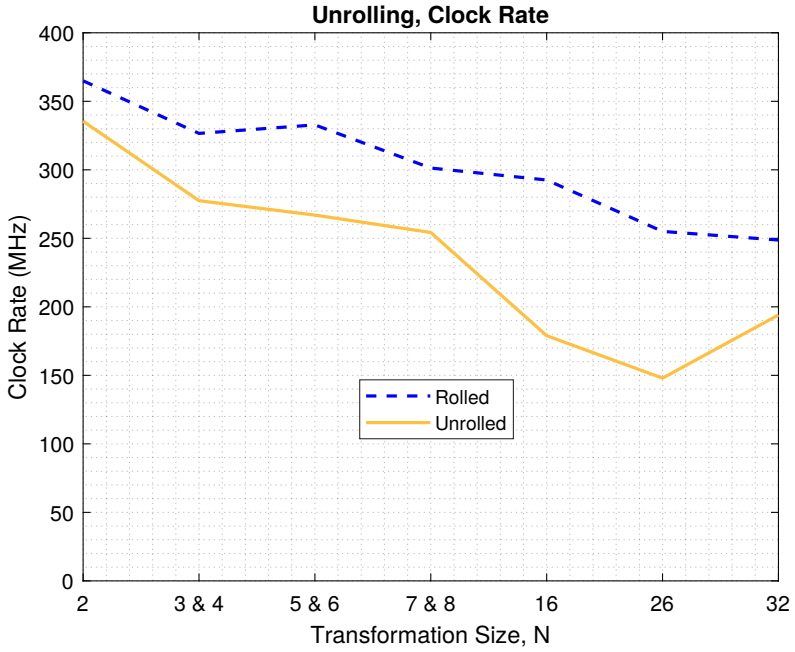

Figure 24: Maximum achived clock rates of the class based fixed-point real transformations when the square root and division functions are fully unrolled and rolled

4.2.2 Fixed-point Complex Number Implementation

The same optimization steps were repeated for the complex number implementation. The bitwidths were optimized with the same four-decimal accuracy level for 80% of the output values. For a value to be counted as above the accuracy-level both the imaginary and real part needed to have at least four-decimal accuracy when compared to the Matlab reference calculations. The results are shown in table 9. The bitwidths used in the transformations are presented in appendix B.

After the bitwidths were found, the kernel arrays were again mapped to different memory elements. As can be seen from figure 25, similar results were achieved for the complex number implementation as for the real number implementations. When compared to the real number implementation the overall resource usage is higher. This is expected since complex number processing requires logic for the real and imaginary parts. For example, two kernel arrays, one for imaginary and one for real values, are inferred from the C++ model by the Catapult HLS tool.

Table 9: The results for the complex number implementation with at least 80% four-decimal accuracy level. The table is showing the total number of elements from all test vectors, elements above and below the level and the total percentage.

Size	**Total**	**Above**	**Below**	**Percentage**
N = 32	33	32	1	93.9394%
N = 26	27	24	3	88.8889%
N = 16	17	14	3	82.3529%
N = 8&7	17	14	3	82.3529%
N = 6&5	13	11	2	84.6154%
N = 4&3	9	8	1	88.8889%
N = 2	3	3	0	100.000%

Again, the ALM usage could not be significantly affected by mapping the kernel arrays to different memory elements. The ALM utilization for the smallest transformation size, $N = 2$, was over 1000 ALMs and the ALM utilization for the largest transformation, $N = 32$, was more than two times higher for the complex number implementation than for the real number implementation. Also, the variation in the total register usage was higher.

From the results in figure 25 it cannot be confirmed to which memory element the kernel arrays should be mapped to achieve the minimal overall resource usage for all sizes. The maximum achieved clock rates are shown in figure 26. Mapping the kernel arrays to M20K memory elements achieved highest or nearly the highest clock rates for five of the transformation sizes. For the largest transformation size $N = 32$ the MLAB implementation clearly scored highest clock rate. The results in figure 26 provide some evidence that mapping the kernel arrays to the largest M20K memory elements could be beneficial when higher clock rates are wanted in the complex number transformations.

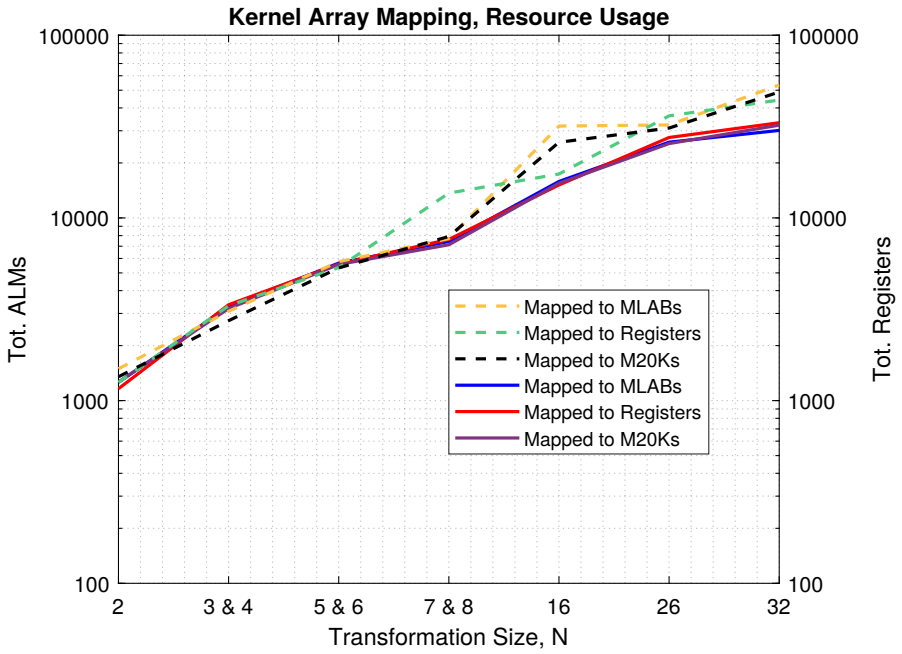

Figure 25: Resource usage when kernel arrays are mapped to memory elements. Dashed lines represent the total register usage and solid lines represent the total ALM usage.

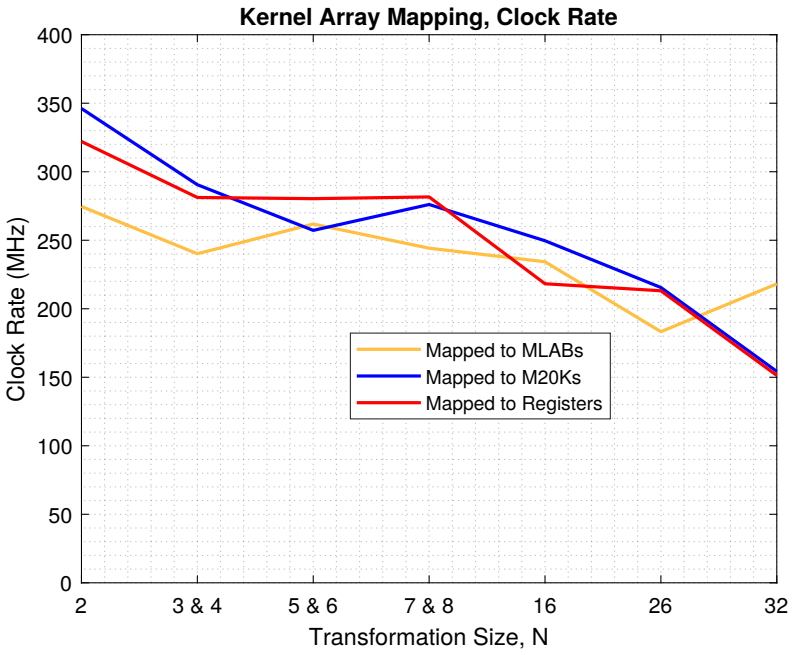

Figure 26: The maximum achieved clock rate when kernel arrays are mapped to memory elements.

The effects of unrolling the square root and division functions were investigated and similar results were achieved. As can be seen from figure 27, the latencies can be reduced significantly when unrolling is used but this comes again at a high cost in terms of resource usage and clock rate. Again, the throughput of the transformation was limited by the first stage.

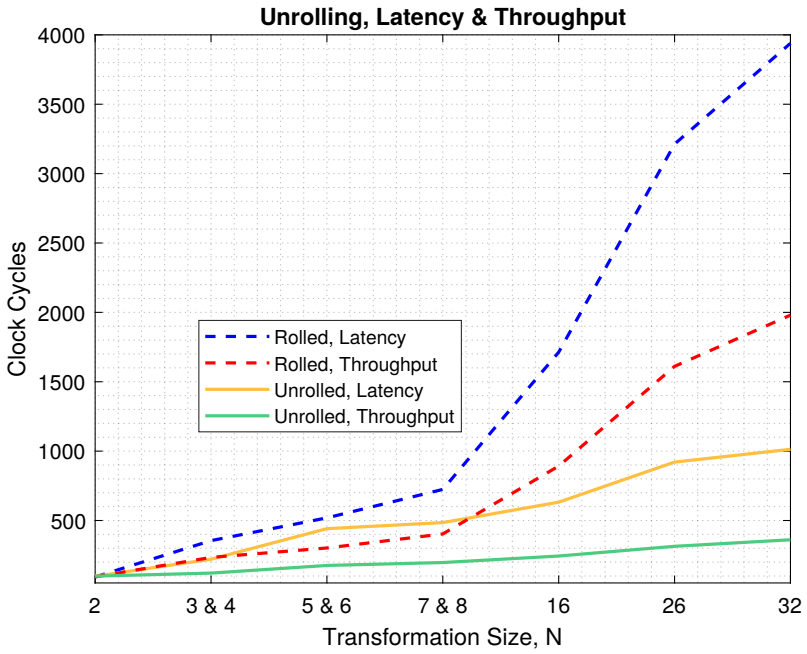

Figure 27: Latencies of the class based fixed-point complex transformations when the square root and division functions are fully unrolled and rolled.

The resource usage and the clock rates are illustrated in figures 28 and 29. From figure 28 it can be seen that the total register utilization may be in fact lower in some cases when the functions are unrolled. When the results form figure 28 are compared to the real number implementation in figure 23, the resource usage was higher in absolute numbers but the relative increase in the resource usage between the unrolled and rolled versions was lower for the complex number implementation. For example, the real number transformation of size $N = 32$ had almost 5 times higher ALM utilization for the unrolled version but the complex number version had only 2.5 times higher ALM utilization for the unrolled version.

As can be seen from figure 29, the relative decrease in clock rates is smaller for the complex number implementation than the real number implementation. For the complex number transformation of size $N = 3\,\&\,4$ the clock rate was actually higher for the unrolled version than the rolled version.

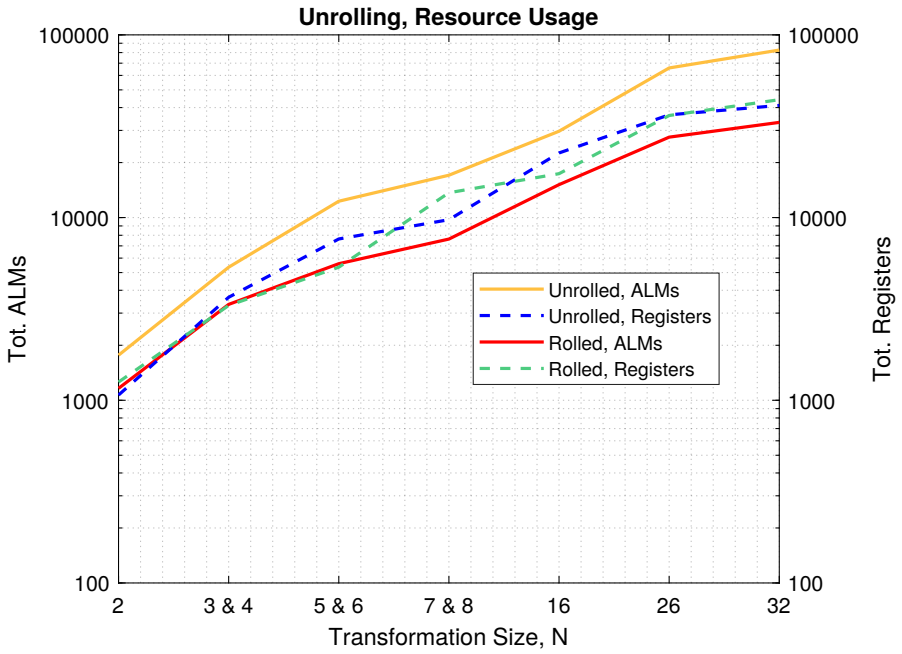

Figure 28: Resource usage of the class based fixed-point complex transformations when the square root and division functions are fully unrolled and rolled.

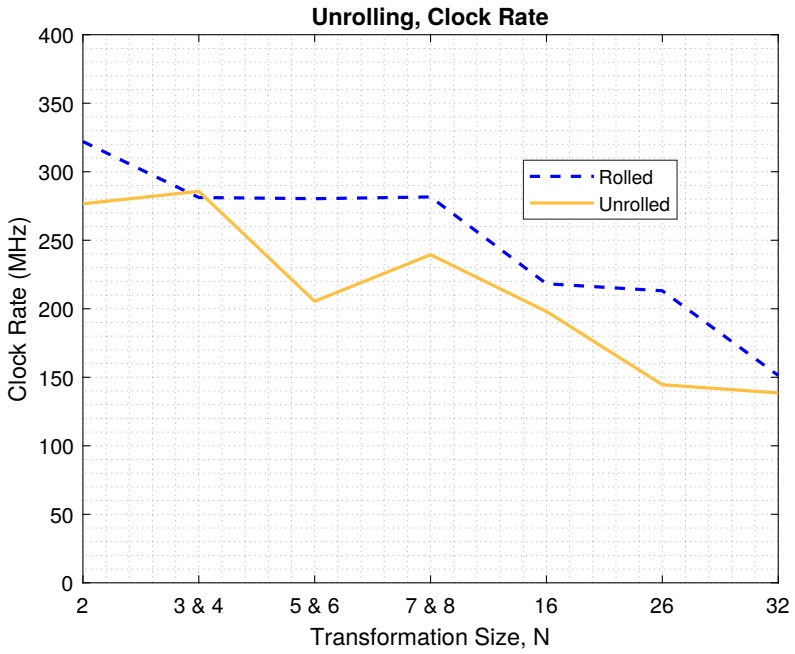

Figure 29: Maximum achived clock rates of the class based fixed-point complex transformations when the square root and division functions are fully unrolled and rolled.

To conclude, the real number implementations have higher clock rates, smaller resource usage and latencies. This is largely due to the fact that the complex number implementations need real and imaginary part processing and partly because smaller bitwidths were needed to achieve the wanted accuracy levels for the real number implementations. Some evidence pointed to that mapping the kernel arrays to M20Ks may improve the clock rates for complex number implementations. For class based hierarchies a more *»pipelined»* operation is achieved and the throughput is limited by the latency of the first stage. Latencies of the transformations can be significantly reduced by unrolling the square root and division functions. Although, this comes at a high cost in reduced clock rates and increased resource usage.

4.3 Flat Hierarchy

In this section the results for the flat hierarchy designs described in section 3.2.4 are presented. All the optimization steps done for the class based hierarchy design were repeated for the flat hierarchy design. The optimization results for the fixed-point real number design are presented in section 4.3.1 and the results for the fixed-point complex number design are presented in section 4.3.2.

4.3.1 Fixed-point Real Number Implementation

As the PE functionality in C++ was identical for the flat hierarchies and the class based hierarchies, the same accuracy results were achieved when the same bitwidths from appendix B were employed (see table 8). Again, the kernel arrays were mapped to dual-port MLAB, M20K and registers in the Catapult HLS tool to see the effects on resource usage and clock rate. The synthesis results for resource usage are shown in figure 30.

The ALM usage could not be significantly affected by the memory mapping but the overall total register usage was lowest when the kernel array was mapped to M20K or MLAB memory elements. When compared to the results from the class based hierarchy in figure 20, the overall ALM utilization for the flat hierarchy was significantly lower for transformation sizes above $N = 7 \,\&\, 8$. Although, the total register usage spiked for transformation sizes $N = 16$ and $N = 26$ when the kernel arrays were mapped to registers.

The maximum achieved clock rates are presented in figure 31. When the kernel array was mapped to MLABs the flat hierarchy achieved highest clock rates for five transformation sizes. For the flat and class based hierarchies the clock rates are similar for transformation sizes $N = 7 \,\&\, 8$ and below. For larger sizes, the clock rates start to deteriorate more rapidly for the flat hierarchy designs. For example, the transformation of size $N = 32$ achieved a maximum clock rate of roughly 150 MHz while the maximum clock rate for the class based design was approximately 100 MHz higher (see figure 21).

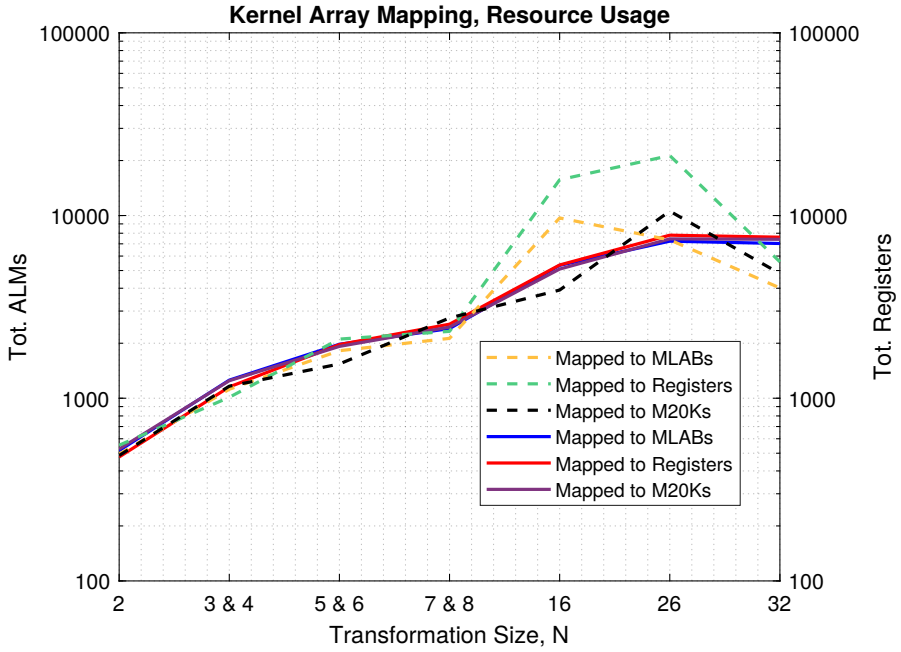

Figure 30: Resource usage when kernel arrays are mapped to memory elements. Dashed lines represent the total register usage and solid lines represent the total ALM usage.

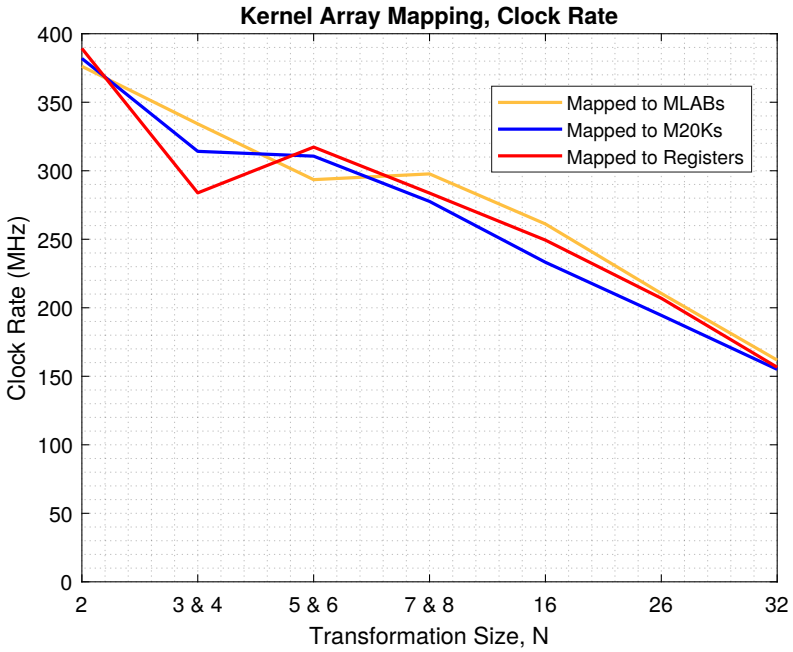

Figure 31: The maximum achieved clock rate when kernel arrays are mapped to memory elements.

The unrolling of the division and square root functions were investigated for the flat hierarchy design as well. Again, the latency can be significantly reduced by unrolling the functions but this comes at a high cost in resource usage and reduction in clock rate. As there are no channels between the stage loop iterations implementing FIFO data transfer, the throughput of the whole transformation equals the latency of the transformation. The latencies and throughputs for the unrolled and rolled versions shown in figure 32.

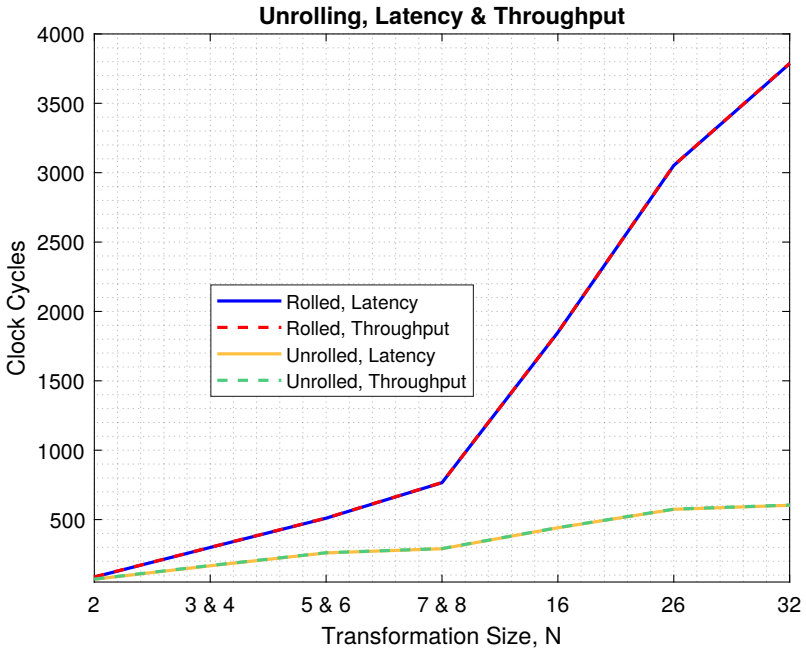

Figure 32: Latencies of the flat hierarchy fixed-point real transformations when the square root and division functions are fully unrolled and rolled.

The synthesis results are presented in figure 33 for resource usage and in figure 34 for the clock rates. The overall resource usage was again significantly higher for the transformations when the division and square root functions are unrolled. Although, the total register usage is similar for unrolled and rolled versions for transformation sizes $N = 16$ and $N = 26$ due to the spike. When compared to the class based implementation in figure 23, the unrolled versions are very similar from resource usage point of view. The rolled versions, on the other hand, differ due to the spike in total resource usage experienced for sizes $N = 16$ and $N = 26$ as well as the lower overall ALM utilization.

In figure 34 the achieved clock rates are shown. The clock rates for the unrolled versions of the flat hierarchies have less variance than for class based implementations in figure 24. Thus, the relative drop in clock rates is also more constant. The relative drop between the unrolled and rolled versions is approximately 70-100 MHz, except for $N = 3 \,\&\, 4$.

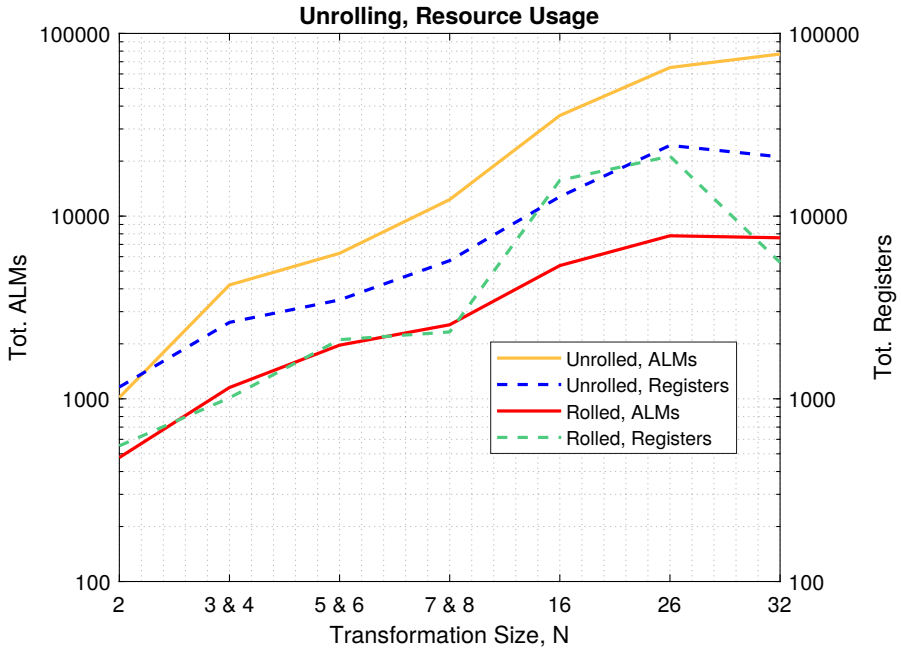

Figure 33: Resource usage of the flat hierarchy fixed-point real transformations when the square root and division functions are fully unrolled and rolled.

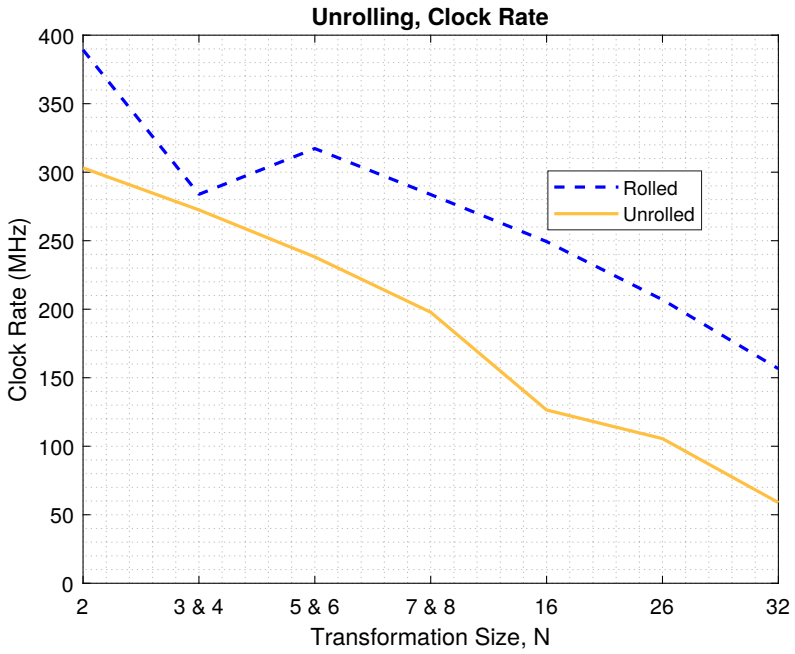

Figure 34: Maximum achived clock rates of the flat hierarchy fixed-point real transformations when the square root and division functions are fully unrolled and rolled.

4.3.2 Fixed-point Complex Number Implementation

The same accuracy results were achieved for the complex flat hierarchy design as for the class based design when the same bitwidths from appendix B were used (see table 9). To investigate possible resource usage reductions, the kernel array mapping was again investigated. For the complex number implementation, no spike in total register usage was shown for sizes $N = 16$ and $N = 26$. The overall total register usage was actually lowest when the kernel arrays were mapped to registers. An unintuitive result. The ALM usage had more variation than previously. Although, this variation was still small and no significant reductions could be achieved in ALM usage. The results are presented in figure 35.

When compared to the class based complex number implementation in figure 25, the flat hierarchy implementation had significantly lower resource usage for transformation sizes above $N = 7$ & 8. For example, the largest transformation size $N = 32$ for the class based hierarchy implementation uses approximately 10,000 ALMs more than its flat hierarchy counterpart.

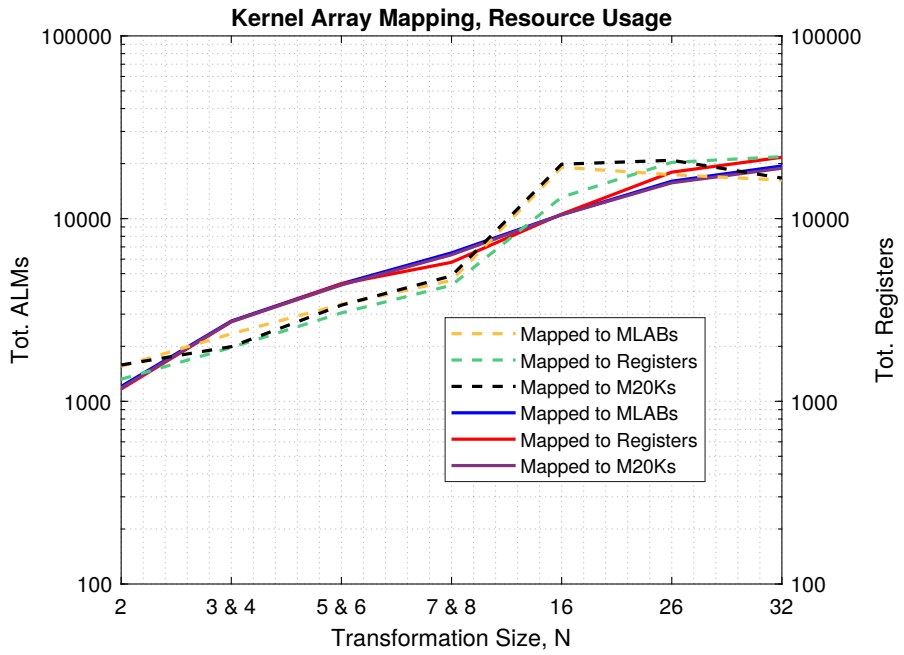

Figure 35: Resource usage when kernel arrays are mapped to memory elements. Dashed lines represent the total register usage and solid lines represent the total ALM usage.

The maximum achieved clock rates are presented in figure 36. For the complex implementation, the results do not indicate a single kernel array mapping method that would achieve the highest clock rates for all sizes. When the results are compared to the class based implementation in figure 26 the overall trend in both cases is similar. Although, the flat hierarch usually scores roughly 10-60 MHz lower than the class based hierarchy.

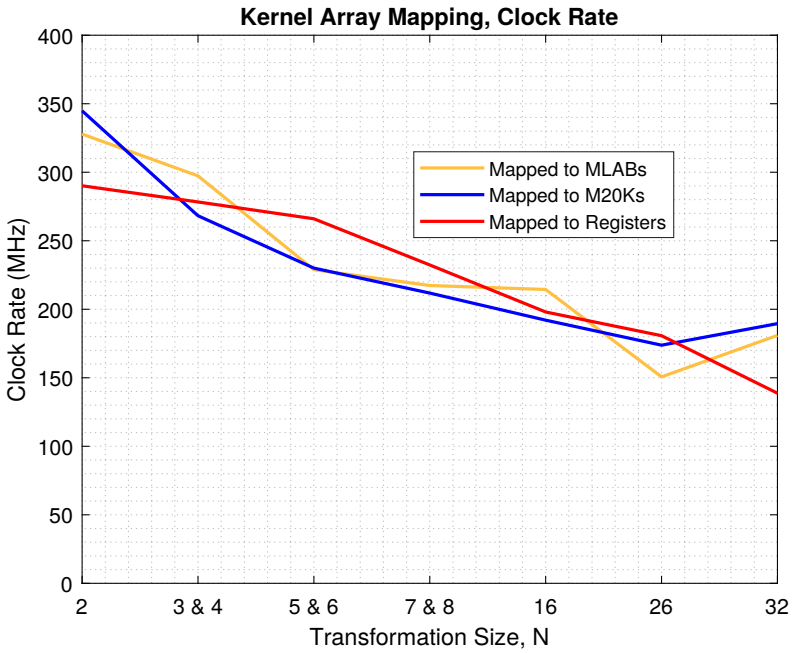

Figure 36: The maximum achieved clock rate when kernel arrays are mapped to memory elements.

Again, the latency and throughput results for the rolling and unrolling of the square root and division functions were similar to the previous results. Latency can be reduced at the expense of resource usage and clock rate and the throughput of the transformation equals the latency of the transformation. The latencies and throuhputs are presented in figure 37.

The resource usages are presented in figure 38. From the figure it can be seen that the resource usage stops increasing when the largest transformation size $N = 32$ is reached. When the unrolled results from figure 38 are compared to the unrolled complex class based implementation in figure 28, the ALM usage is similar but the total register utilization is smaller particularly for sizes larger than $N = 7 \,\&\, 8$. For example, for transformation sizes $N = 32$ and $N = 26$ the difference in total register utilization is almost 20,000 registers.

The clock rates for the unrolled and rolled versions are shown in figure 39. Again, for the flat hierarchies the variance in clock rates is smaller than for the class based versions. Thus, the difference between rolled and unrolled versions is also more constant. The difference is approximately 40-80 MHz for transformation sizes above $N = 3 \,\&\, 4$. Although, the clock rates for the complex class based hierarchy implementations are usually 20-50 MHz higher (see figure 29).

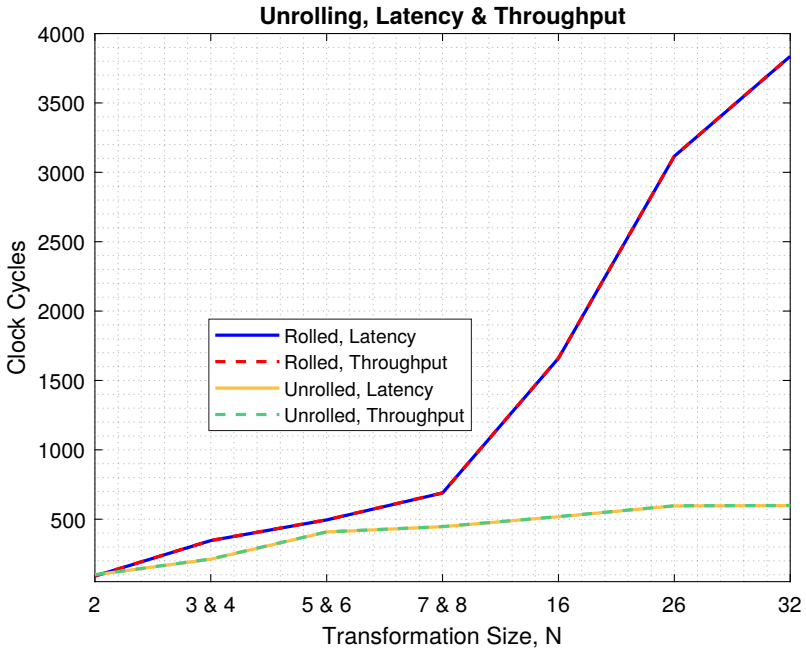

Figure 37: Latencies of the flat hierarchy fixed-point complex transformations when the square root and division functions are fully unrolled and rolled.

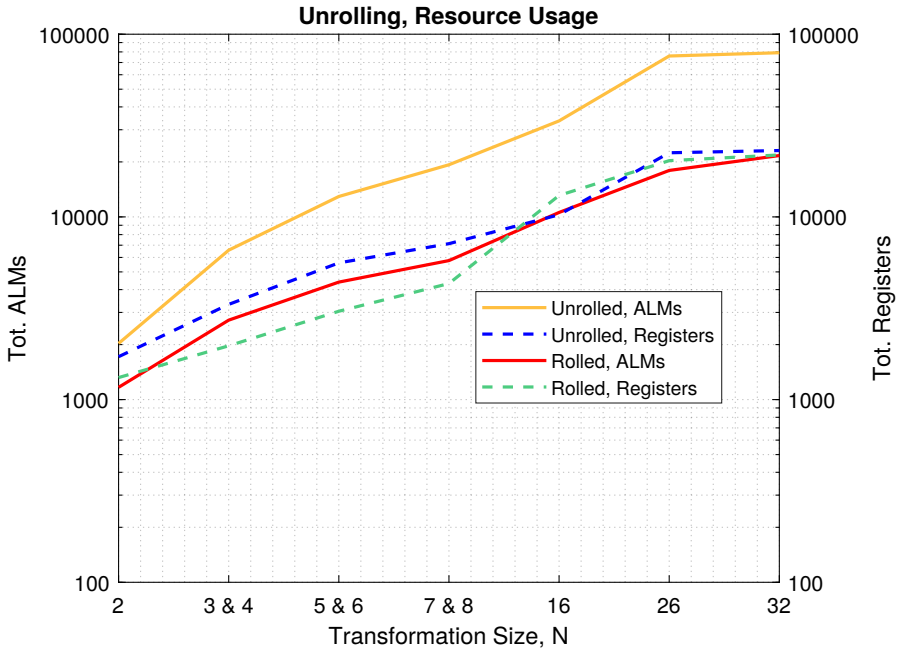

Figure 38: Resource usage of the flat hierarchy fixed-point complex transformations when the square root and division functions are fully unrolled and rolled.

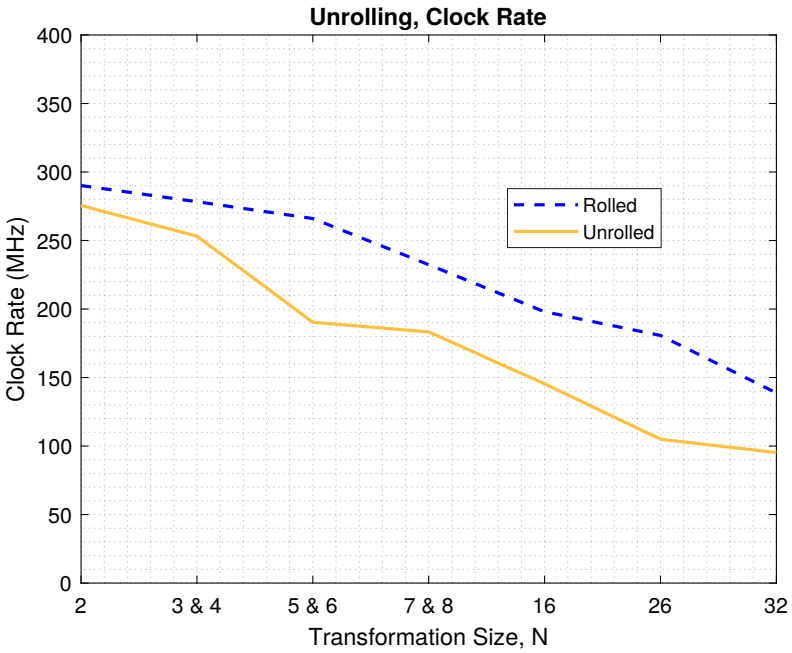

Figure 39: Maximum achived clock rates of the flat hierarchy fixed-point complex transformations when the square root and division functions are fully unrolled and rolled.

The main findings for the flat hierarchies were that in general they used less resources than the class based hierarchies, particularly for sizes above $N = 7 \,\&\, 8$, but they also had lower clock rates. Some evidence was obtained that for real number implementations resource usage can be reduced and the clock rate increased by mapping the kernel array to MLABs. For complex number implementations the resource usage was reduced by mapping the kernel arrays to registers.

The latencies are in the same order of magnitude for the flat and class based hierarchies. Because no hierarchy and AC channels are present in a flat hierarchy, the throughput of the whole transformation equals the latency. This is a drawback of the flat hierarchy. Also for flat hierarchies, the latencies can be significantly reduced by unrolling the division and square root functions at the cost of increased resource usages and decreased clock rates.

4.4 Summary and Design Questions

In this section a summary for the class based and flat hierarchy implementations is given and answers to the the design questions presented in Section 3.1.4 are formulated.

In general, the resource usage for the class based hierarchies was found to be higher for most of the transformation sizes than for the flat hierarchies (with rolled division and square root functions). On the other hand, higher clock rates and throughputs were achieved for the class based hierarchies for most of the transformation sizes.

In table 10 a qualitative comparison is presented for the class based and flat hierarchies with respect to the performance metrics. In the table the class based real number implementations are compared against their flat hierarchy counterparts. Color coding is used to indicate when the relationship is better (green), similar (orange) or worse (red) for the class based hierarchies. This comparison is also a good approximation of the differences for the complex number implementations.

Table 10: Class based real number implementations compared to their flat hierarchy counter parts. Color coding is used to indicate when the relationship is better (green), similar (orange) or worse (red) for the class based hierarchies.

Size	**Tot. ALMs**	**Tot. Regs**	**Clock Rate**	**Throughput**
N = 32	Higher	Higher	Higher	Higher
N = 26	Higher	Higher	Higher	Higher
N = 16	Higher	Higher	Higher	Higher
N = 8&7	Higher	Higher	Higher	Higher
N = 6&5	Higher	Higher	Higher	Higher
N = 4&3	Higher	Higher	Similar	Higher
N = 2	Similar	Similar	Similar	Similar

Here the answers to the design questions presented in Section 3.1.4 are formulated.

- The inverse square root logic can be implemented with the functions provided by the AC libraries in Catapult HLS as demonstrated in Section 4.1. As expected, the inverse square root logic had multicycle operation and this reflected on the behavior of the whole system. By unrolling iterative loops in the inverse square root logic latency could be reduced but this came at a high cost in reduced clock rates and increased resource usages.

- With the PWL inverse square root function in use, accuracies from 1 to 3 decimals were achieved when the PE logic was implemented with bitwidths ranging from 3 to 26 bits for real number implementations and 8 to 26 bits for complex number implementations. When the square root and division functions were used, above 4 decimal accuracies for 80% of the output values could be achieved with the same bitwidths. No improvement could be achieved with floating-point arithmetic with the PWL function in use. Although, it remains feasible that accuracy could be increased by doing the division in floating-point. This requires fixed- to floating-point conversion logic between the square root and division functions. Another possibility is always to implement a completely new inverse square root function.

- With the complex AC data types extending the real number implementation to a complex number implementatoin was easily achieved with Catapult HLS. As is expected, the complex number implementations utilized more resources than the real number implementations and had lower clock rates.

5 Conclusions

The aim of this master's thesis was to study how parametric Haar-like transformations could be efficiently used as a part of a larger FPGA based multimedia or telecommunications system. From the parametric equations describing unitary transformations the Haar-like transformations could be specified by defining a set of parameters. Hence, the name parametric Haar-like transformations.

Algorithms for generation and multiplication as well as flow graph representations could be derived from the equations to describe the functionality of the parametric Haar-like transformations. By utilizing the algorithms and flow grapsh, unified hardware architectures capable of implementing both algorithms could be created. Thus, the hardware architectures worked in a generation mode where the Haar-like transformation matrix was generated with a generating vector and a multiplication mode where the input matrix was multiplied with the generated transformation matrix using a fast algorithm.

The hardware architectures were first explored with VHDL and during the exploration three design questions were formulated. In effort to answer these design questions and to complete the thesis in a timely manner, the final RTL models for the FPGA synthesis were implemented using Catapult HLS. Two different hardware architectures were created. A class based hierarchy and a more generic algorithmic style description where the hierarcies were flattened, named as a flat hierarchy.

Also, the PEs utilized in the architectures were synthesized separately to examine the inverse square root and floating-point arithmetic implementation possibilities. In Catapult HLS the inverse square root logic could be implemented using a PWL approximation function or the standard AC square root and division functions. The square root and division functions were found to be faster and more accurate but the resource utilization was also higher.

In Catapult HLS both complex and real number implementations could easily be created using the complex AC data types. The synthesis results suggested that the ALM utilization could not be affected by mapping the kernel array to embedded memory elements in the FPGA fabric. For register utilization the results were not conclusive.

A more *»pipelined»* operation was achieved for the class based hierarchy and the throughput of the hardware was limited by the latency of the first stage. For the flat hierarchies the throughput was equal to the latency of the whole transformation due to the more algorithmic description. Decreasing the latencies was possible by unrolling the square root and division functions but this came at a high cost in terms of increased resource usage and decreased clock rates.

In this master's thesis the FPGA implementations were created and the design questions were answered. In general, the class based hierarchies might be more suitable for FPGA applications where faster clock rates and higher throughputs are needed. On the other hand flat hierarchies might provide smaller resource utilization on the FPGA. To conclude, the parametric Haar-like transformations are a feasible linear transformation candidate to be used in FPGA based multimedia or telecommunications systems.

References

[1] Golub, G. H., Van Loan, C. F. *Matrix Computations.* 4. Edition. Baltimore, The Johns Hopkins University Press, 2013.

[2] Cooley, J. W., Tukey, J. W. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation,* 1965, vol. 19, pp. 297–301.

[3] Guevorkian, D., Agaian, S. Synthesis of a class of orthogonal transforms parallel SIMD algorithms and specialized processors. *Pattern Recognition and Image Analysis,* 1992, vol. 2, no 4, pp. 396–416.

[4] Minasyan, S., Guevorkian, D., Agaian, S. and Sarukhanyan, H. On "slant-like" fast orthogonal transforms of arbitrary order. *VIPromCom-2002, 4th EURASIP - IEEE Region 8 International Symposium on Video/Image Processing and Multimedia Communications*, Zadar, Croatia, 2002.

[5] Minasyan, S., Astola, J. and Guevorkian, D. On unified architectures for synthesizing and implementation of fast parametric transforms. *5th International Conference on Information Communications and Signal Processing*, Bangkok, Thailand, 2005.

[6] Minasyan, S., Astola, J. and Guevorkian, D. An image compression scheme based on parametric Haar-like transform. *IEEE International Symposium on Circuits and Systems*, Kobe, Japan, 2005.

[7] Minasyan, S., Astola, J., Egiazarian, K. and Guevorkian, D. Parametric Haar-like Transforms in Image Denoising. *IEEE International Conference on Image Processing*, Atlanta, USA, 2006.

[8] Rabaey, J. M., Chandrakasan, A. and Nicolic, B. *Digital Integrated Circuits, A design Perspective* 2. Edition. New Jersey, Pearson Education, Inc., 2003.

[9] Intel, Stratix 10 Product Table. Web document. Cited 25.2.2018. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/pt/stratix-10-product-table.pdf

[10] Intel, Stratix 10 Logic Array Blocks and Adaptive Logic Modules User Guide. Web document. 2017. Cited 25.2.2018. Available: https://www.altera.com/en_US/pdfs/literature/hb/stratix-10/ug-s10-lab.pdf

[11] Intel, Stratix 10 GX/SX Device Overview. Web document. 2017. Cited 25.2.2018. Available: https://www.altera.com/en_US/pdfs/literature/hb/stratix-10/s10-overview.pdf

[12] Ashenden, P. J. *The Designer's Guide to VHDL.* 3. Edition. Burlington, Morgan Kaufmann Publishers, 2009.

[13] Bollaert, T. Catapult Synthesis: A Practical Introduction to Interactive C Synthesis. In work: Coussy, P., Morawiec, A. *High-Level Synthesis from Algorithm to Digital Circuit.* Springer, 2008, pp. 29–52.

[14] Strang, G. *Introduction to Linear Algebra.* 4. Edition. Wellesley, Wellesley-Cambridge Press, 2009.

[15] Istoan, M., Pasca, B. Fixed-Point Implementations of the Reciprocal, Square Root and Reciprocal Square Root Functions. *Open HAL Archive*, 2015. Cited 24.2.2018. Available: https://hal.archives-ouvertes.fr/hal-01229538

[16] Lachowicz, S., Pfleiderer, HJ. Fast Evaluation of the Square Root and Other Nonlinear Functions in FPGA. *4th IEEE International Symposium on Electronic Design, Test and Applications*, Hong Kong, China, 2008.

[17] Ercegovac, M. D., Lang, T., Muller, JM., and Tisserand, A. Reciprocation, square root, inverse square root, and some elementary functions using small multipliers. *IEEE Transactions on Computers* 2000, vol. 49, no 7, pp. 628–637. DOI: 10.1109/12.863031

[18] Salmela, P., Burian, A., Järvinen, T., Happonen, A., Takala, J. H. Low-Complexity Inverse Square Root Approximation for Baseband Matrix Operations. *International Scholarly Research Network ISRN Signal Processing,* 2011, vol. 2011, DOI:10.5402/2011/615934

[19] Luo, J., Huang, Q. Luo, H., Zhi, Y., and Wang, X. Hardware Implementation of Single Iterated Multiplicative Inverse Square Root. *Elektronika IR Elektrotechnika* 2017, vol. 23, no 4, pp. 18–23. DOI: 10.5755/j01.eie.23.4.18717

[20] Jose, W., Silva, A. R., Neto, H. and Vestias, M. Efficient implementation of a single-precision floating-point arithmetic unit on FPGA. *24th International Conference on Field Programmable Logic and Applications*, Munich, Germany, 2014.

[21] Borwein, J. M., Borwein, P. B. *Pi and the AGM: A Study in the Analytic Number Theory and Computational Complexity.* New York, Wiley-Interscience, 1987.

[22] Hasnat, A., Bhattacharyya, T., Dey, A., Halder, S., and Bhattacharjee, D. A fast FPGA based architecture for computation of square root and Inverse Square Root. *2017 Devices for Integrated Circuit*, Kalyani, India, 2017.

[23] Zafar, S., Adapa, R. Hardware architecture design and mapping of 'Fast Inverse Square Root' algorithm. *International Conference on Advances in Electrical Engineering*, Vellore, India, 2014.

[24] Moroz, L. V., Walczyk, C. J., Hrynchyshyn, A., Holimath, V. and Cieslinski, J. L. Fast calculation of inverse square root with the use of magic constant –

analytical approach *Applied Mathematics and Computation* 2018, vol. 316, no 1, pp. 245–255. DOI: 10.1016/j.amc.2017.08.025

[25] Lomont, C. Fast Inverse Square Root. Web document. 2003. Cited 24.2.2018. Available: https://www.lomont.org/Math/Papers/2003/InvSqrt.pdf

[26] Volder, J. E. The CORDIC Trigonometric Computing Technique. *IRE Transactions on Electronic Computers* 1959, vol. EC-8, no 3, pp. 330–334. DOI: 10.1109/TEC.1959.5222693

[27] Andraka, R. A survey of CORDIC algorithms for FPGA based computers. *FPGA '98 Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays,* 1998, pp. 191–200.

[28] Deprettere, E., Dewilde, P., and Udo, R. FPGA based architectures for high performance adaptive FIR filter systems. *ICASSP '84. IEEE International Conference on Acoustics, Speech, and Signal Processing*, San Diego, USA, 1984.

[29] Sufeng, N., Aslan, S., and Jafar, S. FPGA implementation of fast QR decomposition based on givens rotation. *IEEE 55th International Midwest Symposium on Circuits and Systems*, Boise, USA, 2012.

[30] Prabhu, G. R., Johnson, B., and Rani, J. S. FPGA Based Scalable Fixed Point QRD Core Using Dynamic Partial Reconfiguration. *28th International Conference on VLSI Design*, Bangalore, India, 2015.

[31] Aslan, S., Sufeng, N., and Jafar, S. FPGA based architectures for high performance adaptive FIR filter systems. *IEEE International Instrumentation and Measurement Technology Conference*, Minneapolis, USA, 2013.

[32] Heuring, V. P., Jordan, H. F. *Computer Systems Design and Architecture* Menlo Park, Addison Wesley Longman, Inc, 1997.

[33] Libessart, E., Arzel, M., Lahuec, C. and Andriulli, F. A scaling-less Newton-Raphson pipelined implementation for a fixed-point inverse square root operator. *15th IEEE International New Circuits and Systems Conference*, Strasbourg, France, 2017.

[34] Standard VHDL Mathematical Packages (IEEE Std 1076.2-1996, MATH_REAL) Cited 20.2.2018. Available: https://standards.ieee.org/downloads/1076/1076.2-1996/math_real.vhdl

[35] Intel, SCFIFO and DCFIFO IP Cores User Guide. Web document. 2017. Cited 25.2.2018. Available: https://www.altera.com/en_US/pdfs/literature/ug/ug_fifo.pdf

[36] Mentor Graphics, HLS Blue Book. 2017.

[37] Mentor Graphics, Catapult Synthesis User and Reference Manual. 2017.

[38] Mentor Graphics, Algorithmic C (AC) Datatypes. 2017.

[39] 754-2008. IEEE Standard for Floating-Point Arithmetic. Institute of Electrical and Electronics Engineers, 2008. DOI: 10.1109/IEEESTD.2008.4610935

A QR-Decomposition Example

In a QR-decomposition a matrix $\mathbf{A}$ is factorized to an orthogonal matrix $\mathbf{Q}$ and an upper triangular matrix $\mathbf{R}$ such that $\mathbf{A} = \mathbf{QR}$. The QR-decomposition is used in many applications and it can be used to solve problems such as least squares and eigenvalue problems. Further, exploration of the QR-decomposition is beyond the scope of this thesis.

First, we will derive the relationships between the coefficients in a 2 by 2 QR-decomposition and then the relationships will be applied in a 3 by 3 QR-decomposition example. The following example is based on the QR-decomposition algorithm using Givens rotations presented in [1]. A Givens rotation is an orthogonal plane rotation matrix of the form

$$\mathbf{G} = \begin{pmatrix}
1 & 0 & 0 & \cdots & & \cdots & 0 & \cdots & & & \cdots & 0 \\
0 & 1 & 0 & \cdots & & & \vdots & & & & & \vdots \\
0 & 0 & \ddots & & & & & & & & & \\
\vdots & \vdots & & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & & \vdots \\
 & & & 0 & c_{ii} & 0 & & 0 & s_{ji} & 0 & & \\
\vdots & & & 0 & 0 & 1 & & 0 & 0 & 0 & & \vdots \\
0 & \cdots & & \vdots & & & \ddots & & & \vdots & \cdots & 0 \\
\vdots & & & 0 & 0 & 0 & & 1 & 0 & 0 & & \vdots \\
 & & & 0 & -s_{ij} & 0 & & 0 & c_{jj} & 0 & & \\
\vdots & & & 0 & 0 & 0 & \cdots & 0 & 0 & 1 & \vdots & \vdots \\
 & & & & & & & & & & \ddots\ 0 & 0 \\
\vdots & & & & & & \vdots & & & \cdots\ 0 & 1 & 0 \\
0 & \cdots & & & & \cdots & 0 & \cdots & & \cdots\ 0 & 0 & 1
\end{pmatrix} \tag{A1}$$

Where $s_{ij} = \sin(\theta)$ and $s_{ij} = \cos(\theta)$. According too equation (A1) the 2 by 2 Givens rotation is given as

$$\mathbf{G} = \mathbf{Q} = \begin{pmatrix} c_{11} & s_{12} \\ -s_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix} \tag{A2}$$

The QR-decomposition decomposes the matrix $\mathbf{A}$ to an orthogonal matrix $\mathbf{Q}$ and an upper triangular matrix $\mathbf{R}$ so that $\mathbf{A} = \mathbf{QR} \Leftrightarrow \mathbf{Q}^T\mathbf{A} = \mathbf{R}$. This matrix equation can be written explicitly as

$$\mathbf{Q}^T\mathbf{A} = \mathbf{R} \Leftrightarrow \begin{pmatrix} c_{11} & -s_{12} \\ s_{21} & c_{22} \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} \\ 0 & r_{22} \end{pmatrix} \tag{A3}$$

From equation (A3) the following relationships can be derived for the matrix coefficients by considering the equations $s_{21}a_{11} + c_{22}a_{21} = 0$ and $c_{11}a_{11} - s_{12}a_{21} = r_{11}$

when solving for coefficients s_{ij} and c_{ij} while the coefficients are constrainde by the relationships $s_{ij} = \sin(\theta)$ and $c_{ij} = \cos(\theta)$.

$$\begin{cases} s_{21} = \sin(\theta) = \frac{-a_{21}}{r_{11}}, \quad c_{11} = \cos(\theta) = \frac{a_{11}}{r_{11}}, \quad r_{11}^2 = a_{11}^2 + a_{21}^2 \\ s_{21} = \frac{a_{21}}{\sqrt{a_{11}^2+a_{21}^2}}, \quad c_{11} = \frac{a_{11}}{\sqrt{a_{11}^2+a_{21}^2}} \end{cases} \tag{A4}$$

By applying the relationships from equation (A4) the Givens rotation matrix can be constructed from the elements of matrix $\mathbf{A}$.

$$\mathbf{Q}^T = \mathbf{G}^T = \frac{1}{\sqrt{a_{11}^2 + a_{21}^2}} \begin{pmatrix} a_{11} & a_{12} \\ -a_{21} & a_{22} \end{pmatrix} \tag{A5}$$

The equations derived above can be used to decompose a larger matrix. Let a 3 by 3 matrix $\mathbf{A}_1$ be defined as

$$\mathbf{A}_1 = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 5 \\ 2 & 3 & 0 \\ 0 & 4 & 6 \end{pmatrix} \tag{A6}$$

The decompostion of matrix $\mathbf{A}_1$ starts by applying a 3 by 3 Givens rotation matrix that maps the element a_{21} to zero. As in equation (A5) the Givens rotation matrix can be constructed from the elements of matrix $\mathbf{A}_1$.

$$\mathbf{G}_1^T = \begin{pmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{pmatrix} = \frac{1}{\sqrt{5}} \begin{pmatrix} 1 & 2 & 0 \\ -2 & 1 & 0 \\ 0 & 0 & 1/\sqrt{5} \end{pmatrix} \tag{A7}$$

Where the coefficients s and c are defined with elements of $\mathbf{A}_1$ as

$$s = \frac{a_{21}}{\sqrt{a_{11}^2 + a_{21}^2}} \quad \& \quad c = \frac{a_{11}}{\sqrt{a_{11}^2 + a_{21}^2}} \tag{A8}$$

By applying the Givens rotation matrix $\mathbf{G}_1$ to matrix $\mathbf{A}_1$ the following matrix equation is obtained.

$$\mathbf{A}_2 = \mathbf{G}_1^T \mathbf{A}_1 = \frac{1}{\sqrt{5}} \begin{pmatrix} 1 & 2 & 0 \\ -2 & 1 & 0 \\ 0 & 0 & 1/\sqrt{5} \end{pmatrix} \begin{pmatrix} 1 & 0 & 5 \\ 2 & 3 & 0 \\ 0 & 4 & 6 \end{pmatrix} \tag{A9}$$

Solving the matrix equation (A9) explicitly gives the next matrix $\mathbf{A}_2$

$$\mathbf{A}_2 = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} \sqrt{5} & 6/\sqrt{5} & \sqrt{5} \\ 0 & 3/\sqrt{5} & -2/\sqrt{5} \\ 0 & 4 & 6 \end{pmatrix} \tag{A10}$$

As the QR-decomposition seeks to decompose the original matrix as $\mathbf{Q}^T\mathbf{A} = \mathbf{R}$, where the matrix $\mathbf{R}$ is of upper triangular form, the next Givens rotation should map the element a_{32} of $\mathbf{A}_2$ to zero. Thus, the next Givens rotation matrix is defined as

$$\mathbf{G}_2^T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & c & -s \\ 0 & s & c \end{pmatrix} = \frac{\sqrt{455}}{89} \begin{pmatrix} 89/\sqrt{455} & 0 & 0 \\ 0 & 3/\sqrt{5} & 4 \\ 0 & -4 & 3/\sqrt{5} \end{pmatrix} \tag{A11}$$

Where the coefficients s and c are defined with elements of $\mathbf{A}_2$ as

$$s = \frac{a_{22}}{\sqrt{a_{32}^2 + a_{22}^2}} \quad \& \quad c = \frac{a_{32}}{\sqrt{a_{32}^2 + a_{22}^2}} \tag{A12}$$

Again by applying the Givens rotation matrix $\mathbf{G}_2$ to matrix $\mathbf{A}_2$ we can solve the matrix $\mathbf{A}_3$.

$$\mathbf{A}_3 = \mathbf{G}_2^T \mathbf{A}_2 = \frac{\sqrt{455}}{89} \begin{pmatrix} 89/\sqrt{455} & 0 & 0 \\ 0 & 3/\sqrt{5} & 4 \\ 0 & -4 & 3/\sqrt{5} \end{pmatrix} \begin{pmatrix} \sqrt{5} & 6/\sqrt{5} & \sqrt{5} \\ 0 & 3/\sqrt{5} & -2/\sqrt{5} \\ 0 & 4 & 6 \end{pmatrix} \tag{A13}$$

$$\mathbf{A}_3 = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} \sqrt{5} & 6/\sqrt{5} & \sqrt{5} \\ 0 & \sqrt{455}/5 & 18\sqrt{455}/89 \\ 0 & 0 & 58\sqrt{455}/89 \end{pmatrix} \tag{A14}$$

From the equation (A14) we can see that the upper triangular form has been achieved and the QR decomposition is finished. The orthogonal matrix $\mathbf{Q}$ is the product of the sequential transfomations, the Givens rotations, and the upper triangular matrix $\mathbf{R}$ is the final matrix $\mathbf{A}_3$.

$$\mathbf{A}_1 = \mathbf{QR}, \quad \textit{where} \quad \mathbf{R} = \mathbf{A}_3 \quad \& \quad \mathbf{Q} = \mathbf{G}_1\mathbf{G}_2 \tag{A15}$$

B Bitwidths for the Fixed-Point Operations

Table B1: The bitwidths and the radix-point placements (I-parameter) used in all the fixed-point transformations. R = Real fixed-point, C = Complex fixed-point. See figure 16 and the example PE code in section 3.2.2.

Type, N=32	**Width (R)**	**I (R)**	**Width (C)**	**I (C)**
Input_type	24	8	24	8
Output_type	24	8	24	8
Kernel_type	24	1	24	1
Sqr_type	20	14	20	14
Sum_type	20	14	20	14
Sqroot_type	26	7	26	7
Norm_type	26	0	26	0
Type, N=26	**Width (R)**	**I (R)**	**Width (C)**	**I (C)**
Input_type	24	8	24	8
Output_type	24	8	24	1
Kernel_type	24	1	24	8
Sqr_type	20	14	20	14
Sum_type	20	14	20	14
Sqroot_type	26	7	26	7
Norm_type	26	0	26	0
Type, N=16	**Width (R)**	**I (R)**	**Width (C)**	**I (C)**
Input_type	26	8	24	8
Output_type	26	8	24	8
Kernel_type	26	1	24	1
Sqr_type	11	11	12	12
Sum_type	11	11	12	12
Sqroot_type	26	6	20	6
Norm_type	26	0	22	0
Type, N=8&7	**Width (R)**	**I (R)**	**Width (C)**	**I (C)**
Input_type	20	5	24	8
Output_type	20	5	24	8
Kernel_type	20	1	24	1
Sqr_type	8	8	9	9
Sum_type	8	8	9	9
Sqroot_type	22	5	22	5
Norm_type	24	0	24	0

Table B2: The bitwidths and the radix-point placements (I-parameter) used in all the fixed-point transformations. R = Real fixed-point, C = Complex fixed-point. See figure 16 and the example PE code in section 3.2.2.

Type, N=6&5	**Width (R)**	**I (R)**	**Width (C)**	**I (C)**
Input_type	20	5	22	6
Output_type	20	5	22	6
Kernel_type	20	1	22	1
Sqr_type	7	7	9	9
Sum_type	7	7	9	9
Sqroot_type	18	7	22	5
Norm_type	23	0	22	0
Type, N=4&3	**Width (R)**	**I (R)**	**Width (C)**	**I (C)**
Input_type	18	4	20	5
Output_type	18	4	20	5
Kernel_type	18	1	20	1
Sqr_type	5	5	8	8
Sum_type	5	5	8	8
Sqroot_type	18	3	20	5
Norm_type	20	0	20	0
Type, N=2	**Width (R)**	**I (R)**	**Width (C)**	**I (C)**
Input_type	16	3	20	5
Output_type	16	3	20	5
Kernel_type	16	1	20	1
Sqr_type	3	3	8	8
Sum_type	3	3	8	8
Sqroot_type	14	3	20	5
Norm_type	14	0	20	0

C Accuracy Results for PE operations

Table C1: The accuracy results for the fixed-point real number PE implementations' kernel generation calculations.

Operation output	PE, Sqrt&Div	PE, Inv. Sqrt	PE, Reference
In1	1.0000000000	1.0000000000	1.0000000000
In2	2.0000000000	2.0000000000	2.0000000000
Inverse Norm	0.4472122192	0.4467773437	0.4472135954
Kernel1	0.4472045898	0.4467773437	0.4472135954
Kernel2	0.8944396972	0.8935546875	0.8944271909
Out1	2.2360839843	2.2338867187	2.2360679774
Out2	0.0000000000	0.0000000000	0.0000000000

Table C2: The accuracy results for the fixed-point complex number PE implementations' kernel generation calculations.

Operation output	PE, Sqrt&Div	PE, Inv. Sqrt	PE, Reference
In1 Re	1.0000000000000	1.000000000000	1.000000000000
In1 Im	2.0000000000000	2.000000000000	2.000000000000
In2 Re	2.0000000000000	2.000000000000	2.000000000000
In2 Im	3.0000000000000	3.000000000000	3.000000000000
Inverse Norm	0.2357025146484	0.235534667968	0.235702260395
Kernel1 Re	0.2357025146484	0.235534667968	0.235702260395
Kernel1 Im	0.4714050292968	0.471069335937	0.471404520791
Kernel2 Re	0.4714050292968	0.471069335937	0.471404520791
Kernel2 Im	0.7071075439453	0.706604003906	0.707106781186
Out1 Re	4.2426452636718	4.239624023437	4.242640687119
Out1 Im	0.0000000000000	0.000000000000	0.000000000000
Out2 Re	0.0000000000000	0.000000000000	0.000000000000
Out2 Im	0.0000000000000	0.000000000000	0.000000000000

Table C3: The accuracy results for the fixed-point and floating-point real number PE implementations' kernel generation calculations when using the PWL inverse square root funtion.

Operation output	**Fixed-point**	**Floating-point**	**Reference**
In1	1.0000000000	1.0000000000	1.0000000000
In2	2.0000000000	2.0000000000	2.0000000000
Inverse Norm	0.4467773437	0.4468140602	0.4472135954
Kernel1	0.4467773437	0.4468140602	0.4472135954
Kernel2	0.8935546875	0.8936281204	0.8944271909
Out1	2.2338867187	2.2340707778	2.2360679774
Out2	0.0000000000	0.0000000000	0.0000000000