Adaptive Beamforming Using the Recursive Least Squares Algorithm on an FPGA

Andreas Bertheussen

Master of Science in Electronics

Submission date:	June 2015
Supervisor:	Bjørn B. Larsen, IET
Co-supervisor:	Erik Narverud, Kongsberg Defence Systems
	Lars Lundheim, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Assignment text

This text details a Masters thesis proposal put forward by Kongsberg Defence Systems (KDS) for the spring semester of 2015. The task is intended for students in their final year at the Institute of Electronics and Telecommunication at NTNU. Required skills are digital signal processing and preferably VHDL along with Matlab programming.

When receiving radio signals it is sometimes desirable to cancel out signals from certain directions in space while maintaining the rest of the spectrum. One could achieve this by making use of an array of antenna elements. The purpose of this system is to reduce the mean power of the signals in the directions where the sources that are to be canceled are located, while maintaining the rest of the spectrum. This can be done by exploitation of destructive interference demanding an array that is designed so that the incoming signals of the different antennas will be out of phase with each other. In order to achieve this destructive interference the signals have to be scaled and phase shifted by weights, and then added together. Both the signals and the weights will be complex numbers. The algorithm should find a set of weights that corresponds to a minimization of the mean power of the signals in the directions of the sources that are to be canceled.

It is intended that the task is limited to developing and implementing one or several variations of a recursive least squares null-steering algorithm on an FPGA and comparing the performance and complexity of these versus the analytical approach. It is expected that a Matlab simulation is developed first and used to assess the suggested design. The results from the simulation may then be used as a basis for implementation and test on a real FPGA, and the resulting performance compared to the performance estimates. A reference data set will be provided for the performance assessment.

Summary

This thesis describes the design and implementation of a five-channel beamformer using a Space-Time Adaptive Processing (STAP) filter with Recursive Least Squares (RLS) as the adaptive algorithm. The objective of the algorithm is to compute of a set of filter weights for a STAP filter, such that the channels are filtered and combined into a signal with minimized power. Two test signal sets containing a high-powered jammer signal and a noise floor are used for performance evaluation. Three goals are set for this thesis; comparison of RLS to Sample Matrix Inversion (SMI) algorithm when used in a beamformer, comparison of various architectures which implement RLS, and the implementation and test of one of the architectures for a Xilinx Virtex 6 XC6VLX240T-1 Field-Programmable Gate Array (FPGA)

Simulations comparing RLS to SMI show that a beamformer using RLS performs the same as a beamformer using SMI for 3-5 antennas (channels) and 1-4 temporal taps in the STAP filter.

Litterature review shows that conventional RLS is unsuitable for FPGA implementation due to numerical instability. Comparison of IQRD-RLS, FQRD-RLS and MCFQRD-RLS architectures which are claimed to be stable RLS variants, shows that IQRD-RLS is the least computationally expensive of the algorithms.

IQRD-RLS is implemented using Givens rotations in a systolic array architecture. Floating point, fixed point and CORDIC-based Givens rotation algorithms are compared with regard to speed and area, and floating point is chosen. Hardware simulations reveal that the filter weights returned by IQRD-RLS exhibit a drift, and is *not* stable in finite-precision arithmetic. The main cause is accumulated quantization error from the forgetting factor and its inverse ($\lambda^{\pm 1/2}$).

The IQRD-RLS systolic array is reduced to a (stable) QRD-RLS systolic array, approximately halving the number of systolic array nodes. Filter weights are not computed directly by QRD-RLS, and are instead recovered from the QRD-RLS least squares filtering error output by the method of weight flushing.

Results show that the QRD-RLS systolic array using 14 mantissa bits is sufficient as it performs equivalently to conventional RLS using double precision (53 mantissa bits). If only 11 mantissa bits are used, the output power increases by 3.3 dB. The final design can operate at sample rates from 19.4 MHz to 24.6 MHz, for a mantissa precision range of 14 to 11 bits. At this rate, the QRD-RLS systolic array can converge and output filter weights in 5.3 μs, significantly faster than the target of 100 μs. It is found that the current design has fully utilized its speed potential/limit due to the recursive nature of the algorithm. Processing of signals at the desired rate of 125 MHz would require changes to the algorithm itself. The implementation size is such that a 5-channel QRD-RLS array with one tap can fit on the FPGA. Channel-interleaving is proposed as a method to reduce system size, at the expense of slower operation.

All hardware is designed, simulated and tested using Simulink together with Xilinx System Generator and its co-simulation and hardware-in-the-loop features.

Sammendrag

Denne oppgaven beskriver design og implementasjon av en fem-kanals stråleformer som benytter et rom- og tids-adaptivt filter (STAP) med rekursiv minste kvadraters metode (RLS) som adaptiv algoritme. Målet til algoritmen er å finne filtervektene til STAP-filteret slik at kanalene filtreres og kombineres til ett signal med minimert effekt. To sett med testsignaler som inneholder et kraftig jammesignal og støygulv brukes for å vurdere ytelsen. Tre mål er satt for oppgaven; sammenlikning av RLS-algoritmen med direkte invertering av korrelasjonsmatriseestimat (SMI), sammenlikning av forskjellige arkitekturer for implementasjon av RLS, og implementasjon og test av én av arkitekturene på en Xilinx Virtex 6 XC6VLX240T-1 Field-Programmable Gate Array (FPGA).

Simuleringer som sammenlikner RLS med SMI viser at en stråleformer som bruker RLS har tilsvarende ytelse som SMI når det brukes 3-5 antenner (kanaler) og 1-4 tapper (tidsforsinkelser) i STAP-filteret.

Med grunnlag i litteratur vurderes konvensjonell RLS som uegnet for implementasjon på FPGA på grunn av numerisk ustabilitet. Sammenlikning av arkitekturene IQRD-RLS, FQRD-RLS og MCFQRD-RLS, som påstås å være stabile varianter av RLS, viser at IQRD-RLS krever minst regnekraft.

IQRD-RLS implementeres ved bruk av Givens rotasjoner i en "systolisk array"-arkitektur. Flyttall, fastkomma og CORDIC-baserte metoder for Givens rotasjon sammenliknes med hensyn til implementasjonens hastighet og størrelse, og flyttall velges. Simulering av maskinvare viser at filtervektløsningene fra IQRD-RLS-arrayet drifter vekk fra korrekt løsning og at IQRD-RLS derfor er ustabilt. Hovedårsaken er akkumulert kvantiseringsfeil fra glemmefaktoren og dens inverse ($\lambda^{\pm 1/2}$).

IQRD-RLS-arrayet forenkles til et (stabilt) QRD-RLS systolisk array, noe som halverer antallet noder brukt i arrayet. Filtervekter beregnes ikke direkte av QRD-RLS, men må i stedet hentes ut fra QRD-RLS arrayet sin impulsrespons.

Resultater viser at 14 bit mantisse er tilstrekkelig for at det QRD-RLS systoliske arrayet har en ytelse tilsvarende konvensjonell RLS med dobbel presisjons aritmetikk (53 bits mantisse). Reduksjon til 11 bit mantisse øker uteffekten med 3.3 dB. Maskinvareimplementasjonen kan operere opp til 19.4 MHz til 24.6 MHz avhengig av mantissepresisjon fra 14 til 11 bit. Ved denne raten kan systemet konvergere og finne filtervektene innen 5.3 μs, som er betydelig raskere enn målsetningen på 100 μs. Det er også funnet at den aktuelle implementasjonen har nådd hastighetsbegrensningen forårsaket av QRD-RLS-algoritmens rekursive definisjon. For å håndtere signaler ved ønsket rate på 125 MHz må selve QRD-RLS-algoritmen modifiseres. Implementasjonens størrelse er slik at et QRD-RLS array med 5 kanaler og en tap får plass på aktuell FPGA. Kanal-interleaving er foreslått som metode som vil redusere størrelsen, på bekostning av redusert hastighet.

All maskinvare er designet, simulert og testet ved bruk av Simulink med Xilinx System Generator og dets funksjonalitet for ko-simulering og simulering med FPGA i sløyfa.

Preface

This thesis is submitted for the degree of Master of Science (MSc) at the Norwegian University of Science and Technology (NTNU). This work has been carried out during the spring semester of 2015 with supervision and guidance by Assoc. Professor Bjørn B. Larsen and Professor Lars M. Lundheim (NTNU) as well as Erik Narverud, systems engineer at Kongsberg Defence Systems (KDS). KDS defined the assignment and supplied the hardware needed for implementation and test.

When I started on this thesis, I had no experience with beamformers or adaptive filters. Working out the implementation details from math to hardware has been challenging and very rewarding. I'd like to thank my supervisors Bjørn, Lars and Erik for their guidance and advice. In particular, I'd like to thank Lars for his constructive and detailed feedback on matters relating to signal processing theory.

Trondheim, Norway, June 2015

Andreas Bertheussen

Contents

Figures

Tables

Algorithms

Abbreviations

FPGA	Field-Programmable Gate Array
FIR	Finite Impulse Response
KDS	Kongsberg Defence Systems
RLS	Recursive Least Squares
SMI	Sample Matrix Inversion
LUT	Look-Up Table
STAP	Space-Time Adaptive Processing
MMSE	Minimum Mean Square Error
DMI	Direct Matrix Inversion
CORDIC	Coordinate Rotation Digital Computer
LMS	Least Mean Squares
GNSS	Global Navigation Satellite System
WE	Weight Extraction
SRF	Square Root Free

Notation and units

Notation that is extensively used is summarized here.

i	Imaginary unit or time (iteration) index
$x[i]$, $\mathbf{A}[i]$	Discrete-time signal (1D or vector/matrix)
x^*	Complex conjugate of x
$\mathbf{I}_p$	Identity matrix with number of columns/rows given by p
$\mathbf{A}^H$	Hermitian transpose of matrix or vector, $= (\boldsymbol{A}^*)^T$
$\mathbf{A}^{-T}$	Transpose of matrix inverse, $= (A^{-1})^T$
c, s	$c = \cos(\theta)$, $s = \sin(\theta)$, rotation factors for Givens rotation
M	Number of channels excluding reference channel
N	Number of filter taps per channel
λ	Forgetting factor for conventional RLS algorithm
K	Window length for correlation estimate in SMI algorithm
δ	Used for RLS initialization or diagonal loading term in SMI
$\arctan(y, x)$	Inverse tangent with two arguments, covering all quadrants. Equal to $\arctan(y/x)$ for positive y and x. Also known as `atan2()`.

Decibel quantities are relative to some reference. The reference is denoted in the unit. For instance, the quantity P_{out} [dB$_{noise}$] implies the calculation $10\log(P_{out}/P_{noise})$.

1 Introduction

1.1 Motivation

Many types of passive disturbances or interference can impair the function of a radio system, such as shadowing, multipath effects, variable propagation speeds and others. Disturbances can also be caused by radio transmitters, accidentally or intentionally. Intentional disturbances are commonly called *jamming*. Jammers may use one or several strategies to disturb the target radio system, such as single tone, multi tone, narrowband or broadband signals and frequency sweeps. A jammer could emit a signal which overpowers and spoofs the original signal, or a signal that will exceed the dynamic range of the receiver. A jammer could also replay the original signals, creating artificial multipath effects.

For a radio receiver that must function in an environment with jammers, it is desirable to suppress disturbances originating from certain directions or disturbances in certain frequency bands. This can be achieved using an antenna array where the signal from each individual antenna is filtered and combined so that the interference is canceled through destructive interference. The filtering operation for each channel can be a single complex multiplication, or a more general complex FIR (Finite Impulse Response) filter. The filter coefficients may be computed based on prior knowledge of the direction and/or frequency band of the interference and desired signal, or they may be adaptively computed by an algorithm, based on the received signals. The process of finding the appropriate weights is called *beamforming*, since the choice of weights controls the antenna array's sensitivity in both direction and frequency.

The assignment was proposed by KDS (Kongsberg Defence Systems), a Norwegian supplier of defence-related systems. KDS has several radio communication products, and are naturally interested in anti-jamming measures. KDS has supplied test signal sets and a Xilinx Virtex 6 Field-Programmable Gate Array (FPGA) development board for this thesis.

1.2 Goals

The goals set for this thesis are:

- Compare performance Recursive Least Squares (RLS) with the reference algorithm Sample Matrix Inversion (SMI) (the *analytical approach*).
- Compare various architectures for implementing RLS on the FPGA.
- Implement and test a suitable architecture on the FPGA.

1.3 Tools

Mathworks MATLAB R2012b is used throughout to write and run scripts that implements various algorithms. A MATLAB extension called Simulink is used to create and simulate block diagram models.

The target hardware for the implementation is a Xilinx Virtex 6 ML605 development board. The board contains many peripherals, but only the Virtex 6 XC6VLX240T-1 FPGA and the included debugging connectivity is used for this thesis.

The Virtex 6 series FPGAs contain an array of (re)configurable logic blocks (CLBs) connected to a programmable switch matrix for signal routing. Each CLB contains two *slices*, and each slice contains four 6-input look-up tables (LUTs), eight register flip flops (Reg.), multiplexers and carry logic. Some slices contain additional memory and shift register functionality. There is also a special type of slice named *DSP48E1* (DSP48 for short) which implements multiply-accumulate functionality. The XC6VLX240T-1 device has a total capacity of 150720 LUTs (37680 slices), as well as 768 DSP48 slices. These are the basic resources of the FPGA. Functionality implemented using LUTs is referred to as a *fabric* implementation.

The FPGA configuration is typically derived from hardware description languages like VHDL or Verilog. For this thesis, an extension to Simulink called *System Generator* is used. This extension is developed by Xilinx and includes a hardware block library containing functions such as adders, multipliers, dividers and other functions which can be synthesized for Xilinx FPGAs. When a Simulink model is constructed using these blocks, it can be processed using System Generator resulting in a synthesizable VHDL description of the model. The conventional Xilinx ISE toolchain can then used to process the VHDL, resulting in a configuration which can be loaded into the FPGA. In this way, System Generator works as a translator from the Simulink block diagram format to a more conventional VHDL description of a system.

System Generator is also able to accurately simulate the hardware blocks within the Simulink environment by using the Xilinx ISim HDL simulator in the background. System Generator can also be used do a hardware-in-loop simulation. This means that System Generator takes the block diagram model and generates a FPGA configuration which is loaded onto the FPGA. When the model is simulated in Simulink, the signals that are entering/leaving the hardware blocks is routed to/from the physical FPGA across its debug interface. This method is used to check that the implementation matches simulation results.

2 Theory

This chapter holds theory that will be referred to and/or applied in later chapters. The theory behind least squares filtering is described, followed by the Sample Matrix Inversion and Recursive Least Squares algorithms which can be used to solve the least squares problem.

The Givens rotation and CORDIC algorithms are explained as they can be used to realize QR-decomposition. QR-decomposition is explained since it is the core of the QRD-RLS and IQRD-RLS algorithms which are practical implementations of the RLS algorithm.

2.1 The Space-Time Adaptive Processing (STAP) filter

STAP [1,ch12] is an extension to the M-channel beamformer which introduces tapped delay line (FIR) filters of order N instead of a simple complex gain factor for each channel. The STAP filter linearly combines multiple channels into a single channel, and can adjust the phase and frequency response (gain) by modifying the weights $w_{m,n}$. The space-time filtering structure is shown in figure 2.1.

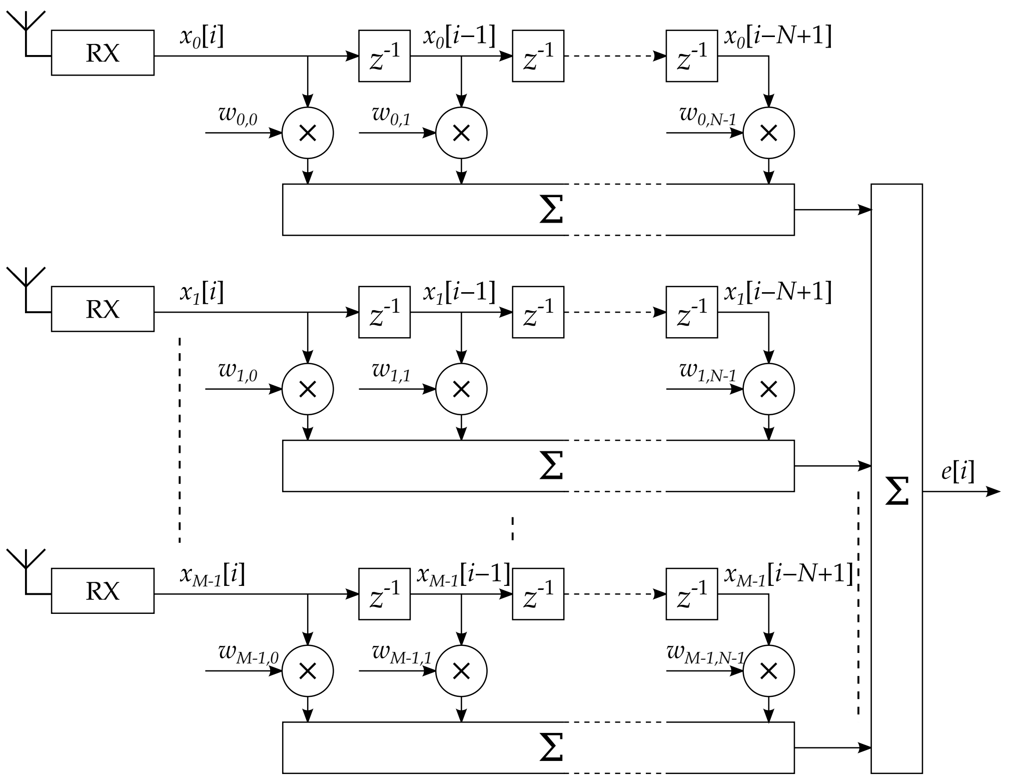


Figure 2.1 A STAP filter with M channels and N taps.

Mathematically, it may be described by:

$$e[i] = \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} w^*_{m,n} x_m[i-n] \qquad \text{or more simply} \qquad e[i] = \boldsymbol{w}^H \boldsymbol{x}[i]$$

Here, we have chosen to define the $MN{\times}1$ vector $\boldsymbol{x}[i]$ as a snapshot of the signal available from the delay elements in the STAP structure at time i:

$$\boldsymbol{x}[i] = \begin{bmatrix} \begin{bmatrix} x_0[i] & x_1[i] & \cdots & x_{M-1}[i] \end{bmatrix}^T \\ \begin{bmatrix} x_0[i-1] & x_1[i-1] & \cdots & x_{M-1}[i-1] \end{bmatrix}^T \\ \vdots \\ \begin{bmatrix} x_0[i-N+1] & x_1[i-N+1] & \cdots & x_{M-1}[i-N+1] \end{bmatrix}^T \end{bmatrix} \tag{2.1}$$

Similarly, the filter weights are also formed into a $MN \times 1$ vector:

$$\boldsymbol{w} = \begin{bmatrix} \begin{bmatrix} w_{0,0} & w_{1,0} & \cdots & w_{M-1,0} \end{bmatrix}^T \\ \begin{bmatrix} w_{0,1} & w_{1,1} & \cdots & w_{M-1,1} \end{bmatrix}^T \\ \vdots \\ \begin{bmatrix} w_{0,N-1} & w_{1,N-1} & \cdots & w_{M-1,N-1} \end{bmatrix}^T \end{bmatrix}$$

In STAP the filter weights are adapted by an algorithm based only on the input $\boldsymbol{x}$. In that case the weight vector changes over time and is denoted $\boldsymbol{w}[i]$.

2.2 The Minimum Mean Square Error (MMSE) criterion

Consider the general filter $y[i] = \boldsymbol{w}^H\boldsymbol{x}[i]$. This filter can represent a FIR filter when elements of $\boldsymbol{x}$ are samples of the same signal at different times. It can also represent a linear combiner if elements of $\boldsymbol{x}$ are taken from different signals. It can also represent a combination of the two, like the STAP structure described earlier.

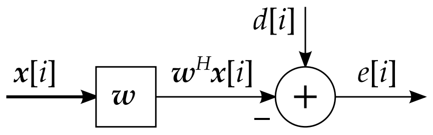


Figure 2.2 MMSE block diagram

A new signal $d[i]$ is introduced as the *desired* signal, and the difference, or *error*, $e[i] = d[i] - \boldsymbol{w}^H\boldsymbol{x}[i]$ is computed as illustrated in figure 2.2. The MMSE criterion is used to find the required properties of the filter weight values which minimize the squared error $E[|e[i]|^2] = E[e[i]e^*[i]]$.

$$E[e[i]e^*[i]] = E\left[d[i]d^*[i] - 2\boldsymbol{w}^H\boldsymbol{x}[i]d^*[i] + \boldsymbol{w}^H\boldsymbol{x}[i]\boldsymbol{x}^H[i]\boldsymbol{w}\right]$$

$$\begin{aligned} E[e[i]e^*[i]] &= E\left[d[i]d^*[i]\right] - 2\boldsymbol{w}^H E\left[\boldsymbol{x}[i]d^*[i]\right] + \boldsymbol{w}^H E\left[\boldsymbol{x}[i]\boldsymbol{x}^H[i]\right]\boldsymbol{w} \\ &= \sigma_d^2 - 2\boldsymbol{w}^H\boldsymbol{r}_{xd} + \boldsymbol{w}^H\boldsymbol{R}_{xx}\,\boldsymbol{w} \end{aligned}$$

Stationarity is assumed. The minimum of $|e[i]|^2$ is found by setting its gradient equal to zero, and solving for $\boldsymbol{w}$.

$$\nabla_{\boldsymbol{w}} E[e[i]e^*[i]] = 0 = -2\boldsymbol{r}_{xd} + 2\boldsymbol{R}_{xx}\,\boldsymbol{w}$$

Which is fulfilled when $\boldsymbol{w} = \boldsymbol{w}_{opt}$ according to:

$$\boldsymbol{R}_{xx}\,\boldsymbol{w}_{opt} = \boldsymbol{r}_{xd} \qquad \Leftrightarrow \qquad \boldsymbol{w}_{opt} = \boldsymbol{R}_{xx}^{-1}\,\boldsymbol{r}_{xd} \tag{2.2}$$

Equation (2.2) is known as the Wiener-Hopf equation.

2.3 Algorithms for solving MMSE problems

The expressions in **equation (2.2)** depend on the statistical properties R_{xx} and r_{xd} which describe the data. The Sample Matrix Inversion and Recursive Least Squares algorithms presented in this section are two approaches to estimating these statistics, resulting in the optimal filter weights.

2.3.1 The Sample Matrix Inversion (SMI) algorithm

The SMI algorithm uses the simple estimators of **equation (2.3)** to estimate R_{xx} and r_{xd}. The estimates are then used in place of R_{xx} and r_{xd} in **equation (2.2)** to directly compute $\boldsymbol{w}_{opt}$. This is referred to as Sample Matrix Inversion and sometimes also Direct Matrix Inversion (DMI).

$$\hat{\boldsymbol{R}}_{xx}[i] = \frac{1}{K}\sum_{j=i-K+1}^{i} \boldsymbol{x}[j]\boldsymbol{x}^H[j] \qquad \hat{\boldsymbol{r}}_{xd}[i] = \frac{1}{K}\sum_{j=i-K+1}^{i} \boldsymbol{x}[j]d^*[j] \tag{2.3}$$

Inversion of $\hat{\boldsymbol{R}}_{xx}[i]$ may be problematic since the estimators do not guarantee non-singularity. One commonly proposed technique to avoid singular matrices is [1][2,ch7.3][3] termed *diagonal loading*. It is done by adding a small term $\delta\boldsymbol{I}$ to the estimate $\hat{\boldsymbol{R}}_{xx}$ before inverting it.

2.3.2 The Recursive Least Squares (RLS) algorithm

The RLS algorithm [1,p273][4,p209][5] applies MMSE but starts with a slightly different formulation, so that the solution can be expressed on a recursive form. This means the solution is found in terms of modifications to the previous solution. Let a new scalar signal $f[i]$ be defined as:

$$f[i] = \sum_{j=0}^{i} \lambda^{i-j}\,e^2[j] \tag{2.4}$$

$e[j]$ is the error signal and $\lambda \in (0,1)$ is a "forgetting factor" which exponentially reduces the importance of previous squared errors. A value of λ very close to 1 means that the algorithm responds slower to change; it has longer memory. A small value has the opposite effect. By minimizing $f[i]$ with respect to $\boldsymbol{w}$, it can be shown [4,ch5.2] that the solution must satisfy a modified Wiener-Hopf equation:

$$\boldsymbol{R}'_{xx}[i]\,\boldsymbol{w}'[i] = \boldsymbol{r}'_{xd}[i] \qquad \Leftrightarrow \qquad \boldsymbol{w}'[i] = (\boldsymbol{R}'_{xx}[i])^{-1}\boldsymbol{r}'_{xd}[i]$$

where $\boldsymbol{R}'_{xx}[i]$ and $\boldsymbol{r}'_{xd}[i]$ are are the sample autocorrelation of $\boldsymbol{x}[i]$ and sample cross correlation between $\boldsymbol{x}[i]$ and $\boldsymbol{d}[i]$, on a form that is exponentially weighted similarly to the error signal of **equation (2.4)**:

$$\boldsymbol{R}'_{xx}[i] = \sum_{j=0}^{i} \lambda^{i-j}\boldsymbol{x}[j]\boldsymbol{x}^H[j] \qquad \boldsymbol{r}'_{xd}[i] = \sum_{j=0}^{i} \lambda^{i-j}\boldsymbol{x}[j]d^H[j]$$

It is possible to write $\boldsymbol{R}'_{xx}[i]$ and $\boldsymbol{r}'_{xd}[i]$ in a recursive form. The term corresponding to "now" index i can be taken out of the summation:

$$\begin{aligned}\boldsymbol{R}'_{xx}[i] &= \sum_{j=0}^{i} \lambda^{n-j}\boldsymbol{x}[j]\boldsymbol{x}^H[j] \\ &= \lambda\sum_{j=0}^{i-1} \lambda^{(n-1)-j}\boldsymbol{x}[j]\boldsymbol{x}^H[j] + \boldsymbol{x}[i]\boldsymbol{x}^H[i] \\ &= \lambda\boldsymbol{R}'_{xx}[i-1] + \boldsymbol{x}[i]\boldsymbol{x}^H[i]\end{aligned}$$

The same can be done for $\boldsymbol{r}'_{xd}[i]$:

$$\boldsymbol{r}'_{xd}[i] = \sum_{j=0}^{i} \lambda^{i-j}\boldsymbol{x}[j]d^H[j] = \lambda\boldsymbol{r}'_{xd}[i-1] + \boldsymbol{x}[i]d^H[i]$$

We have here found an expression describing how $\boldsymbol{R}'_{xx}[i-1]$ and $\boldsymbol{r}'_{xd}[i-1]$ (previous timestep correlation matrices) can be updated to the current time step. To avoid computing the inverse $(\boldsymbol{R}'_{xx}[i])^{-1}$, we apply the Woodbury matrix identity. To simplify notation, we define $\boldsymbol{P}[i] = (\boldsymbol{R}'_{xx}[i])^{-1}$ and $\boldsymbol{z}[i] = \boldsymbol{r}'_{xd}[i]$.

$$\begin{aligned}(\boldsymbol{R}'_{xx}[i])^{-1} = \boldsymbol{P}[i] &= \frac{1}{\lambda}\boldsymbol{P}[i-1] - \frac{\frac{1}{\lambda}\boldsymbol{P}[i-1]\boldsymbol{x}[i]\boldsymbol{x}^H[i]\frac{1}{\lambda}\boldsymbol{P}[i-1]}{1 + \boldsymbol{x}^H[i]\frac{1}{\lambda}\boldsymbol{P}[i-1]\boldsymbol{x}[i]} \\ &= \frac{1}{\lambda}\boldsymbol{P}[i-1] - \frac{1}{\lambda}\boldsymbol{K}[i]\boldsymbol{x}^H[i]\boldsymbol{P}[i-1]\end{aligned}$$

Where we have defined

$$\boldsymbol{K}[i] = \frac{\boldsymbol{P}[i-1]\boldsymbol{x}[i]}{\lambda + \boldsymbol{x}^H[i]\boldsymbol{P}[i-1]\boldsymbol{x}[i]}$$

The denominator here is a scalar making the expression somewhat simpler to compute. By multiplying up the denomitator, we can show that:

$$\begin{aligned}
\boldsymbol{K}[i] + \frac{1}{\lambda}\boldsymbol{K}[i]\boldsymbol{x}^H[i]\boldsymbol{P}[i-1]\boldsymbol{x}[i] &= \frac{1}{\lambda}\boldsymbol{P}[i-1]\boldsymbol{x}[i] \\
\boldsymbol{K}[i] &= \left(\frac{1}{\lambda}\boldsymbol{P}[i-1] - \frac{1}{\lambda}\boldsymbol{K}[i]\boldsymbol{x}^H[i]\boldsymbol{P}[i-1]\right)\boldsymbol{x}[i] \\
\boldsymbol{K}[i] &= \boldsymbol{P}[i]\boldsymbol{x}[i]
\end{aligned}$$

By inserting previous results in the modified Wiener-Hopf equation, we can find a recursive expression for the filter weights which does not require a matrix inversion:

$$\begin{aligned}
\boldsymbol{w}'[i] &= (\boldsymbol{R}'_{xx}[i])^{-1}\boldsymbol{r}'_{xd}[i] \\
&= \boldsymbol{P}[i]\boldsymbol{z}[i] \\
&= \boldsymbol{P}[i]\left(\lambda\boldsymbol{z}[i-1] + \boldsymbol{x}[i]d[i]\right) \\
&= \lambda\boldsymbol{P}[i]\boldsymbol{z}[i-1] + \boldsymbol{P}[i]\boldsymbol{x}[i]d[i] \\
&= \lambda\left(\frac{1}{\lambda}\boldsymbol{P}[i-1] - \frac{1}{\lambda}\boldsymbol{K}[i]\boldsymbol{x}^H[i]\boldsymbol{P}[i-1]\right)\boldsymbol{z}[i-1] + \boldsymbol{P}[i]\boldsymbol{x}[i]d[i] \\
&= \underbrace{\boldsymbol{P}[i-1]\boldsymbol{z}[i-1]}_{\boldsymbol{w}'[i-1]} - \boldsymbol{K}[i]\boldsymbol{x}^H[i]\underbrace{\boldsymbol{P}[i-1]\boldsymbol{z}[i-1]}_{\boldsymbol{w}'[i-1]} + \underbrace{\boldsymbol{P}[i]\boldsymbol{x}[i]}_{\boldsymbol{K}[i]}d[i] \\
&= \boldsymbol{w}'[i-1] - \boldsymbol{K}[i]\boldsymbol{x}^H[i]\boldsymbol{w}'[i-1] + \boldsymbol{K}[i]d[i] \\
&= \boldsymbol{w}'[i-1] + \boldsymbol{K}[i]\left(d[i] - \boldsymbol{w}'^H[i-1]\boldsymbol{x}[i]\right) \\
&= \boldsymbol{w}'[i-1] + \boldsymbol{K}[i]\epsilon[i]
\end{aligned}$$

Note that the scalar signal $\epsilon[i]$ represents the error output from filtering the current input signal with the previous filter. It is termed the *a priori* error signal. The a priori error is multiplied with the *gain matrix* $\boldsymbol{K}[i]$ in order to modify the filter weights. The conventional RLS algorithm is summarized in **algorithm 2.1** [5].

```
$\boldsymbol{P}[-1] = \delta^{-1}\boldsymbol{I}$
$\boldsymbol{w}'[-1] = \begin{bmatrix}1 & 0 & \cdots & 0\end{bmatrix}^T$
for each timestep $i \geq 0$ do
    $\boldsymbol{K}[i] = \frac{\boldsymbol{P}[i-1]\boldsymbol{x}[i]}{\lambda + \boldsymbol{x}^H[i]\boldsymbol{P}[i-1]\boldsymbol{x}[i]}$
    $\boldsymbol{w}'[i] = \boldsymbol{w}'[i-1] + \boldsymbol{K}[i]\left(d[i] - \boldsymbol{w}'^H[i-1]\boldsymbol{x}[i]\right)$
    $\boldsymbol{P}[i] = \mathrm{Tri}\left(\frac{1}{\lambda}\left(\boldsymbol{P}[i-1] - \boldsymbol{K}[i]\boldsymbol{x}^H[i]\boldsymbol{P}[i-1]\right)\right)$
    $e[i] = d[i] - \boldsymbol{x}^H[i]\boldsymbol{w}'[i]$ {apply filter weights to signal}
end for
```

Algorithm 2.1 The conventional RLS algorithm

The initialization of the $\boldsymbol{P}$ diagonal with δ^{-1} is performed to bring it close to its final value when the signal power range is known in advance. Tri() signifies that only one triangular half of the matrix needs to be calculated, while the other triangualar section is filled in with the complex conjugate due to the hermitian symmetry of the matrix $\boldsymbol{P}$ [5,p510].

2.4 Givens rotation

A Givens rotation [6] is a matrix transformation (named after Wallace Givens) which rotates two rows of a matrix. It can be used to transform matrices to an upper triangular or lower triangular form as part of a QR-decomposition. For an example of a Givens rotation, consider the illustration in figure 2.3 (a). Here, the vector $(a, b) = a\boldsymbol{e}_1 + b\boldsymbol{e}_2$ is rotated by θ, creating a new vector (x, y).

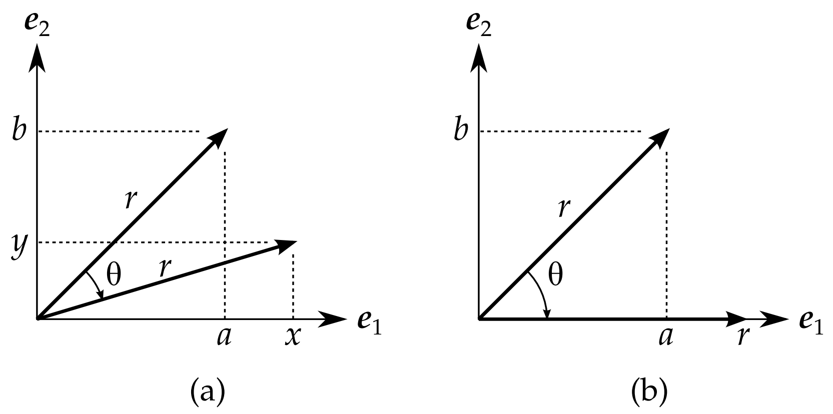


Figure 2.3 Examples of givens rotation. On the left is the general case, and on the right is the special case when the rotation is used to annihilate the $\boldsymbol{e}_2$ dimension.

Mathematically, this clockwise[1] rotation can be described by left-multiplication with a rotation matrix as follows:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin^*(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \boldsymbol{G}_\theta \begin{bmatrix} a \\ b \end{bmatrix}$$

The givens rotation may be applied to n-th and m-th row in any matrix of size p, by defining the rotation matrix as the identity matrix I_p (of size p), with elements $G_{\theta n,n} = c = \cos(\theta)$, $G_{\theta m,m} = s = \sin(\theta)$, $G_{\theta m,n} = -s^* = -\sin^*(\theta)$ and $G_{\theta m,m} = c = \cos(\theta)$: (zero values are omitted)

$$\boldsymbol{G}_\theta = \begin{bmatrix} \boldsymbol{I}_n & & & & \\ & c & & s & \\ & & \boldsymbol{I}_{m-n-1} & & \\ & -s^* & & c & \\ & & & & \boldsymbol{I}_{p-m-1} \end{bmatrix}$$

The transform will only affect the m-th and n-th row, and will transform each column individually, for example if:

[1] In some cases the Givens rotation is defined for counterclockwise rotation, for which the sine terms are exchanged.

$$\begin{bmatrix} x & r \\ y & s \end{bmatrix} = \boldsymbol{G}_{\theta_0} \begin{bmatrix} a & f \\ b & g \end{bmatrix}, \text{ then we have } \begin{bmatrix} x \\ y \end{bmatrix} = \boldsymbol{G}_{\theta_0} \begin{bmatrix} a \\ b \end{bmatrix} \text{ and } \begin{bmatrix} r \\ s \end{bmatrix} = \boldsymbol{G}_{\theta_0} \begin{bmatrix} f \\ g \end{bmatrix}$$

2.5 CORDIC algorithm

The CORDIC algorithm [7] (Coordinate Rotation Digital Computer) is an implementation that can compute a real-valued Givens rotation. Like the Givens rotation, it can be used to perform a QR-decomposition. We start from the Givens rotation by angle θ, and rewrite it in terms of $\tan(\theta)$:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} \Rightarrow \begin{cases} x = & a\cos(\theta) + b\sin() \\ y = & -a\sin(\theta) + b\cos() \end{cases}$$

$$x = \cos(\theta)(a + b\tan(\theta)) \quad (2.5)$$

$$y = \cos(\theta)(b - a\tan(\theta)) \quad (2.6)$$

The CORDIC algorithm can implement Givens rotations by computing **equation (2.5)** and 2.6 iteratively, but restricting each iteration to angles $\theta = d_i\beta_i$ chosen so that $\beta_i = \arctan(2^{-i})$ and $d_i \in \{-1, 1\}$. When the $\cos(\theta)$ term is omitted, each iteration is reduced to an addition and bit shift. The first iteration uses $\tan(\beta_0) = 1, \beta_0 = 45°$. The next iteration uses $\tan(\beta_1) = 0.5, \beta_1 = 26.56...°$ and so on with smaller and smaller angles.

The key to CORDIC is how d_i is chosen. In the *vectoring mode*, d_i is chosen to be $\text{sign}(y_{i-1})$. Each iteration will therefore compute (x_i, y_i) where the y-component goes toward 0. If the input vector is (a, b), the result approaches $(K\sqrt{a^2 + b^2}, 0)$. Here, $K = \prod_{i=0}^{k-1} \cos(d_i\beta_i)$ which accounts for the $\cos(d_i\beta_i)$ factors in each iteration for a total of k iterations. By computing $\theta_i = \sum_{j=0}^{i} d_j\beta_j$, the total accumulated angle is also found. This way the original vector has been converted from rectangular to polar coordinates.

If d_i is instead chosen to be $\text{sign}(\theta - \theta_i)$ where θ is some desired rotation angle, the CORDIC is said to be in *rotating mode*. Each iteration will rotate the input vector (a, b) closer and closer to $K(x, y)$, being the original vector rotated by θ. If the desired rotation angle comes from a vectoring CORDIC algorithm, there is no need to compute θ. Instead, the sequence d_i can be directly transmitted from vectoring to rotating CORDIC as described by Gao [8]. Gao also shows one method for making CORDIC handle *coarse rotation*, i.e. cases where a or b are negative.

The scaling factor K is shown in **equation (2.7)**, for k number of stages. Its inverse $S = 1/K$ is used after the last stage to correct the result. Since $\cos(\cdot)$ is even, the choices of d_i do not affect the result. For large k, $K \approx 1.6467602...$ and $S = 1/K \approx 0.6072529....$ Typically, the number of stages is chosen to be one larger than the number of bits precision needed.

$$K = \prod_{i=0}^{k-1} \cos(d_i \arctan(2^{-i})) = \prod_{i=0}^{k-1} \frac{1}{\sqrt{1 + 2^{-2i}}} \qquad (2.7)$$

2.6 QR-decomposition

A QR-decomposition is a factorization of a matrix $\boldsymbol{A}$ into the product $\boldsymbol{QR}$ where $\boldsymbol{Q}$ is an orthogonal matrix and $\boldsymbol{R}$ is an upper triangular matrix. Since $\boldsymbol{Q}$ is orthogonal, we also have $\boldsymbol{Q}^H A = R$.

$\boldsymbol{Q}^H$ may be constructed by a series of Givens rotations or CORDIC operations. This concept is shown for a 3×3 matrix below. In each step the the two rightmost matrices are multiplied.

$$\begin{aligned}
R = \boldsymbol{Q}^H\boldsymbol{A} &= \boldsymbol{G}_3\boldsymbol{G}_2\boldsymbol{G}_1\boldsymbol{A} \\
&= \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_3 & s_3 \\ 0 & -s_3^* & c_3 \end{bmatrix}
\begin{bmatrix} c_2 & s_2 & 0 \\ -s_2^* & c_2 & 0 \\ 0 & 0 & 1 \end{bmatrix}
\begin{bmatrix} 1 & 0 & 0 \\ 0 & c_1 & s_1 \\ 0 & -s_1^* & c_1 \end{bmatrix}
\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_3 & s_3 \\ 0 & -s_3^* & c_3 \end{bmatrix}
\begin{bmatrix} c_2 & s_2 & 0 \\ -s_2^* & c_2 & 0 \\ 0 & 0 & 1 \end{bmatrix}
\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a'_{1,0} & a'_{1,1} & a'_{1,2} \\ \underline{0} & a'_{2,1} & a'_{2,2} \end{bmatrix} \\
&= \begin{bmatrix} 0 & 0 & 0 \\ 0 & c_3 & s_3 \\ 0 & -s_3^* & c_3 \end{bmatrix}
\begin{bmatrix} a'_{0,0} & a'_{0,1} & a'_{0,2} \\ \underline{0} & a''_{1,1} & a''_{1,2} \\ 0 & a'_{2,1} & a'_{2,2} \end{bmatrix} \\
&= \begin{bmatrix} a'_{0,0} & a'_{0,1} & a'_{0,2} \\ 0 & a'''_{1,1} & a'''_{1,2} \\ 0 & \underline{0} & a''_{2,2} \end{bmatrix}
\end{aligned}$$

In the first step, the Givens rotation $\boldsymbol{G}_1$ that affects all of row 2 and 3 is chosen based on vector $(a_{1,0}, a_{2,0})$ (highlighted in blue) so that element $a_{2,0}$ becomes zero (shown underlined in subsequent line). This same processing step repeats until all elements under the diagonal in the first column are zeroed, before continuing with the second column.

In general, each step takes a vector (a, b), rotates it to $(r, 0)$, and applies the same rotation angle to the other vectors occupying the same rows. The rotation angle is $\arctan(b, a)$, but explicit computation of $\arctan(\cdot)$ is in fact not required. By use of basic trigonometric relationships with the illustration of figure 2.3 (b), we find that:

$$r = \sqrt{a^2 + b^2} \qquad c = \frac{a}{\sqrt{a^2 + b^2}} \qquad s = \frac{b}{\sqrt{a^2 + b^2}}$$

Clearly, s and c can be found directly without any trigonometric function. An algorithm for Givens rotation for the case when a and b are complex is presented by Bindel et.al [6], and is shown in algorithm 2.2.

```
if $b = 0$ then
    $c = 1; s = 0; r = a$
else if $a = 0$ then
    $c = 0; s = \text{sign}(b^*); r = |b|$
else
    $c = |a|/\sqrt{|a|^2 + |b|^2}$
    $s = \text{sign}(a)b^*/\sqrt{|a|^2 + |b|^2}$
    $r = \text{sign}(a)\sqrt{|a|^2 + |b|^2}$
```

```
$r = cf + sg$
$s = -s^*f + cg$
```

Algorithm 2.2 Complex givens rotation. Left side rotates (a, b) to zero b, resulting in $(r, 0)$ and rotation parameters c and s. The right side uses c and s to rotate a different vector $(f, g) \rightarrow (r, s)$.

CORDIC can also be used for QR-decomposition, performing the same function as algorithm 2.2. First assume real matrix elements. In the first step of the earlier example, applying CORDIC in vectoring mode to $(a_{1,0}, a_{2,0})$ (highlighted in blue) gives the result $(a'_{1,0}, 0)$ and the angle $\theta_1 = \arctan(a_{2,0}, a_{1,0})$. CORDIC in rotating mode can then take this θ_1 angle and rotate the remaining vectors $(a_{1,1}, a_{2,1}) \rightarrow (a'_{1,1}, a'_{2,1})$ and $(a_{1,2}, a_{2,2}) \rightarrow (a'_{1,2}, a'_{2,2})$. This is repeated to bring all elements under the diagonal to zero, just like with algorithm 2.2. Handling of complex values is described in more detail by Rader, Gao and Maltsev [9, 10, 11].

2.7 QR-decomposition-based RLS (QRD-RLS)

The QRD-RLS algorithm implements the RLS recursion by use of a QR-decomposition. It is described by equation (2.8) [12,ch10]:

$$\begin{bmatrix} \boldsymbol{R}[i] & \boldsymbol{p}[i] & \boldsymbol{s}[i] \\ \boldsymbol{0}^T & \alpha[i] & \gamma[i] \end{bmatrix} = \boldsymbol{Q}[i] \begin{bmatrix} \lambda^{1/2}\boldsymbol{R}[i-1] & \lambda^{1/2}\boldsymbol{p}[i-1] & \boldsymbol{0} \\ \boldsymbol{x}^T[i] & d[i] & 1 \end{bmatrix} \tag{2.8}$$

A QR-decomposition is applied to the right hand side matrix, which consists of the square root exponentially weighted covariance matrix $\boldsymbol{R}$ and $\boldsymbol{p}$ from the previous iteration, a constant column vector $[0, \cdots, 0, 1]^T$, and the current data $\boldsymbol{x}[i]$ and $d[i]$ which the algorithm is to operate on.

The QR-decomposition generates $\alpha[i] = \gamma[i]\epsilon[i]$ and $\gamma[i]$ explicitly. $\epsilon[i]$ is the a priori estimation error in conventional RLS, which relates to the a posteriori error by the conversion factor $\gamma[i]^2$ according to $e[i] = \gamma^2[i]\epsilon[i]$. It follows [12,ch3] that the a posteriori error can be calculated from the QR-decomposition result by $e[i] = \gamma[i] \cdot \alpha[i]$.

The algorithm is summarized in algorithm 2.3.

2.8 Inverse QRD-RLS (IQRD-RLS)

The IQRD-RLS algorithm is an implementation of the RLS recursion, which also makes use of a QR-decomposition. It builds on the QRD-RLS algorithm by appending an

```
$\boldsymbol{R}[-1] = \delta^{1/2}\boldsymbol{I}$
for each timestep $i \geq 0$ do
    $\begin{bmatrix} \boldsymbol{R}[i] \\ \boldsymbol{0}^T \end{bmatrix} = \boldsymbol{Q}[i] \begin{bmatrix} \lambda^{1/2}\boldsymbol{R}[i-1] \\ \boldsymbol{x}^T[i] \end{bmatrix}$ {Find and apply $\boldsymbol{Q}[i]$, update $\boldsymbol{R}[i]$}
    $\begin{bmatrix} \boldsymbol{p}[i] & \boldsymbol{s}[i] \\ \alpha[i] & \gamma[i] \end{bmatrix} = \boldsymbol{Q}[i] \begin{bmatrix} \lambda^{1/2}\boldsymbol{p}[i-1] & \boldsymbol{0} \\ d[i] & 1 \end{bmatrix}$ {Apply $\boldsymbol{Q}[i]$, find $\alpha[i]$, $\gamma[i]$}
    $\boldsymbol{e}[i] = \gamma[i] \cdot \alpha[i]$ {Calculate error signal}
end for
```

Algorithm 2.3 The QRD-RLS algorithm

"inverse" matrix that can compute the optimal weights directly. It it is described by equation (2.9) [12,ch10]:

$$\begin{bmatrix} \widetilde{\boldsymbol{R}}[i] & \widetilde{\boldsymbol{R}}^{-T}[i] \\ \boldsymbol{0}^T & \boldsymbol{v}'^T[i] \end{bmatrix} = \boldsymbol{Q}[i] \begin{bmatrix} \widetilde{\boldsymbol{R}}[i-1] & \widetilde{\boldsymbol{R}}^{-T}[i-1] \\ \tilde{\boldsymbol{x}}^T[i] & \boldsymbol{0}^T \end{bmatrix} \tag{2.9}$$

$\widetilde{\boldsymbol{R}}$ consists of the square-root exponentially weighted covariance matrices $\boldsymbol{R}$ and $\boldsymbol{p}$ from QRD-RLS and the scalar $\gamma[i]$. The rightmost columns contain $\widetilde{\boldsymbol{R}}^{-T} = (\widetilde{\boldsymbol{R}}^{-1})^T$, and as a consequence, scaled weights $-\boldsymbol{w}[i]/\gamma[i]$ appear in the rightmost column of $\widetilde{\boldsymbol{R}}^{-1}$ (or the bottom row of $\widetilde{\boldsymbol{R}}^{-T}$). The inverse matrix uses $\lambda^{-1/2}$ instead of $\lambda^{1/2}$ as forgetting factor.

$$\widetilde{\boldsymbol{R}}[i] = \begin{bmatrix} \lambda^{1/2}\boldsymbol{R}[i] & \lambda^{1/2}\boldsymbol{p}[i] \\ \boldsymbol{0}^T & \gamma[i] \end{bmatrix} \qquad \widetilde{\boldsymbol{R}}^{-T}[i] = \begin{bmatrix} \lambda^{-1/2}\boldsymbol{R}^{-T}[i] & 0^T \\ -\boldsymbol{w}[i]/\gamma[i] & 1/\gamma[i] \end{bmatrix}$$

$$\tilde{\boldsymbol{x}}^T[i] = \begin{bmatrix} \boldsymbol{x}^T[i] & d[i] \end{bmatrix}$$

The algorithm is summarized in algorithm 2.4.

```
$\widetilde{\boldsymbol{R}}[-1] = \begin{bmatrix} \delta^{1/2}\boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{0}^T & 0 \end{bmatrix}$
$\widetilde{\boldsymbol{R}}^{-1}[-1] = \begin{bmatrix} \delta^{-1/2}\boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{0}^T & 0 \end{bmatrix}$
for each timestep $i \geq 0$ do
    $\begin{bmatrix} \widetilde{\boldsymbol{R}}[i] \\ \boldsymbol{0}^T \end{bmatrix} = \widetilde{\boldsymbol{Q}}[i] \begin{bmatrix} \lambda^{1/2}\widetilde{\boldsymbol{R}}[i-1] \\ \tilde{\boldsymbol{x}}^T[i] \end{bmatrix}$ {Find and apply $\boldsymbol{Q}[i]$, update $\widetilde{\boldsymbol{R}}[i]$}
    $\begin{bmatrix} \widetilde{\boldsymbol{R}}^{-T}[i] \\ \boldsymbol{v}'^T[i] \end{bmatrix} = \widetilde{\boldsymbol{Q}}[i] \begin{bmatrix} \lambda^{-1/2}\widetilde{\boldsymbol{R}}^{-T}[i-1] \\ \boldsymbol{0}^T \end{bmatrix}$ {Apply $\boldsymbol{Q}[i]$, update $\widetilde{\boldsymbol{R}}^{-T}[i]$ }
    $\boldsymbol{w}[i] = -\gamma[i] \cdot -\boldsymbol{w}[i]/\gamma[i]$ {Found in $\widetilde{\boldsymbol{R}}[i]$ and $\widetilde{\boldsymbol{R}}^{-T}[i]$ }
end for
```

Algorithm 2.4 The IQRD-RLS algorithm

2.9 Systolic array implementation of RLS algorithms

The QRD-RLS and IQRD-RLS algorithm may be implemented by using a systolic array architectures shown in figure 2.4. Figure is adapted from Harteneck and Stewart, and Ma and Parhi [13, 12]. The array consists of two types of nodes; boundary nodes (circles), and internal nodes.

Every labeled node stores a recursive variable indicated by the label, corresponding to elements from the matrices of equation (2.8) or equation (2.9) depending on which system is implemented. The recursive path from the previous iteration to the next includes a multiplication by the square root of the forgetting factor $\lambda^{\pm 1/2}$. This is contained within each node and not shown in the figure. Nodes labeled p_m store elements of $\boldsymbol{p}$, etc. For instance, at any given iteration i, node p_0 computes $p_0[i]$ by using the previous value weighted with the forgetting factor; $\lambda^{1/2} \cdot p_0[i-1]$. Column nodes marked (0) in the QRD-RLS do not have any recursive variable, and the value used in the QR-decomposition is a constant 0.

The boundary nodes compute c and s parameters for a Givens rotation of the vector $(\lambda^{1/2} r_{n,n}[i], x_{in}[i])$ where $r_{n,n}[i]$ is the variable stored in the node, and $x_{in}[i]$ is the signal received from the nodes top input. Internal nodes propagate c and s parameters to the right, and apply the received c and s parameters to rotate the vector $(\lambda^{\pm 1/2} r_{m,n}[i], x_{in}[i])$. Again, $r_{m,n}[i]$ is the internal variable and $x_{in}[i]$ is the nodes top input. The internal nodes drawn with dotted lines represent $\widetilde{\boldsymbol{R}}^{-T}$ of equation (2.9) and use $\lambda^{-1/2}$, while all other nodes use $\lambda^{1/2}$.

Boundary and internal node operation may be described in a new algorithm, similar to the givens rotation from algorithm 2.2. If we assume that boundary nodes $r_{n,n}$ are initialized with real, non-negative numbers and connected like shown in figure 2.4, it follows that they will remain real and positive, leading to simplifications shown in algorithm 2.5. Here, the multiplication with the square root forgetting factor is included.

```
if $x_{in} = 0$ then
    $c = 1; s = 0; r = \lambda^{1/2} r$
else if $r = 0$ then
    $c = 0; s = \mathrm{sign}(x_{in}^*); r = |x_{in}|$
else
    $c = \lambda^{1/2} r / \sqrt{(\lambda^{1/2} r)^2 + |x_{in}|^2}$
    $s = x_{in}^* / \sqrt{(\lambda^{1/2} r)^2 + |x_{in}|^2}$
    $r = \sqrt{(\lambda^{1/2} r)^2 + |x_{in}|^2}$
```

```
$r = c\lambda^{\pm 1/2} r + s x_{in}$
$x_{out} = -s^* \lambda^{\pm 1/2} r + c x_{in}$
$c = c$
$s = s$
```

Algorithm 2.5 QRD-RLS and IQRD-RLS boundary node operation (left) and internal node operation (right) using givens rotations. Inverse internal nodes of IQRD-RLS use $\lambda^{-1/2}$, all others use $\lambda^{1/2}$

A benefit of the systolic array is its regularity and locality. Blue dotted lines in figure 2.4 shows how latency can be added so that each node only has to provide its results to its neighbors during one cycle, without the need for global signals except the clock. This reduction of signal path length and delay gives an increase in the possible execution speed. A consequence is that inputs at the top of the array must be delayed by different amounts.

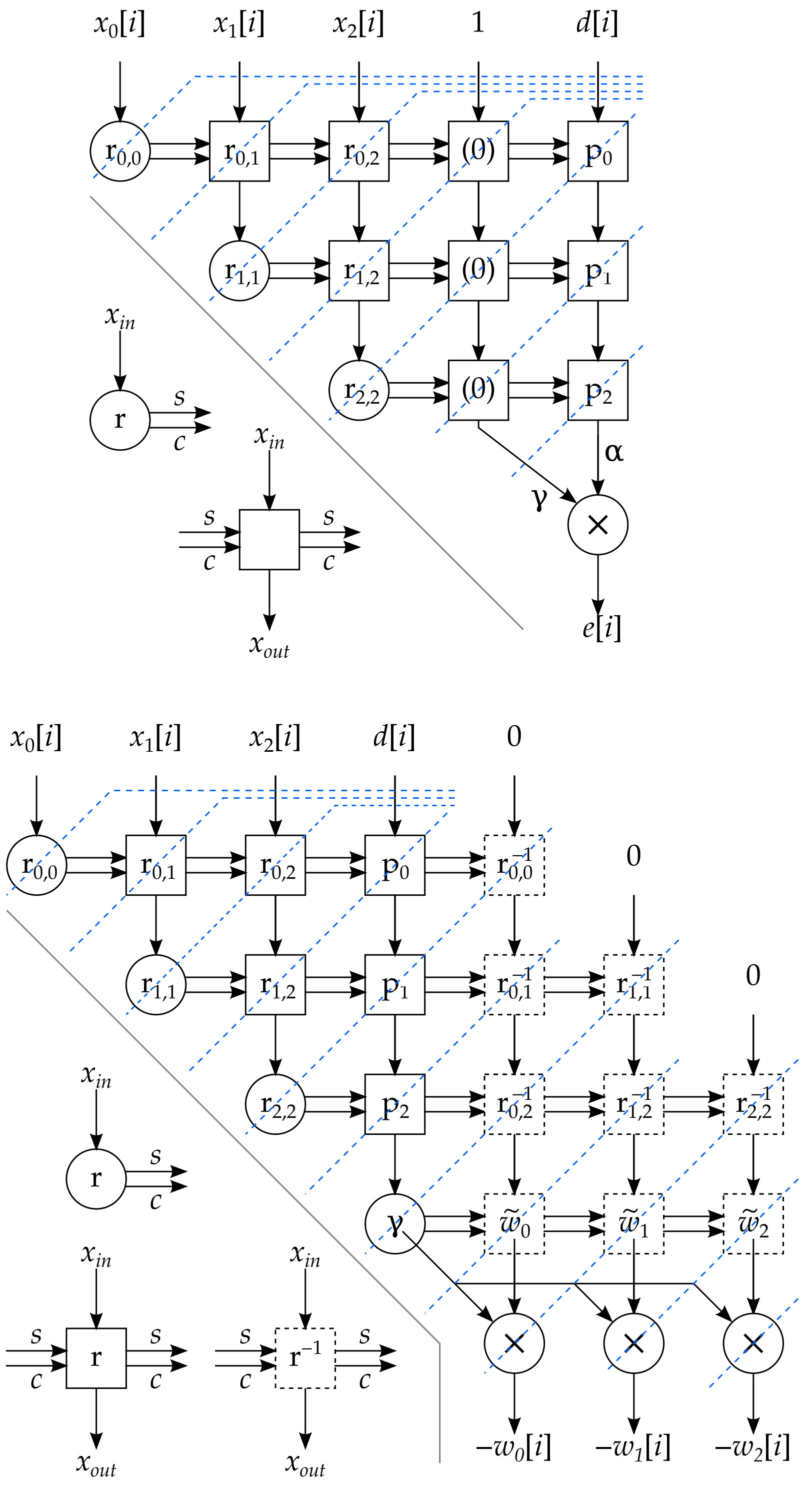


Figure 2.4 Examples of QRD-RLS (top) and IQRD-RLS (bottom) systolic array structures for $MN = 3$. Latencies can be added as suggested with blue dotted lines.

3 Design

3.1 Requirements

This section describes the problem from the assignment text in more detail. The beamformer, which is the focus of this thesis, is shown in relation to surrounding systems in figure 3.1. Each antenna is followed by a digital demodulator which downmixes and samples the received RF signals. This is contained in the radio front-end block.

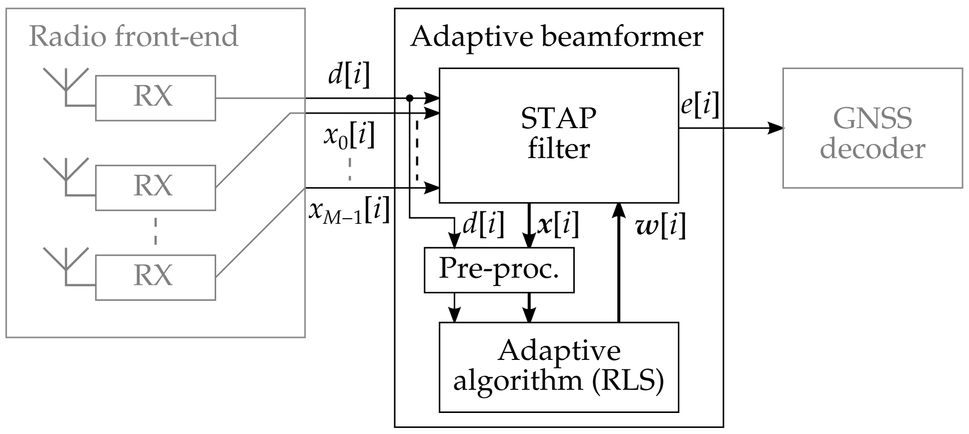


Figure 3.1 System diagram showing the multi channel radio front-end, the adaptive beamformer and the GNSS decoder.

The STAP filter is shown in more detail in figure 3.2. It is a combination of the STAP filter described in section 2.1, and the MMSE criterion of section 2.2. The first channel is designated as the reference channel $d[i]$. Signals from the remaining M channels at N sample times (taps) are grouped into the $1 \times MN$ vector $\boldsymbol{x}[i]$ according to equation (2.1). The STAP filter computes $e[i] = d[i] - \boldsymbol{w}[i]^H \boldsymbol{x}[i]$ which is passed on to the downstream signal processing which can further decode and demodulate the desired signal.

Signals $d[i]$ and $\boldsymbol{x}[i]$ are also passed to the adaptive algorithm. The objective of this algorithm in this system is to find the filter weights $\boldsymbol{w}[i]$ that minimize the power in $e[i]$, which is equivalent to least squares filtering if $e[i]$ is considered to be the error signal. Least squares filtering can be performed with many algorithms[1], for instance Sample Matrix Inversion (SMI), Least Mean Squares (LMS), genetic algorithms or Recursive Least Squares (RLS). This thesis is limited to considering RLS which compared to the other methods has fast convergence speed but high computational load.

The desired signal is supposed to be a spread-spectrum Global Navigation Satellite System (GNSS) signal. A pre-processor is drawn in figure 3.1 that will “hide” this desired signal in the signal passed on to the adaptive algorithm, via an unspecified method. The desired signal is hidden to prevent the beamformer from suppressing it. The test signals used in this thesis do not contain GNSS signals, only only jammer

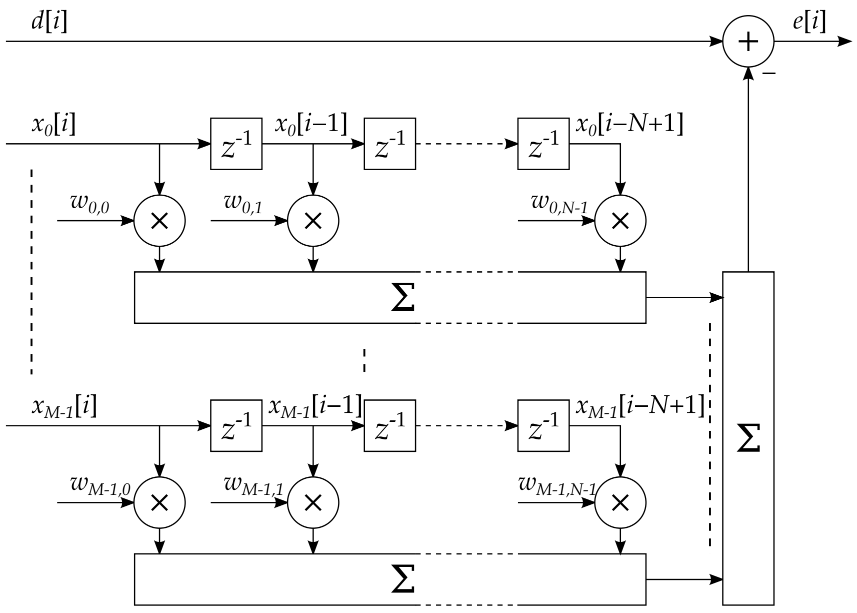


Figure 3.2 Modified STAP filter modelled after the MMSE criterion. Adaptive algorithm (not shown) is used to update weights $w_{m,n}$.

and noise floor. For this reason, the pre-processor can be assumed to pass $d[i]$ and $\boldsymbol{x}[i]$ through unmodified.

A total of $M + 1 = 6$ antennas are available, of which the first is designated as the reference antenna. The number of taps N is suggested to be in the range 1-4. $d[i]$ and $\boldsymbol{x}[i]$ are complex, discrete-time signals available at a rate of 125 MHz. The data is available in signed integer format, with 16 bits for each of the real and imaginary parts. It is desired for the system to find a solution to the filter weights within 0.1 ms after the jammer is enabled.

3.2 Test signal set

Test signals were provided by KDS to use for evaluation of the algorithms performance. Two sets of signals were supplied, each containing different jammer signal characteristics. Each set consisted of six channels, one for each of the antenna. Both data sets start with approximately 100 samples of uncorrelated noise floor, followed by a strong jamming signal from a single direction that does not move. The noise floor samples were approximately in the value range ±60 for each of the real and imaginary part. Description of each signal set follows.

3.2.1 Multi-tone step from static jammer

In the left column of figure 3.3 it can be seen that the jammer signal amplitude spans roughly ±5000 in both the real and imaginary components (imaginary part not shown).

Distinct tones/peaks are visible in the spectrum. This signal is referred to as the *multi-tone* test signal.

3.2.2 Broadband step from static jammer

This data set was similar to the Multi-tone in that it is a static scenario with a power step, but the spectrum is broader and the signal amplitude and power is greater. It is illustrated in the right column of figure 3.3. The jammer signal amplitude spans approximately ±15000 in both the real and imaginary components (imaginary part not shown). This signal is referred to as the *broadband* test signal.

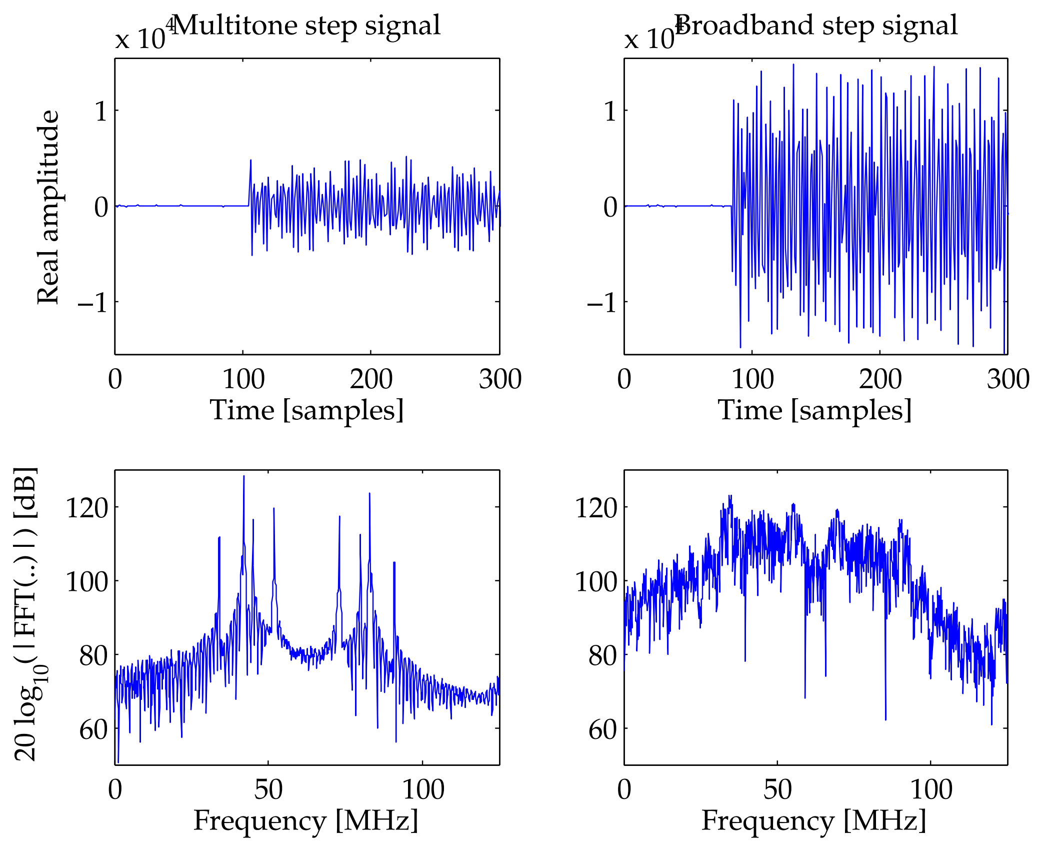


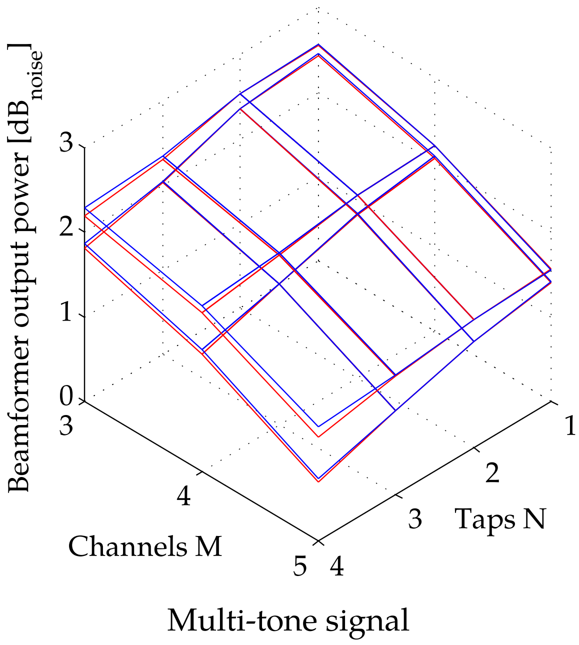
Figure 3.3 Test signals showing a multitone signal (left) and a broadband signal (right). The time domain plots (top) show the step in power near sample 100, while the FFT plots (bottom) show relative differences between the spectra. Only reference channel is shown.

3.3 Sample Matrix Inversion (SMI) vs Recursive Least Squares (RLS) performance

To evaluate the performance of RLS, the SMI method is used as a reference. RLS was compared with SMI by applying both algorithms to the test data sets, with varying antenna/tap configurations, K and λ parameters. Performance is quantified by first estimating the time-averaged power level of the reference channel when jammer is *not* active. This section of the signal will be referred to the *noise* portion of the signal. The power level at the beamformer output $e[i]$ is then estimated using the same method on a section where both noise and jammer signals are present on the input. This number is then converted to decibels relative to the noise power level and is used for comparisons. Signal power is estimated with **equation (3.1)** where $f[i]$ is the chosen section of the signal.

$$P = \frac{1}{N} \sum_{i=k}^{k+N-1} f[i] f^{*}[i] \tag{3.1}$$

RLS was implemented according to **algorithm 2.1**, and MATLAB code is included in **appendix C.3**. The additional signals introduced by STAP filter taps were created using code in **appendix C.1**. For the comparison, a causal SMI was used as in **equation (2.3)**, with code included in **appendix C.2**. A principal difference between RLS and SMI is that RLS uses an exponential window for estimating the correlation matrices, while SMI uses a rectangular window. This makes direct comparison difficult, since the rectangular window length K has no direct relationship to the exponential forgetting factor λ. To handle this, values of K and λ were chosen to get approximately similar performance. From the plot in **figure 3.4**, it is seen that SMI with $K = 125$ performs similarly to RLS with $\lambda = 0.995$. The same is true for $K = 500$ and $\lambda = 0.999$. This is seen by the two sets of red/blue planes overlapping.



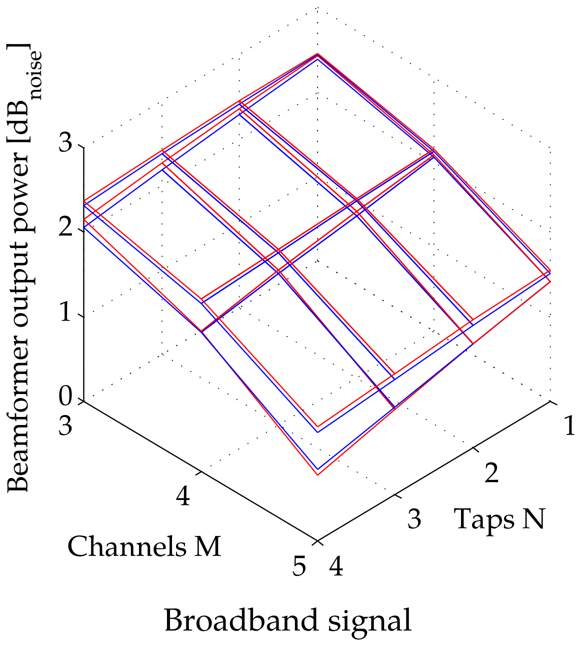


Figure 3.4 Surfaces show output power from SMI (red) and RLS (blue) algorithms for different channel/tap configurations. Within each plot are four surfaces, the blue RLS results for $\lambda = 0.995$ (lower) and $\lambda = 0.999$ (upper), and the red SMI results for $K = 125$ (lower) and $K = 500$ (upper).

Figure 3.4 also shows how the output power is reduced as taps and channels are added. For the broadband signal, additional taps have very little effect. An interesting observation is that the performance for multi-tone and broadband test signals is quite similar, despite the broadband jammer power being approximately 10 times stronger.

The absolute performance is not the main point here, since this can be changed by modifying K and λ. The point is that an RLS-based beamformer can perform equivalently to an SMI algorithm by selecting λ appropriately. In figure 3.4 this is seen by the close overlap of red and blue surfaces.

Another aspect besides steady state performance, is the transient performance of the algorithm. To demonstrate this, the filter weights produced by SMI and RLS were compared and are presented in figure 3.5.

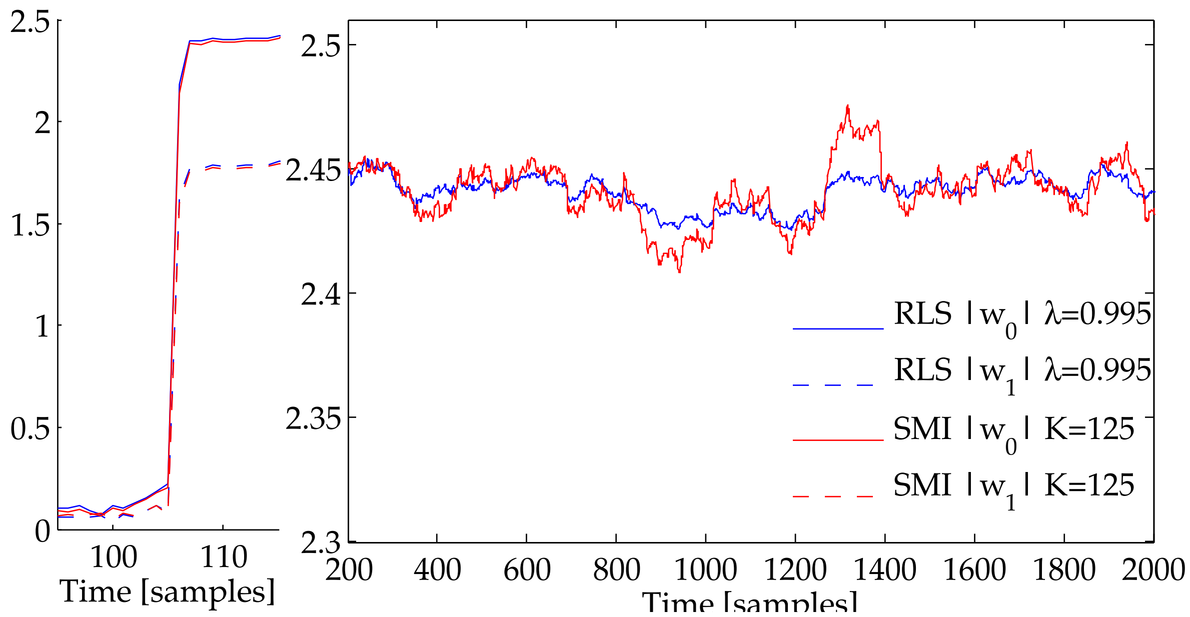


Figure 3.5 Filter weight magnitude behavior for $M = 2$, $N = 1$ configuration, multi-tone test signal. Left graph shows both filter weights at the moment when the jammer is activated. Right graph shows the steady state variation in a single weight $|w_0|$.

From the left part of the figure it is clear that RLS and SMI behave similarly with regards to convergence time, as they both converge within 3-4 samples. SMI and RLS filter weights stay around the same average value, but it is noted that SMI filter weights vary more than those produced by RLS. The variations themselves are caused by the uncorrelated noise in the channels and the variance present in any statistical estimator, and the same effect is also observed with the broadband test signal. This suggests that RLS and SMI react to noise differently.

A conclusion that can be drawn from these results is that RLS can perform equivalently to SMI on the test signals. For further development of an RLS implementation, comparison to conventional RLS is sufficient.

3.4 RLS parameters

Adjustable parameters for RLS consist of the forgetting factor λ, the initialization value δ^{-1} in the inverse correlation matrix $\boldsymbol{P}$ and the initial filter weights $\boldsymbol{w}[-1]$. We can set $\boldsymbol{w}[-1] = \begin{bmatrix} 0 & \cdots & 0 \end{bmatrix}^T$ to start the system with no particular beamforming, effectively eliminating the signals from all except the reference antenna.

To illustrate the effect of the δ parameter, a range of simulations with different values for δ are shown in figure 3.6.

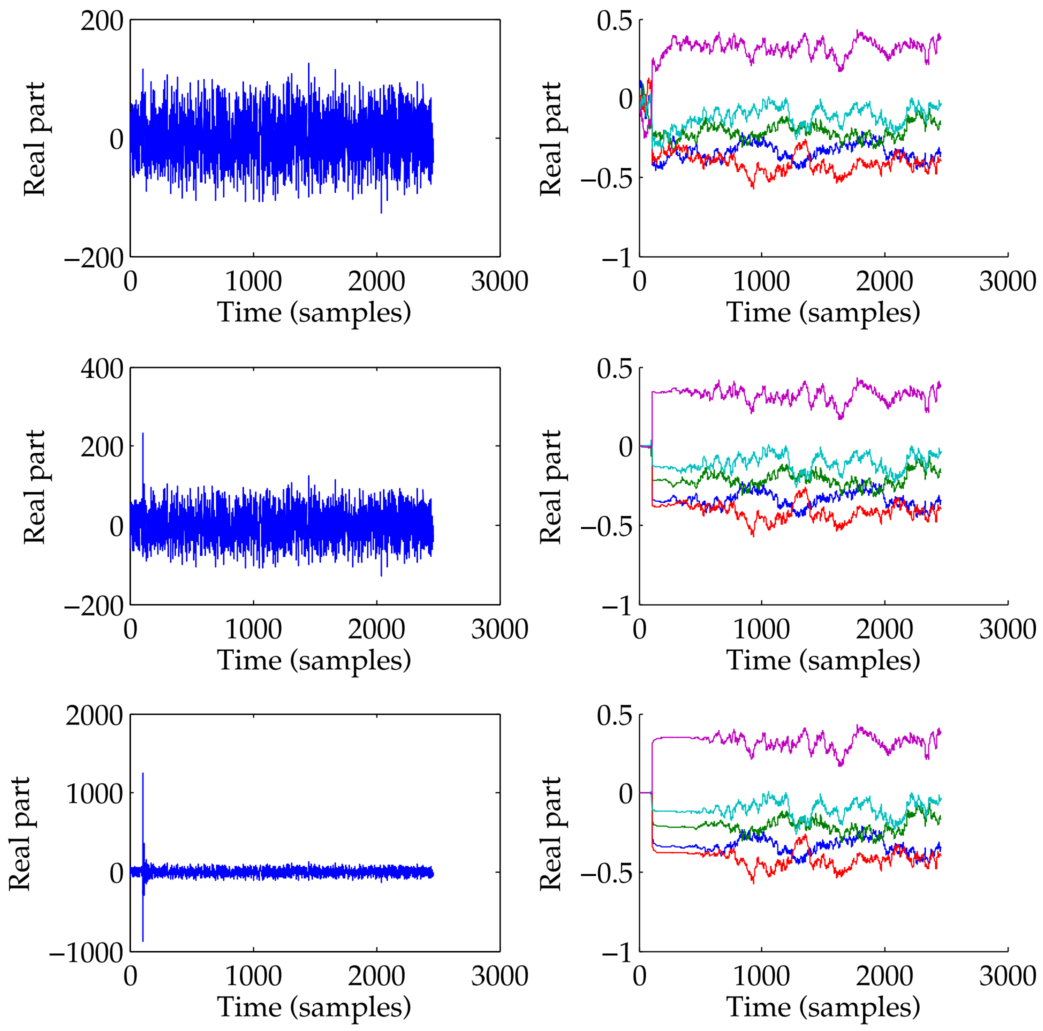


Figure 3.6 Real component of beamformer output $e[i]$ (left), and filter weight behavior (right) for multi-tone signal, $M = 5$, $N = 1$, $\lambda = 0.99$ and varying $\delta \in \{10^5, 10^6, 10^8\}$ from top to bottom. Colors blue, green, red, cyan, purple correspond to weights w_0 through w_4.

In all the simulations the beamformer output stabilizes on the same range of values approximately between ± 100. The effect of σ is seen in the beamformer output around sample 100. When the jammer is enabled, an excursion is visible for $\delta \in \{10^6, 10^8\}$. The cause of the excursion is that the inverse correlation matrix $\boldsymbol{P}$ is initialized for a much higher power than is present in the input signal. The algorithm is therefore less sensitive to the relatively weak jammer, until $\boldsymbol{P}$ converges to reflect the actual signal power level.

This dynamic is also seen in the filter coefficients. For $\delta \in \{10^6, 10^8\}$ the weights start out smoothly, but gradually become more erratic. To get useful results, δ is chosen so that the algorithm converges to the noise power level before the jammer is enabled. $\delta = 10^5$ was found to be acceptable.

When λ is varied as shown in figure 3.7, there is no dramatic effect on the beamformer output amplitude, but filter weight behavior is quite different. When λ is closer to 1, the weights become more stable. From the derivation of RLS, λ is the constant which is used to weight the previous correlation matrix estimates. According to Haykin [5, p450], $1/(1-\lambda)$ can be thought of as the "memory" of the algorithm, thus a high λ should be expected to give more stable results.

3.4.1 Complexity and stability considerations

Conventional RLS has complexity $O(q^2)$ where q is the degrees of freedom, i.e. the number of filter weights MN. For the system considered in this thesis the complexity is therefore $O((NM)^2)$. An overview of one possible breakdown of the calculations required by the RLS algorithm from algorithm 2.1 was created, and is included for reference in appendix D.1. The total number of multiplications are $3q^2 + 3q$, and the number of additions are $2q^2 + 2q$. The single division and a few of the additions and multiplications have one real operand, and will as such require slightly less resources to implement than the remaining complex operations.

There does exist other algorithm variants that are mathematically equivalent to RLS, but use different internal variables and behave differently when implemented with limited numeric precision. The conventional RLS algorithm suffers from instability caused by limited precision, as described by Diniz [4, ch16] and Haykin [5, ch12]. Apolinário [12, ch2] states that instability in conventional RLS is encountered even with double precision floating point arithmetic.

Two other classes of algorithms that implement RLS are described in litterature. The first is the lattice filter. This variant was not considered in depth because it cannot output filter weights, and it also tends to be unstable [12, ch5.7]. The last class is the QRD-RLS filter of which several variants exist. Three variants of QRD-RLS were found in the litterature, and are compared;

- (QRD-RLS [12, 5], described in section 2.7.)
- IQRD-RLS, known as the *inverse-* or *square root* QRD-RLS algorithm [12, 5], described in section 2.8.

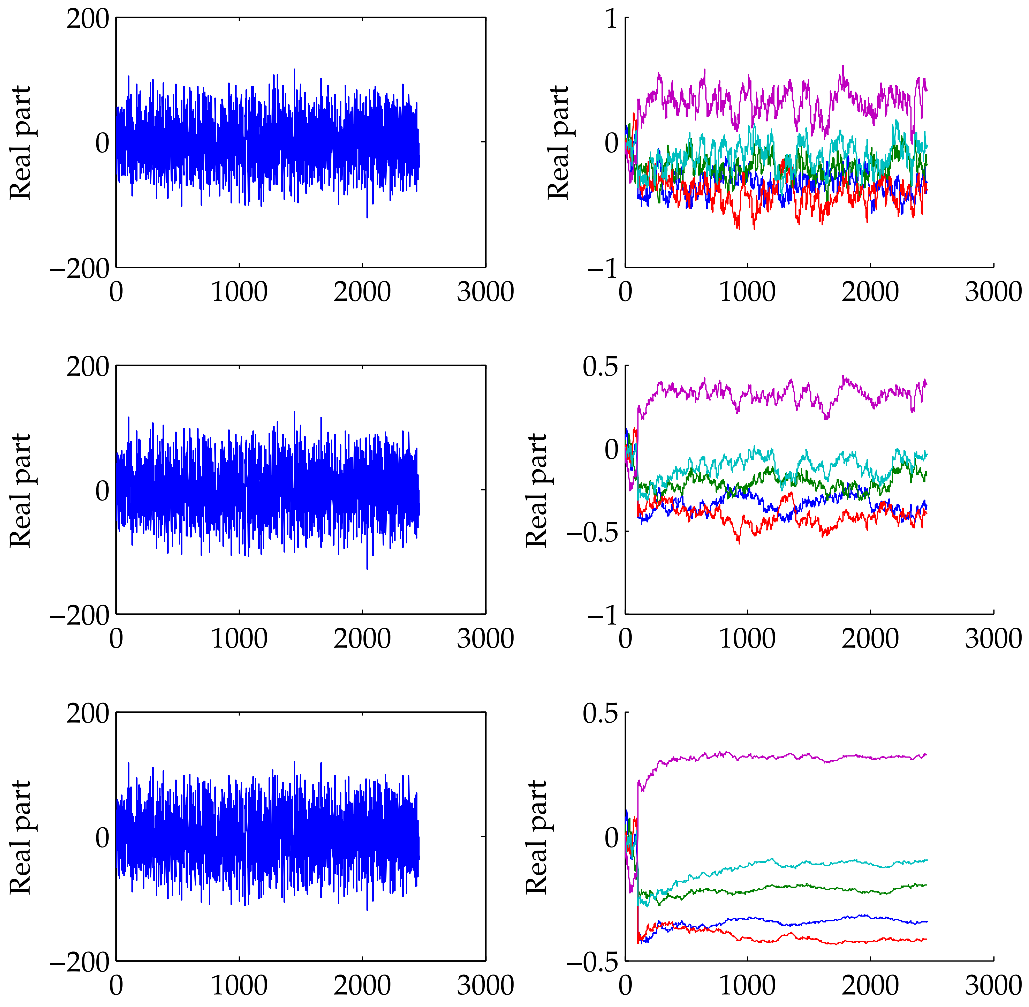


Figure 3.7 Real component of beamformer output $e[i]$ (left), and filter weight behavior (right) for multi-tone signal, $M = 5$, $N = 1$, $\delta = 10^5$ and varying $\lambda \in \{0.98, 0.995, 0.999\}$ from top to bottom. Colors blue, green, red, cyan, purple correspond to weights w_0 through w_4.

- FQRD-RLS, a *fast* QRD-RLS version utilizing the time shifting property of the input data [14, 12].
- MCFQRD-RLS, a *multi-channel* FQRD-RLS variant [15, 12].

At first, QRD-RLS was not considered since it only computes the beamformer output $e[i]$, and not $\boldsymbol{w}[i]$. The inverse QRD-RLS is an extension to QRD-RLS that produces $\boldsymbol{w}[i]$ as part of its normal operation. For the fast variants, an additional procedure is required to extract the weights of the algorithm at a selected point in time. An overview of the computational complexity of the different algorithms is shown in table 3.1. "Filtering" refers to generation of $e[i]$, while "WE" refers to the process of *weight extraction*.

Table 3.1 Computational complexity per iteration for possible QR decomposition based RLS algorithms. M is the number of channels (excluding reference), N is number of taps per channel. [15, 14]

		IQRD-RLS	FQRD-RLS	MCFQRD-RLS
Filtering	Multiplications	$3(MN)^2 + 2MN + 1$	$19MN + 4$	$4M^3N + 11M^2N + 9MN + 5.5M^2 + 7.5M + 1$
	Divisions	$2MN$	$4MN + 1$	$M^2N + MN + 1.5M^2 + M$
	Square roots	MN	$2MN + 1$	$M^2N + MN + M$
WE	Multiplications	-	$7(MN)^2 + MN$	$5M^2N + 5(MN)^2 + M^3N$
	Divisions	(WE not needed)	1	M
	Square roots	-	0	0

From the overview it is seen that the filtering process with the fast algorithms is much simpler than with the inverse QRD-RLS, but the *weight extraction process itself* is more demanding than a normal iteration of inverse QRD-RLS. For a graphical comparison, the variables of **table 3.1** were plotted in **figure 3.8**.

Firstly, from the graph it appears that IQRD-RLS is *always* the simplest, but this is only true if weights are extracted at every iteration (as plotted). The IQRD-RLS algorithm will always produce filter weights, but FQRD-RLS and MCFQRD-RLS only become more efficient as filter weights are extracted rarely. This makes FQRD-RLS and MCFQRD-RLS less suitable for our application where the filter weights must be calculated and applied in near real time.

Second, the weight extraction process is mostly demanding in terms of multiplications, so using FQRD-RLS or MCFQRD-RLS with rare weight extraction would not reduce the number of divisions or square roots significantly. Division and square root operations are slow operations that are not available as hardware blocks in the FPGA and were therefore expected to be a limiting factor to the processing speed of the entire system.

Thirdly, no hardware implementations of FQRD-RLS or MCFQRD-RLS *with* weight extraction are found in the litterature. Based on the presented reasoning, the IQRD-RLS algorithm was selected for further exploration and implementation.

3.5 MATLAB and Simulink simulation of IQRD-RLS

It was necessary to create a MATLAB model of the IQRD-RLS systolic array structure of **figure 2.4** in order to test its equivalence to conventional RLS and to serve as a reference for the hardware implementation.

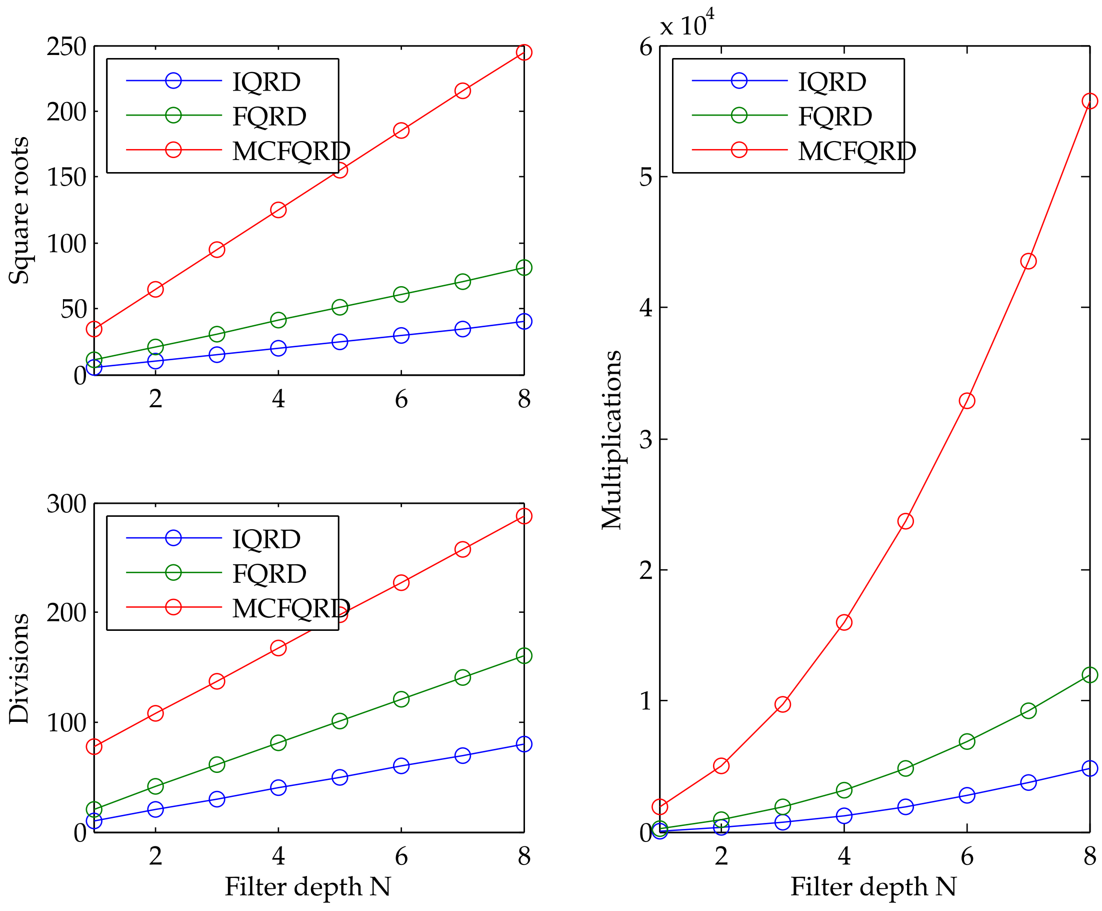


Figure 3.8 Computational complexity per iteration for IQRD-, FQRD- and MCFQRD-RLS algorithms with $M = 5$, and varying number of taps N.

The timing relationship of the IQRD-RLS systolic array in figure 2.4 is illustrated in figure 3.9. Here, the nodes and arrows are arranged to show the data dependency from one cycle to the next as the signals $\boldsymbol{x}[0]$ and $d[0]$ belonging to iteration $i = 0$ flow through the structure from one node to its neighbors, ending up as filter weights $-\boldsymbol{w}[0]$.

To simulate the system, MATLAB functions were first written for each type of node according to algorithm 2.5, describing their input-output relationship. The function arguments is the current state, and the function computes and returns the state for the next iteration. λ (`lambda`) was also included as a parameter. The main reason for this is to be able to switch between two internal node types by supplying $\lambda^{1/2}$ or $\lambda^{-1/2}$ as a parameter, instead of having two separate functions for the regular and inverse internal nodes. Function signatures are as follows ("_n" signifies "next");

```
function [s_n, c_n, r_n] = boundary_givens(r, xin, sqrtlambda)
function [s_n, c_n, r_n, xout] = internal_givens(s, c, r, xin, sqrtlambda)
```

A MATLAB script was then written which manages the current state of the structure, including all node values and signals connecting nodes, and simulates the structure. For each simulated cycle, it iterates through and executes the appropriate function for

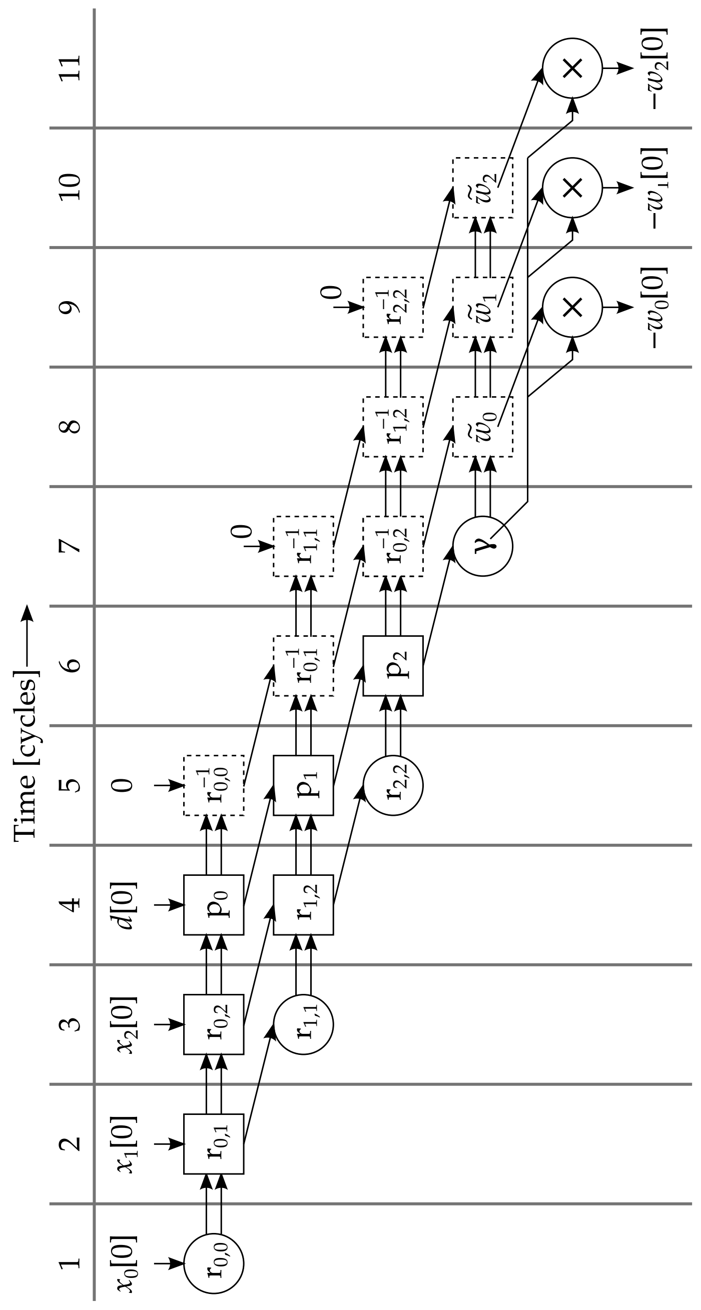


Figure 3.9 Dependency graph showing a single iterations of a IQRD-RLS systolic array structure for $MN = 3$.

all initialized nodes in the structure. Once all nodes are updated, the outputs from the previous cycles become the inputs of the next cycle, and the process is repeated.

Initial conditions of the systolic array are important to control to be able to compare with conventional RLS. Conventional RLS is not implemented as a systolic array, and the computations are not spread over multiple cycles. Due to the distributed, parallel structure of IQRD-RLS, the values "belonging" to one iteration are spread out physically and in time among the processing nodes as seen in the signal flow diagram figure 3.9. One could simulate the systolic array on an iteration-by-iteration basis by propagating a single "wave" of inputs and collecting them on the outputs and repeating this for each set of inputs. The drawback is that this would not simulate the actual cycle-by-cycle behavior of a real implementation, and would be less useful as a reference. Because of this, the algorithm is instead simulated on a cycle-by cycle basis. For the first cycle, only one node (r_{11}) has valid data to process, while in the second cycle both r_{11} and r_{12} can execute and so on according to the signal flow diagram. When valid input signals have propagated to every node, the array is considered fully initialized and every node is evaluated in every cycle of the simulation.

MATLAB code for the simulator is included in **appendix C.7**, and when used with the node functions from **appendix C.4** the results match those of conventional RLS.

As a stepping stone on the way to a System Generator model, a Simulink model was made implementing the nodes and the entire systolic array. The benefits is that Simulink (and System Generator) provides the tools suited for simulating and debugging a model constructed with blocks and signals as opposed to the MATLAB simulator which is a pure software model. A block diagram model of the boundary and internal nodes was made in Simulink, and a top level graphical model was created which connects the nodes together to form the IQRD-RLS array. The diagrams for this model are included in **appendix A.1**.

The results of the Simulink model matches those of the IQRD-RLS simulator, which again matches the conventional RLS algorithm within tolerances on the order of 10^{-15}. This can be attributed to quantization effects of double precision arithmetic.

3.6 Hardware implementation considerations

3.6.1 Systolic array latency

Due to the regular structure of the IQRD-RLS systolic arrays in **figure 2.4**, the area required in terms of nodes is simple to derive. For MN filtered inputs and one reference channel, the structure requires $MN + 1$ boundary nodes and $(MN)^2 - 1$ internal nodes. Since all nodes operate on the same clock frequency, the slowest type of node will be the bottleneck of the system

Based on the data flow diagram in **figure 3.9**, the maximum latency is derived to be $3MN + 2$ cycles from the top left node to the bottom right output node.

3.6.2 Floating point vs fixed point, Givens rotation vs CORDIC

The algorithm may be realized with only Givens rotations or CORDIC. CORDIC operates on fixed-point numbers. Givens rotations could be implemented with either floating or fixed point. To simplify the terminology, a CORDIC based implementation will be referred to as “CORDIC”, whereas a fixed and floating point Givens rotation based implementations are simply referred to as “fixed point” and “floating point” respectively.

Some litterature exists on a variant for floating/fixed point Givens rotation, called the Square Root Free (SRF) Givens rotation. Kile's [16,ch3] comparison of conventional and SRF Givens rotation concluded that the SRF variant needed more computation than conventional Givens rotation when complex numbers are used, and required 10 times the number of divisions. The SRF variant was not used for this reason.

Figure 3.10 shows diagrams of the overall architecture for the different node types. Note that the CORDIC based nodes uses multiple vectoring or rotating units to handle complex numbers as described in works of both Rader, Gao and Maltsev [9, 10, 11].[2]

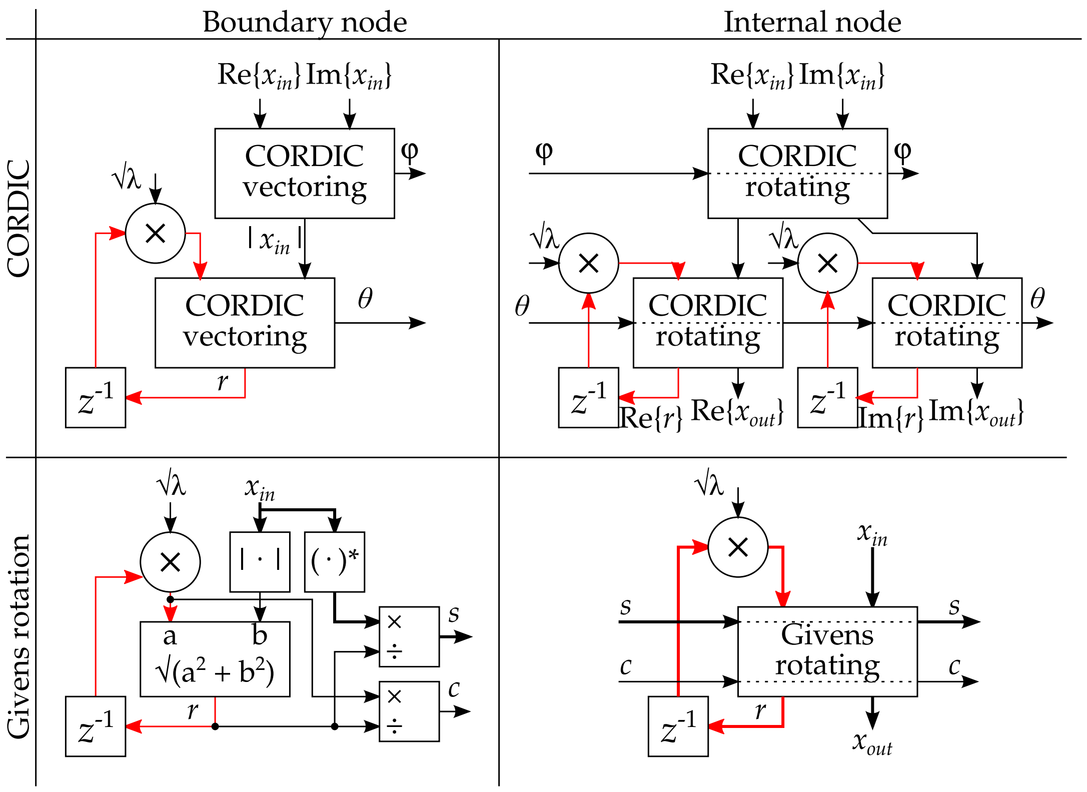


Figure 3.10 Possible architectures for the boundary nodes (left) and internal nodes (right). CORDIC on top, Givens rotation on bottom. Loops highlighted in red. Thick lines represent complex numbers.

Any system having a loop in its signal flow graph is limited in its operating speed. This limitation is commonly quantified as the *loop bound* [17,slide 11.17], defined by (loop delay)/(# registers in loop). The inverse of the loop bound is the theoretical maximum speed, assuming the delay can be divided equally between the registers. The limitation imposed by all the loops in a system is called the *iteration bound* as defined in equation (3.2).

$$1/F_{max} = \max_{\text{all loops}} \frac{\text{loop delay [s]}}{\text{\# registers in loop}} \tag{3.2}$$

[2] In the case of real-valued signals, only a single vectoring/rotating CORDIC block is needed in the boundary/internal node.

In the IQRD-RLS system, one loop exists in every node and is highlighted in red in figure 3.10. For Givens rotation, the loop calculates the magnitude of x_{in}, followed by the square root of the sum of r^2 and magnitude of x_{in}. For the CORDIC based node, the loop involves one CORDIC operation.

To get an estimate for the expected performance, the square root, mulitplication and addition operators required to compute r were synthesized and routed as individual systems using System Generator. A 16 bit fixed point was compared with the 16 bit *IEEE 754 half-precision* format which uses a 5 bits exponent and 11 bit mantissa. The fundamental difference between fixed and floating point formats makes them difficult to compare one-to one, as their benefits and drawbacks depend on the application.

Operators were configured with zero pipelining and an unrealistically high target clock frequency to make the synthesis, place and route tools analyze and optimize the total delay through the operator. The maximum path delay calculated by the analysis tools for each individual operator could then be combined to approximate the delay around the entire loop. Operator configuration was also set for maximum speed when possible. For CORDIC operations the scale compensation factor S was not included as a separate multiplier since it could be included into the multiplier for the $\lambda^{\pm 1/2}$-term. Addition and multiplication operators were tested with both DSP48 and fabric-only configuration. The results reported by the tools are summarized in table 3.2.

From the results, it is noted that fixed point is generally faster than floating point alternative, except for square root. It is also seen that fabric implementations are generally faster than embedded multipliers.

For a CORDIC implementation (fixed point), the loop consists of one multiplication and one CORDIC vectoring operation. Using a fabric-only multiplier gives the fastest solution with a delay of 35 ns with 913 LUTs.

For a Givens implementation (fixed or floating point), the loop is made up of two multiplications - the first for $r \cdot \lambda^{1/2}$ and the second for $(r\lambda^{1/2})^2$. The result must be added to $|x_{in}|$ and followed by a square root. For fixed point integer mode square root, the delay is 44.1ns using 942 LUTS. For floating point the delay is 48.5 ns, and logic usage is 650 LUTs.

Comparing Givens floating point to CORDIC on speed and area usage, it can be seen that they are similar - the speed gained by choosing CORDIC is proportionally offset by its increased size. The fixed point implementation has somewhat more delay and logic usage than CORDIC, leaving CORDIC and floating point for further comparison.

Besides the comparison of floating point and CORDIC, there are other aspects which are difficult to compare quantitatively without a complete implementation of both types in a system.

- CORDIC allows a low-level optimization (not available in the Xilinx IP core), where the direction of each micro-rotation is transmitted to the internal nodes instead of

[3] Results were identical to floating point addition with fabric setting. No DSP48 unit was used.

Table 3.2 Overview of the delay and resource consumption for square root, multiplication and addition operators, for fixed and floating point formats.

Operation		Delay [ns]	Slices	Reg.	LUT	DSP48	Note/setting
CORDIC	Fixed	28.9	173	49	632	0	Vectoring
	Fixed	31.7	263	70	951	0	Rotating
Square root	Fixed	30.1	101	49	361	0	Integer
	Fixed	27.0	105	48	323	0	Fractional
	Float	19.9	30	0	152	0	-
Multiply	Fixed	6.07	0	0	0	1	DSP48
	Fixed	6.08	77	0	281	0	Fabric
	Float	12.4	21	0	45	2	DSP48
	Float	9.29	58	0	170	0	Fabric
Add	Fixed	3.94	0	0	0	1	DSP48
	Fixed	1.86	5	0	19	0	Fabric
	Float	10.0	48	0	158	0	DSP48[3]
	Float	10.0	48	0	158	0	Fabric
Target device capacity			37680	-	150720	768	XC6VLX240T-1

the accumulated angle [8]. This eliminates the unnecessary accumulation and limits the latency between vectoring and rotating iterations.

- CORDIC's regularity makes it pipeline well, because of its many identical stages
- Floating point can make use of the DSP48 resources available, but CORDIC cannot except for the scale factor S.
- Floating point requires additional divisions, increasing boundary node size
- CORDIC requires more logic in internal node than in boundary node
- Floating point has the longest (slowest) loop in the boundary nodes, whereas the loop delay for CORDIC is similar in both types of nodes
- Floating point has a wide dynamic range and is used in MATLAB, making the transition from MATLAB to implementation simpler.

Based on these properties, floating point Givens rotation was selected for further implementation of the IQRD-RLS array.

3.7 Implementation of floating point complex IQRD-RLS

For a system operating on complex values, node signals x_{in}, x_{out} and s are complex, while c and $\lambda^{1/2}$ are real. r is real in the boundary node, but complex in the internal nodes. Based on figure 3.10, a more detailed diagram was drawn as shown in figure 3.11.

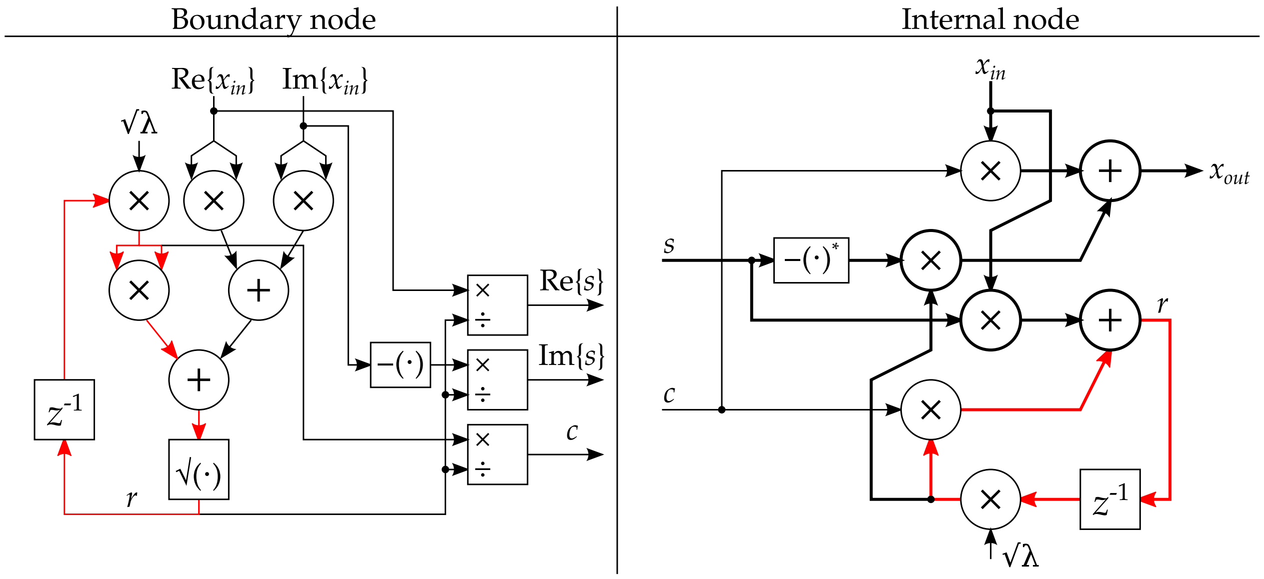


Figure 3.11 Node construction for bounday and internal nodes implementing algorithm 2.5. Bold lines for multipliers and adders indicate complex operations which are not expanded for simplicity. Loops drawn in red.

Due to the fact that the boundary nodes loop consisted of two multiplications, one addition and one square root, it was estimated to have a larger delay than the internal node. The internal nodes loop consists of two *half-complex* multipliers (complex-real multipliers), as well as one complex addition. Maximum delay through these is no different than real-valued multipliers and adders, so in terms of loop delay, the nodes differ only in the square root operation.

Full-complex multipliers were implemented according to equation (3.3), using four multipliers and two adders.

$$(a + ib)(c + id) = (ac - bd) + i(ad + bc) \tag{3.3}$$

3.7.1 Scope of implementation

The implementation focuses on the systolic array itself. To re-iterate, the systolic array implements RLS which performs the job of the adaptive algorithm in the beamformer as shown in figure 3.1. It receives the reference channel $d[i]$ and "tapped" channel signals $\boldsymbol{x}[i]$ from the STAP filter, and produces filter weights $\boldsymbol{w}[i]$ which are passed to the STAP filter where they are used for filtering $\boldsymbol{x}[i]$. The implemented system is illustrated in figure 3.12.

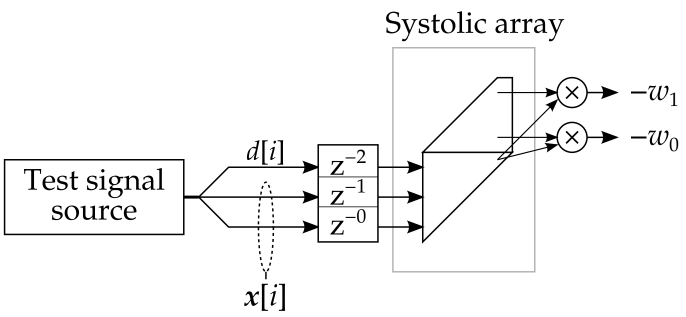


Figure 3.12 Diagram of designed system. The systolic array portion (indicated) is designed for hardware.

3.7.2 Drift issues

MATLAB was first used to evaluate the impact of varying precision in the algorithm. The boundary and internal node functions were modified to simulate limited precision effects. Code is included in appendix C.5. A third party function `roundfloat()` [18] seemed useful for limiting the number of bits of the mantissa, but its behavior was questionable as it did not give identical results when compared against MATLABs built-in conversion from double precision to single precision.

Because MATLAB simulations were inconclusive, the system was implemented with System Generator to see the effects of limited precision by simulating the actual hardware design. A small $MN = 2$ design was made, which is included in appendix B.1.

When simulating the systolic array using limited-precision with 16 mantissa bits (and less), an effect was observed where the filter coefficients appeared to *drift*. The weights drifted exponentially from their correct value, eventually leading to incorrect values. This effect had not been visible in earlier double- or single precision simulations. By changing the the precision it was seen that the drift rate was related to the mantissa precision level, and reducing precision increased the drift rate.

By some trial and error and changing which values of x_{in}, $\lambda^{\pm 1/2}$, s, c, x_{out} and r were quantized using `roundfloat()`, the drift was found to be caused by $\lambda^{\pm 1/2}$. In the first "triangle" of the IQRD-RLS systolic array where signals are applied, $\lambda^{1/2}$ is used, while in the other "triangle" the inverse $\lambda^{-1/2}$ is used. These values are used to independently update the node variables r and r^{-1}. It is believed that since the filter weights are formed by multiplying γ from the left triangle with the $\tilde{w}_n$ of the right triangle, the *difference* in accumulated quantization error explains the drift/divergence of the weights.

Through hardware simulations it was observed that when the quantized $\lambda^{-1/2}$ was smaller than its theoretical value, the filter weights drifted towards zero. When the quantized $\lambda^{-1/2}$ was larger than its theoretical value, the filter weights drifted away from zero in the direction of their sign. For mantissa widths of 16 bits, the drift had a detrimental effect on the output power within rougly 1000 samples, but the drift is visible in the filter coefficients even earlier. An example illustrating the drift is shown in figure 3.13.

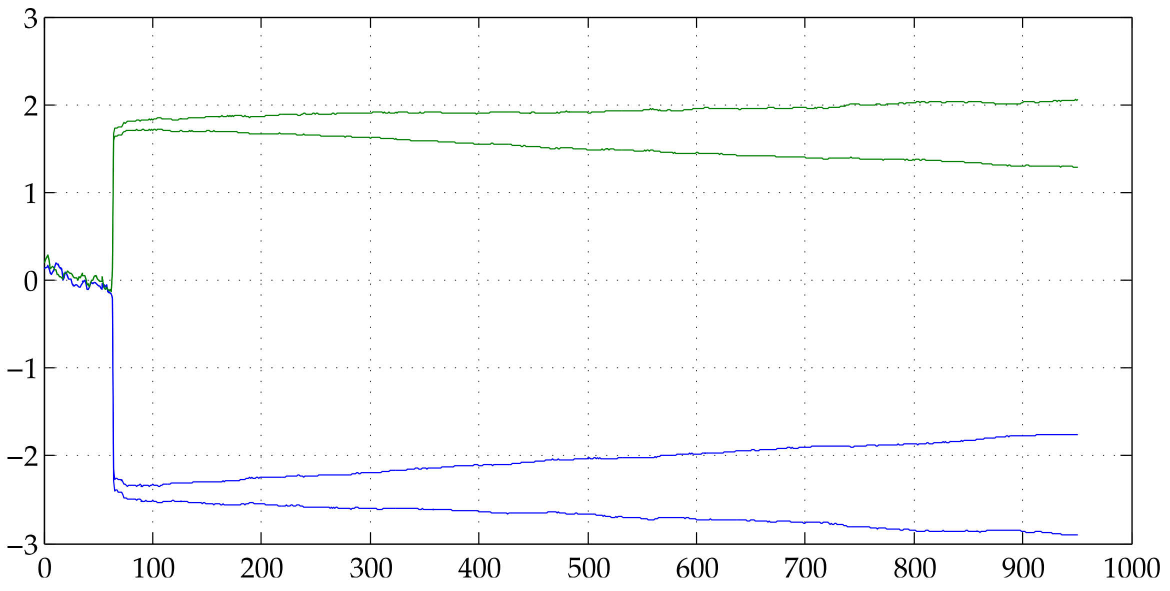


Figure 3.13 Real value of filter weights w_0 (blue) and w_1 (green) from two overlaid simulations for $M = 2$, $N = 1$. 6 bit exponent, 12 bit mantissa, $\lambda^{1/2} = 0.989990234375$ (quantized). The two cases show $\lambda^{-1/2} = 1.01025390625$ (weights drifting away from 0) and $\lambda^{-1/2} = 1.009765625$ (weights drifting towards zero).

The drift is obviously not desired since it leads to incorrect weight solutions, so several methods to mitigate it were considered.

3.7.2.1 Increasing precision of $\lambda^{\pm 1/2}$

Precision could be directly increased by using more mantissa bits in the $\lambda^{\pm 1/2}$ constants and the operators that constitute the node loop. The drawback is that increasing precision will increase loop delay, reducing performance.

Another interesting approach is to make sure that the time-average of the quantized $\lambda^{\pm 1/2}$ is equal to its theoretical non-quantized value. Assuming the example in figure 3.13, this would mean switching between $\lambda^{-1/2} = 1.01025390625$ and $\lambda^{-1/2} = 1.009765625$ in a ratio so that the average approaches the ideal value 1.01011097411 which is not representable in finite precision binary.

However, because of the floating point format, the magnitude and sign of the accumulated quantization error depends on the value r being multiplied by $\lambda^{-1/2}$. The average value of r depends ultimately on the correlation and powers of the input signals $x[i]$ and $d[i]$ to the systolic array, which are not predictable.

3.7.2.2 Careful selection of $\lambda^{\pm 1/2}$

It is conceivable that there are some quantized values of $\lambda^{\pm 1/2}$ with a quantized inverse that is closer to the theoretical value. By restricting the choice of $\lambda^{\pm 1/2}$, drift can be minimized although not eliminated given a certain precision.

Simulations with the only exact solution $\lambda^{1/2} = 0.5$ and $\lambda^{-1/2} = 2$, showed that a similar type of drift still existed, but on a larger time scale. In any case, proper choice of forgetting factor depends on the jamming scenarios and the systems surrounding the beamformer. Small forgetting factors like $\lambda^{1/2} = 0.5$ leads to larger variation in the weigths as was illustrated in **figure 3.7**, and reduces the memory of the algorithm.

It was decided to abandon the idea of solving for filter weights every cycle, and the IQRD-RLS structure. Instead, the system was simplified to leave the left hand triangular systolic array of a QRD-RLS structure.

3.8 Implementation of floating point complex QRD-RLS

Use of the QRD-RLS systolic array left two methods of extracting the filter weights.

One could use a separate *back-substitution* system as described by Diniz [4,p385]. This method uses back-substitution to solve the matrix equation $\boldsymbol{R}[i]\boldsymbol{w}_{opt}[i] = -\boldsymbol{p}[i]$ for the weights $\boldsymbol{w}[i]$. Because of the latency added to the array as shown in **figure 2.4**, elements belonging to the matrices $\boldsymbol{R}[i]$ and $\boldsymbol{p}[i]$ are spread out in time This makes back-substitution a serial process. Back-substitution requires multiplication, addition and division.

The other method is *weight flushing* [19]. The QRD-RLS array produces the error signal (the filtered output) according to $e[i] = d[i] - \boldsymbol{w}^H[i]\boldsymbol{x}[i]$, but the weights $\boldsymbol{w}[i]$ are not explicitly available in the structure. Consider setting $d[i] = 0$ and choosing $\boldsymbol{x}[i] = \boldsymbol{e}_k$. Here, $\boldsymbol{e}_k$ is a unit vector in dimension k.

The systolic array output will be $e[i] = 0 - \boldsymbol{w}[i]^H\boldsymbol{e}_k = -w_k^*$. Thus, by sending such impulses on every channel (for all k), all the negated conjugate filter weights become available on the error output. Like back-substitution, this is a serial process. The nodes in the system must be "frozen", and must not adapt to the impulses since they are not valid data. This is most simply achieved by inhibiting update of r within a node when it processes the impulse. An example impulse for a $MN = 3$ array is:

$$\begin{aligned}
\boldsymbol{x}[i] = \boldsymbol{e}_0 = [1,0,0]^T &\rightarrow e[i] = -w_0^* \\
\boldsymbol{x}[i+1] = \boldsymbol{e}_1 = [0,1,0]^T &\rightarrow e[i+1] = -w_1^* \\
\boldsymbol{x}[i+2] = \boldsymbol{e}_2 = [0,0,1]^T &\rightarrow e[i+2] = -w_2^*
\end{aligned}$$

There are drawbacks and benefits to either method. The main benefit of back-substitution is that the adaptive filter does not need to be "paused" during readout. Pausing the adaptation means that the system will ignore the regular input for the duration of the impulse. However, with weight flushing this blind spot lasts only for a number of samples equal to the number of weights (i.e. the length of the filters impulse response). The RLS algorithm converges very quickly, usually within 5 samples, and is only needed to converge within 0.1 ms. In this context the blind time for the weight flushing method is extremely small and was deemed acceptable.

Weight flushing was chosen as the method for weight extraction because it also is simple to implement.

3.8.1 Simulink model

A new Simulink model was created to simulate both the triangular QRD-RLS systolic array, and the weight flushing method. A MATLAB simulator based on the IQRD-RLS simulator was also created (appendix C.8), but was rarely used since Simulink and hardware simulations were fast enough. The concept is illustrated in figure 3.14. A test signal source produces the test signal (either multi-tone or broadband). A readout sequencer will periodically select and start the impulse generator, applying its signal to the systolic array. The column of delay elements are used to skew the input data in time for correct operation of the systolic array. The impulse signal is marked with the flag "valid=false" which has the effect of inhibiting update of recursive variables within the nodes. The flag follows the impulse through the systolic array to the output $e[i]$. This makes it easy to identify and synchronize to the impulse response when it appears on the output $e[i]$ regardless of the latency through the systolic array. This process is the responsibility of the "weight collector" block, implemented with a MATLAB script run after simulation.

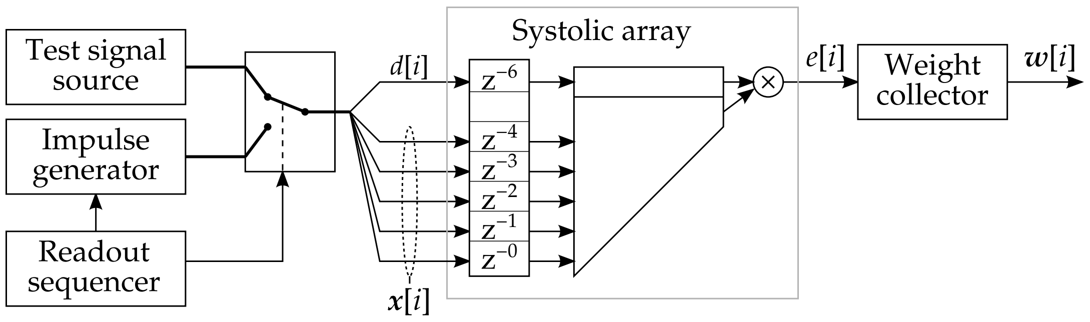


Figure 3.14 Diagram of designed system with a QRD-RLS systolic array and weight flushing. The systolic array portion (indicated) is implemented in hardware.

It should be made clear that figure 3.14 implements the adaptive algorithm block in the adaptive beamformer (figure 3.1). This means that signal $e[i]$ from the systolic array (figure 3.14) is only used to extract filter weights. $e[i]$ from the systolic array is different from $e[i]$ on the system level because of the pre-processor.

Diagrams of the Simulink and System Generator models for the systolic array are included in appendix A.2 and appendix B.2.

Note that the sixth column of the systolic array is fed by the constant value 1 and does not require the full functionality of an internal node. In this type of node r is set to 0, and is not recursively updated. The column is configured this way by setting the *simplify=true* flag on the control bus in the Simulink model. The node effectively computes $x_{out} = cx_{in}$, a half-complex multiplication. The $d[i]$ signal fed into the top row of the systolic array is delayed one extra clock cycle because of the insertion of this extra column.

Correct behavior of the system was checked by comparison with of IQRD-RLS simulations. Both the error signal and extracted weights matched within double precision tolerance. After the first weight extraction, subsequent weight extractions returned slightly different results. This is expected since pausing and flushing of the systolic array causes some information to be discarded.

3.8.2 Preliminary synthesis results

To characterize the system, boundary and internal nodes were built using System Generator according to the diagram of figure 3.11. Complex multipliers in the internal nodes were implemented according to equation (3.3), using four multipliers and two adders. The nodes were implemented using operators configured for lowest possible delay, i.e. no pipelining.

This first design was not expected to be fast, but instead it would give an idea of the amount of logic needed for the nodes and the needed precision. Verification of the node behavior was done by comparing the behavior to the MATLAB functions `boundary_givens()` and `internal_givens()` described in section 3.5. This was straightforward to do with System Generator without having to write HDL test benches.

Synthesis results for boundary and internal nodes are summarized in table 3.3. From the table, some observations can be made;

- The use of DSP48 blocks is mostly unaffected by the number of mantissa bits because the blocks natively operate on larger than 14 bit numbers. Using less bits does not allow (automatic) re-use.
- The number of mantissa bits has no effect on the internal node delay, but a large effect the boundary node delay. This because square root and division delays depend on the precision.
- The internal nodes are faster than the boundary nodes.

It is seen that without further modifications, the nodes have a maximum delay of 80.3 ns and 42.4 ns for the boundary and internal nodes respectively with 14 mantissa bits. The speed is in all cases limited by the boundary node, to between 12.5 MHz and 15.5 MHz depending on precision. Compared to the desired rate of 125 MHz it is clear that more effort is needed to speed up the system. Keep in mind that the boundary node delays shown here are not due to the recursive update of r, but are the delays from x_{in} to the s outputs, which has an additional division after r is computed.

Table 3.3 Overview of the input-output delay and resource consumption for boundary and internal nodes for various mantissa widths (Mant.) [bit], as reported by synthesis tools (no place/route). Exponent width is 6 bits.

	Mant.	**LUTs**	**DSP48**	**Delay** [ns]	**Setting**
Boundary node	14	2917	0	80.0	All fabric
	14	1849	8	80.3	All DSP48
	12	2467	0	71.8	All fabric
	12	1531	8	71.8	All DSP48
	10	1951	0	63.2	All fabric
	10	1287	8	64.7	All DSP48
Internal node	14	5407	0	41.7	All fabric
	14	2669	28	42.4	All DSP48
	12	5609	0	39.4	All fabric
	12	2333	28	40.3	All DSP48
	10	4477	0	37.0	All fabric
	10	2153	28	39.6	All DSP48

3.8.3 Performance vs precision

The chosen implementation uses floating point where the number of bits for mantissa and exponent may be changed. Boundary and internal nodes were assembled to form QRD-RLS systolic arrays of various sizes. The entire system was then simulated with different mantissa precisions and the output power was compared to the input noise level. Because the results from figure 3.4 showed little difference between multi-tone and broadband test signals when using more taps than 1, only a single tap and the multi-tone test were used. Results are shown in table 3.4.

It is seen from table 3.4 that a 14 bit mantissa yields the same performance as the 53 bit double precision reference, and that further reductions in mantissa width leads to degradation of performance (with one exception for (5,1) configuration at 12 mantissa bits).

3.9 Improving speed

To improve the speed of the nodes in table 3.3, latencies were added to the system.

Figure 3.15 illustrates how latency was inserted and redistributed without changing the function of the system. This latency was added on top of existing latency as shown

Table 3.4 Power output for different systolic array dimensions (M,N) and mantissa precision. Multi-tone test signal, $\lambda = 0.995$ quantized to 11 bit mantissa in all cases.

Mantissa	**Output power** [dB$_{noise}$]		
width [bits]	**(3,1)**	**(4,1)**	**(5,1)**
53	2.6	2.3	1.5
14	2.6	2.3	1.5
13	3.0	2.8	2.0
12	4.0	3.9	1.7
11	5.9	5.6	4.5

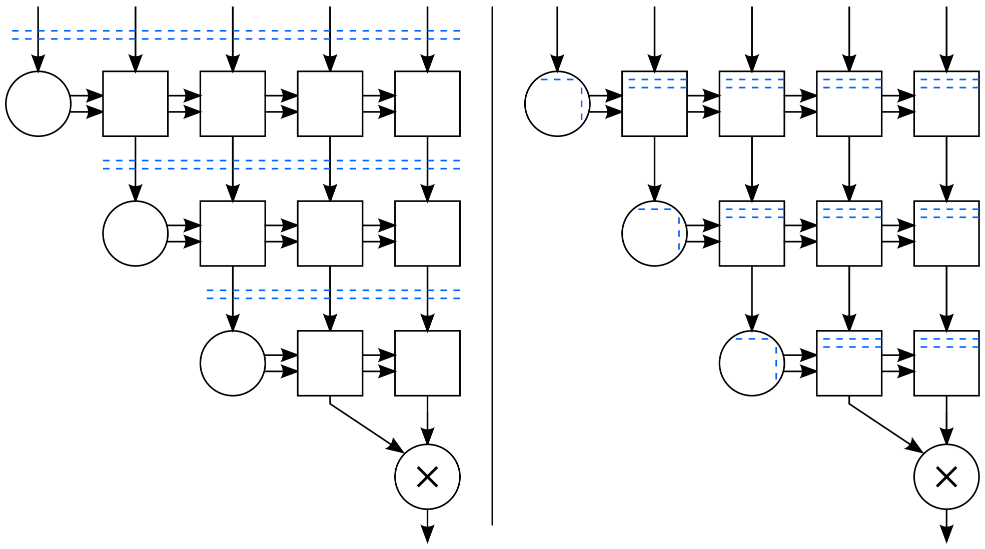


Figure 3.15 Added latency (left) and redistributed latency (right). Blue dotted lines represent cycles of latency.

in figure 2.4. In the boundary nodes, one cycle of latency is pushed to the boundary node output, giving the dividers on the output one full clock cycle to calculate their results. A detailed illustration of the added latency within the nodes is shown in figure 3.16. The latencies were placed in such a way that the loop in the boundary node (red) is the path with longest register-to-register delay and thus the only limit to the operating frequency of the system. This “pipelined” node design is the one included in appendix B.2, and the location of added latency is highlighted in their block diagrams by vertical dotted lines.

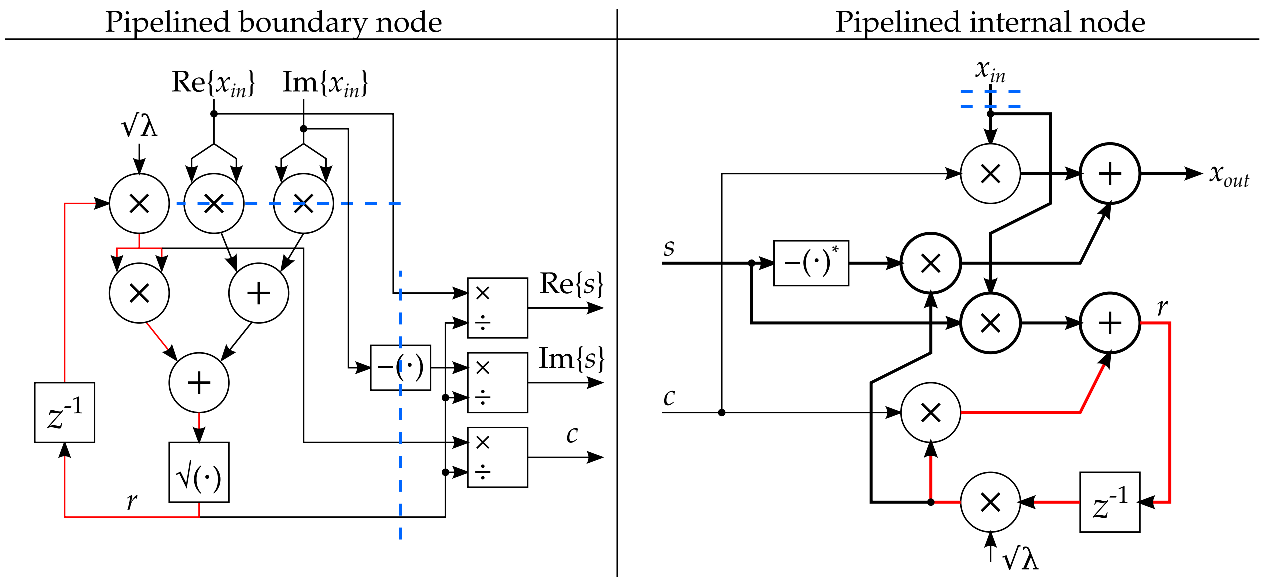


Figure 3.16 Boundary and internal nodes showing where extra latency is added. Added latency is shown with blue dotted lines.

4 Conclusion and further work

This chapter summarizes and discusses the implemented system, suggests improvements and proposes further work.

4.1 Result overview

The preliminary results from table 3.3 was not optimized as described in section 3.9, and used either all fabric, or all DSP48 configured operators. To fit the largest possible array without sacrificing speed, the operator use of fabric and DSP48 resources were balanced. Operators in the boundary node loop were configured for maximum speed and fabric implementation. Operators not in the loop; those used to compute the intermediate result $\mathrm{Re}(x_{in})^2+\mathrm{Im}(x_{in})^2$ were configured to use DSP48 resources and minimum area. Updated results are summarized in table 4.1. The large reduction in delay compared to table 3.3 is mostly due to the addition of latency in the system (section 3.9).

Table 4.1 Overview of resource consumption and max delay for boundary and internal nodes for various mantissa widths (Mant.) [bit], as reported by place and route tools. Exponent width is 6 bits.

	Mant.	**LUTs**	**DSP48**	**Delay** [ns]	**Speed** [MHz]
Boundary node	14	2154	2	51.5	19.4
	13	1950	2	47.8	20.9
	12	1779	2	46.1	21.7
	11	1579	2	42.3	23.6
	10	1439	2	40.6	24.6
Internal node	14	2813	14	37.6	26.6
	13	2578	14	36.9	27.1
	12	2444	14	36.8	27.2
	11	2280	14	36.5	27.4
	10	2193	14	36.0	27.8

Because of the regular composition of the systolic array, predicting resource use and maximum clock rate is straightforward. A QRD-RLS systolic array for M channels and N taps needs MN boundary nodes, and $((MN)^2+3MN)/2$ internal nodes. The maximum operating speed is the minimum of the boundary and internal nodes.

4.2 Algorithmic improvements

The QRD-RLS algorithm relies on a square root computation in each iteration, either implicitly as part of a CORDIC vectoring/rotation operation, or explicitly for a floating point implementation like the one developed in this thesis. This operation consumes most of the time in the loop, and limits the overall throughput of the algorithm. Although faster square root implementations could exist, for operation at the full 125 MHz rate the loop delay including two multipliers, one adder and one square root would have to be 8 ns. This is similar to a single multiplication (**table 3.2**). Since synthesis tools identify the loop as the limit for the speed, it is certain that the current design has reached its limit.

Therefore, the only way to approach 125 MHz is by modifying the algorithm. Some litterature describes a look-ahead transformation which increases speed proportionally to the increase in logic [20][21][22][12,ch10]. Conceptually, the transform changes the order in which elements below the diagonal are zeroed in the QR-decomposition. Instead of rotating each new row $\boldsymbol{x}^T[i]$ against $\lambda^{1/2}\boldsymbol{R}[i-1]$ in **equation (2.8)** so that $\boldsymbol{x}^T[i]$ is zeroed, the transform can rotate $\boldsymbol{x}^T[i]$ against $\boldsymbol{x}^T[i-1]$ first, before rotating against $\lambda^{1/2}\boldsymbol{R}[i-2]$.

The result is that the dependency $r[i] \rightarrow r[i-1]$ is pushed further back in time. By doubling the number of Givens rotations performed by hardware, this dependency becomes $r[i] \rightarrow r[i-2]$. The transformation effectively inserts more z^{-1} registers in the loop path. When these are redistributed around the loop, the operating frequency can be increased.

The mentioned litterature on the look-ahead transform only describes CORDIC implementations for real-valued systems. To implement the look-ahead transform in the floating point system described in this thesis, boundary nodes must first be modified to support complex r. The existing boundary node design supports complex x_{in}, but it was possible to simplify them to use real-valued r. Applying the complex version of **algorithm 2.2** to QRD-RLS, we get $r = \text{sign}(a)\sqrt{|a|^2 + |b|^2}$. Computation of $\text{sign}(a)$ requires another square root and a division[4], which is a significant extension to the current design. Implementing complex r leads to complex c, which will increase node sizes.

4.3 Implementation improvements

4.3.1 Floating-point, fixed-point and CORDIC

Given the current implementation, there are many areas where improvement and optimization is possible. The first is the possibility of converting to fixed point. One major

[4] $\text{sign}(x) = x/|x|$ for $x \neq 0$, 1 for $x = 0$.

advantage of floating point was its dynamic range which would be useful for implementing the IQRD-RLS array, because simulations showed widely different dynamic range in the $\widetilde{\boldsymbol{R}}$ and $\widetilde{\boldsymbol{R}}^{-\boldsymbol{T}}$ matrices. When the IQRD-RLS array turned out to be unfeasible to implement without drift issues, the floating point implementation could still be re-used for the QRD-RLS array. Since fixed-point QRD-RLS is fairly well understood and described in literature [23], the only remaining argument for floating point is the ease of use for the designer.

The current implementation uses Xilinx IP cores [24] for floating point operators. This imposes limits on the mantissa and exponent precision used. For instance, for a 14 bit mantissa the minimum exponent width is 6, even if this is not needed in the system. Another limitation is caused by each operator claiming its own DSP48 block. By converting the current design to fixed-point operators, it becomes possible to schedule a few DSP48 block to implement many of the multiply and accumulate operations such as complex multiplications in the internal nodes which consumes most of the DSP48 blocks (**table 4.1**).

It is also possible to change the balance of addition to multiplication in the internal node complex multipliers. Instead of four multiplications and two additions in **equation (3.3)**, complex multiplication can be done with three multiplications and four additions as in **equation (4.1)**.

$$(a + ib)(c + id) = (ac - bd) + i((a + b)(c + d) - ac - bd) \tag{4.1}$$

An alternative to the floating and fixed point variations is the CORDIC algorithm. CORDIC was initially in **section 3.6.2** found to be comparable to floating point (and fixed point) in speed/area tradeoff. A systolic array implementing QRD-RLS using CORDIC for $MN = 2$ was constructed and tested successfully, but larger array sizes could not be built and tested in time. This implementation is therefore described in more detail separately, in **appendix D.2**.

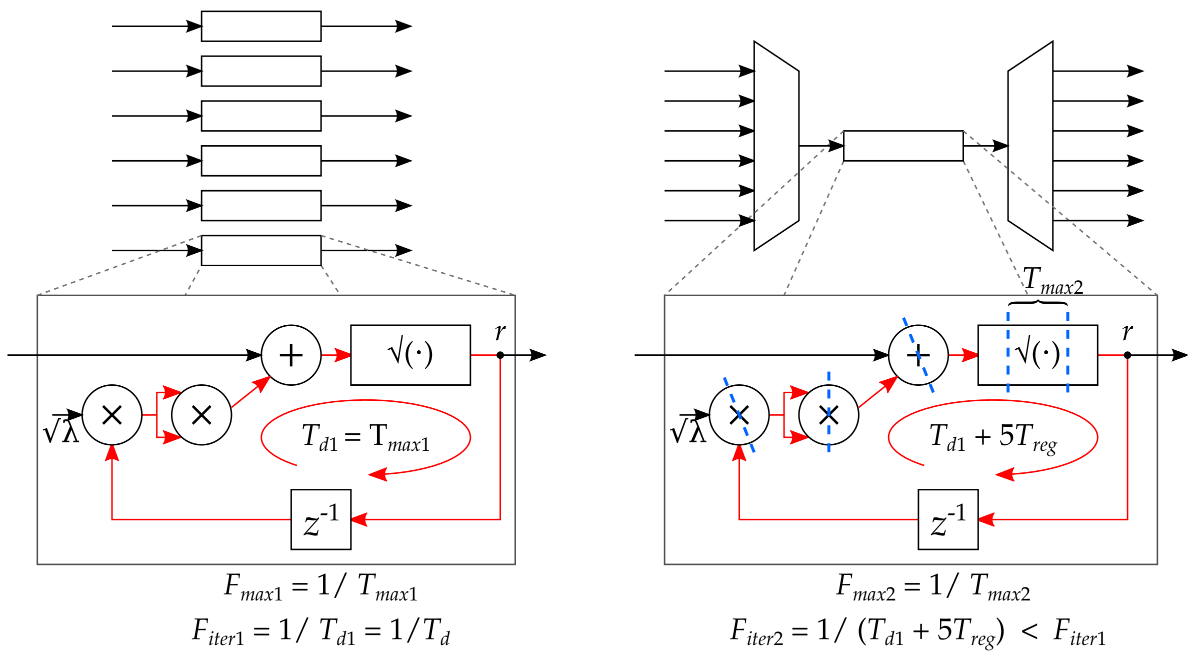
It was found that internal nodes need 4 CORDIC rotators instead of 3 to give results for correct givens rotations matching **algorithm 2.5**. The fact that only three rotator configurations are described in litterature [9, 10], raises doubts as to whether the fourth rotator is necessary. Adding a fourth rotator increases internal node size by $\approx 1/3$.

Similarly to the floating point operators, Xilinx IP cores were also used for the CORDIC blocks. These IP cores did not have the optimization described in **section 2.5** where the d_i sequence can be passed directly from boundary to internal nodes. This means that the third adder used to accumulate θ_i within the CORDIC block can be eliminated, reducing the number of adders in each CORDIC iteration from 3 to 2.

4.3.2 Size reduction

The current design fits up to $MN = 5$ on the target FPGA. It is possible to decrease the amount of logic needed for the system by *channel interleaving* the boundary and internal nodes. Channel interleaving [17,slide 14.19] adds pipeline registers to the loop.

This is illustrated in figure 4.1, where the non-interleaved system has a recursive loop with a single register and loop delay T_{d1}. The added pipeline registers reduce the maximum register-to-register path delay which permits increase in operating frequency. If the system pipelines "well", every register-to-register delay introduced by pipelining is equally long. If this is true for the example in figure 4.1, then $F_{max2} \approx 5F_{max1}$. Each pipeline register adds some delay T_{reg} to the recursive path, slightly decreasing the speed for any single iteration.



√λ
r
√(·)
$T_{d1} = T_{max1}$
z^{-1}
$F_{max1} = 1/\ T_{max1}$
$F_{iter1} = 1/\ T_{d1} = 1/T_d$
T_{max2}
$T_{d1} + 5T_{reg}$
$F_{max2} = 1/\ T_{max2}$
$F_{iter2} = 1/\ (T_{d1} + 5T_{reg})\ <\ F_{iter1}$

Figure 4.1 Illustration on how the size of a system (left) can be reduced by channel interleaving (right). Blue dotted lines represent added pipeline registers. Adding 5 pipeline registers reduces logic to ⅕th.

4.4 Beamforming performance and result validity

Conventional RLS was shown in section 3.3 to have the same potential as SMI for beamforming when a matching forgetting factor is chosen, and sufficient mantissa precision is used. The results of table 3.4 are one example, but selection of the desired value of λ and the required number of bits depends on the properties of the systems surrounding the beamformer (see figure 3.1) and expected jamming scenarios.

Results based on test signals can only be as good as the test signals themselves. There was very little benefit in having more than one tap, especially for large λ and the broadband signal (figure 3.4). For this reason, implementation tests were limited to 1 tap, but with varying number of channels. Because STAP filtering is linear, it does not matter for the adaptive algorithm if it operates on 10 channels with one tap, or one channel with 10 taps or any other combination of constant MN.

One uncertainty was that the test signals were computer-generated and did not contain a desired, "known" GNSS signal which would pass through the beamformer and could be checked to be present on the output. Indeed, if the test signals *had* contained GNSS signals, they might be cancelled or degraded by the beamformer. As shown in **figure 3.1**, an unspecified pre-processor would be used to remove or mask such signals from the adaptive algorithm but not from the STAP filter which creates the final beamformer output.

Since no reference existed for the output signal $e[i]$, validation of conventional- and QRD-RLS algorithms had to be done by comparing filter weights with those from the SMI algorithm. The MATLAB implementation of SMI had earlier been verified by passing the test signal through a STAP filter with weights found by SMI, and observing that the resulting output was reduced in power, close to the noise floor.

The filter weights behave differently depending on the number of channels. From the simulation with $M = 2$ (**figure 3.5**), the RLS weights are stable and within a span of ≈ 0.05 around its mean value, but in another simulation with $M = 5$ and the same λ (**figure 3.7**), the weights vary much more, within a span of ≈ 0.2. The reason for this is that the $M = 5$ array has more degrees of freedom for a weight solution which succssfully suppresses the jammer. Filter weights are free to move around between the multiple equivalently good solutions.

Figure 4.2 compares filter weights flushed from the QRD-RLS array running on actual hardware with those from the double precision conventional RLS algorithm. Flushed values are plotted as discrete time values with solid dots, while the reference weights from conventional RLS are drawn with continuous lines. In this example the weights were chosen to be extracted every 50 iterations. It is clear that the extracted weights match conventional RLS.

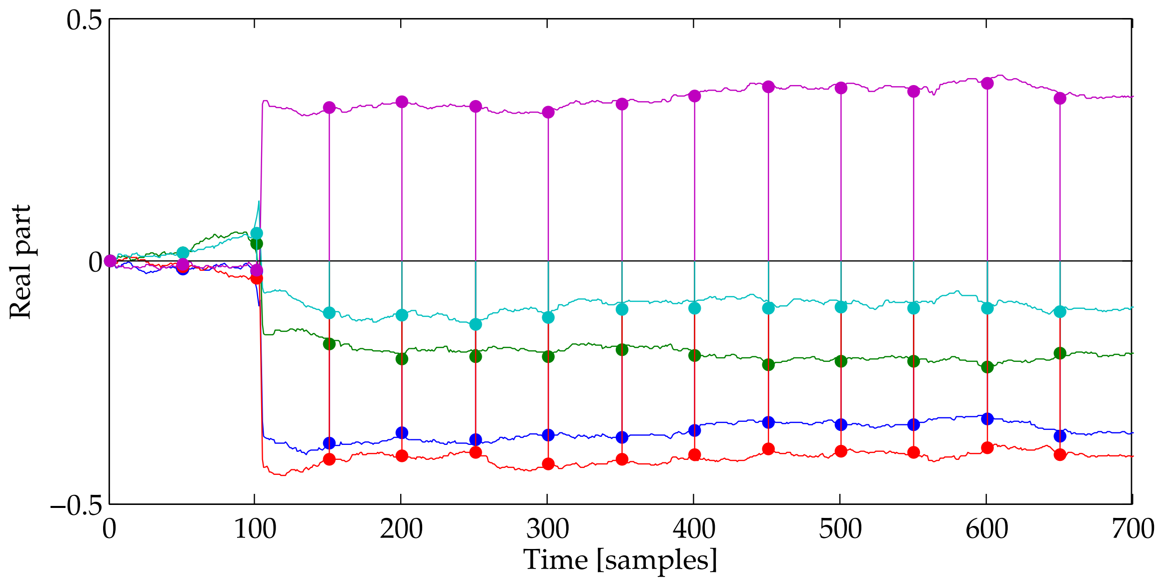


Figure 4.2 Comparison of filter weights from conventional RLS algorithm to filter weights flushed from hardware implementation of QRD-RLS array. Multi-tone test signal, $\lambda = 0.998$, $M = 5$, $N = 1$, 14 bit mantissa. Colors blue, green, red, cyan, purple correspond to weights w_0 through w_4.

4.5 RLS convergence speed and STAP filter update rate

The initially planned IQRD-RLS array would be able to output filter weigths $\boldsymbol{w}$ once per iteration. When IQRD-RLS had to be abandoned due to drift issues, it was decided to simplify the array to a QRD-RLS array which outputs $e[i]$ instead. Filter weights could then be recovered after applying impulses and reading the weights from the impulse response.

Because the current implementation is limited to ≈ 20 MHz, the 125 MHz signal must either be downsampled, or a portion of it must be selected and buffered for processing with the QRD-RLS array. There is also a free choice of when to flush the weights. These can be done automatically at a fixed frequency, or could be triggered based signal characteristics such as the power level at the beamformer output. The best strategy would depends on the jammer signal characteristic, and would require testing with a wider selection of test signals.

RLS and the current QRD-RLS implementation converge within less than 5 iterations (cycles) on the applied test signals. Latency from input to output for the QRD-RLS array with added latency is $4 \cdot MN + 1$ cycles, and weight readout takes MN cycles, one cycle for each of the weights. A conservative estimate of the minimum delay from jammer being enabled to weights being read out is $5 + (4MN + 1) + MN = 5MN + 6$ cycles. At the acheived speed of ≈ 20 MHz, a full size system with $M = 5$ channels and $N = 4$ taps would only need 5.3 μs to output a useful result. Compared to the requirement of convergence within 0.1 ms, the system is performs its job very well.

Figure 4.3 compares the final beamformer output $e[i]$ found using the conventional RLS algorithm, to the $e[i]$ output of a STAP filter using weights flushed from the QRD-RLS array running on actual hardware. The filter weights (shown in figure 4.2) are flushed and updated every 50 samples. Just after a weight flushing at sample 100, the jammer is turned on and becomes visible in the output. At sample 150, the next set of weights is flushed and applied, suppressing the jammer. From this point, the filter weights produced by the hardware implementation are successfully suppressing the jammer.

4.6 Workflow and tools

The overall workflow started with using MATLAB to implement the mathematically formulated conventional RLS algorithm (section 2.3.2) and SMI (section 2.3.1). Once the MATLAB-implemented RLS behaved as it should, it could serve as a reference for MATLAB simulations of the IQRD-RLS.

To have a uniform interface for the algorithm functions, the individual test signal channels were "packaged" into `timeseries` objects. These objects were then grouped using `tscollection` objects (time series collection). The algorithms are implemented to accept the time series collection as input, and generates two time series $e[i]$ and $\boldsymbol{w}[i]$ as

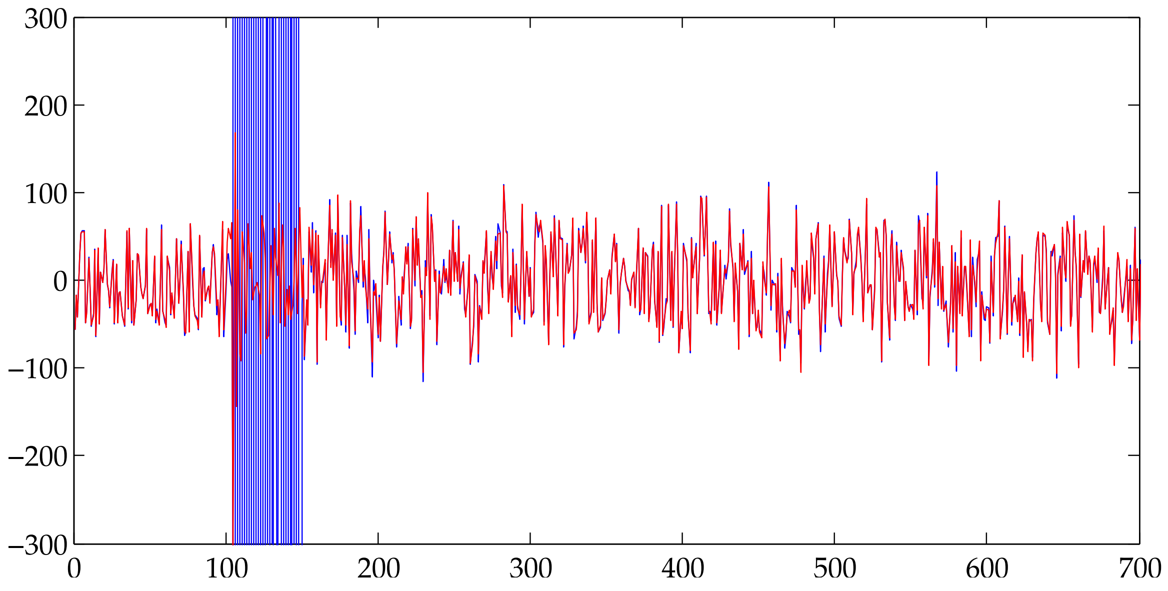


Figure 4.3 Comparison of beamformer output from conventional RLS (red), and hardware simulation of QRD-RLS array (blue). Multi-tone test signal, $\lambda = 0.998$, $M = 5$, $N = 1$, 14 bit mantissa.

outputs[5]. The MATLAB-implementation of algorithms is scalable and accepts an arbitrary number of channels and taps (any MN) without change in code.

Based on the MATLAB model, the systolic array was drawn as a block diagram in Simulink to break it down into simpler mathematical operations, represented as blocks. Once the Simulink model behavior matched the MATLAB model, a hardware model was made using Xilinx System Generator blocks.

The use of System Generator has helped with the practical implementation of the designed system. Without System Generator a lot of work would be required to manually implement the systolic array, test benches and glue logic in VHDL. Test signals would have to be exported to a format which test benches could import. The biggest hassle with System Generator was the difficulty of building large systolic arrays with many blocks and connections. System Generator did not properly support Simulink buses, a grouping of multiple signals of different type into a single connection (the bus). This made the systolic array schematic slow to modify, and the difference is seen by comparing the number of signal connections in appendix A.2.2 and B.2.2. Although the System Generator model can have λ, σ and mantissa/exponent precision parameters controlled by variables, the block diagram itself cannot be constructed or modified programatically. It is for instance not possible to synthesize an $MN = 10$ array without manually assembling the block diagram. A VHDL implementation could more easily generate different sizes by using the `GENERATE` statement [25,ch11.8].

[5] `timeseries` and `tscollection` were eventually subclassed into object types `CustomSeries` and `CustomCollection` to add functionality for visualization.

4.7 Future work

The main challenge for the current implementation is its speed. By developing the details of a look-ahead transform for a complex-valued QRD-RLS systolic array, the speed can be increased at the expense of more nodes.

By applying channel-interleaving at the same time, existing node logic can be pipelined to account for the nodes introduced by the look-ahead transform.

References

[1] R.A. Monzingo, R.L. Haupt and T.W. Miller. *Introduction to Adaptive Arrays*. SciTech Publishing Inc., 2nd edition, 2011.

[2] H.L. Van Trees. Beamformers. In *Optimum Array Processing: Part IV of Detection, Estimation, and Modulation Theory*, chapter 7, pages 710–916. John Wiley & Sons, Inc., 2002.

[3] B.D. Carlson. Covariance matrix estimation errors and diagonal loading in adaptive arrays. *IEEE Transactions on Aerospace and Electronic Systems*, 24(4):397–401, 1988.

[4] P.S.R. Diniz. *Adaptive Filtering*. Springer US, Boston, MA, 2013.

[5] S. Haykin. *Adaptive Filter Theory*. Pearson, 5th edition, 2014.

[6] D. Bindel, J. Demmel, W. Kahan and O. Marques. On computing givens rotations reliably and efficiently. *ACM Transactions on Mathematical Software*, 28(2):206–238, 2002.

[7] P.K. Meher, J. Valls, K. Sridharan and K. Maharatna. 50 Years of CORDIC: Algorithms, Architectures, and Applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 56(9):1893–1907, 2009.

[8] Q. Gao, L.H. Crockett and R. Stewart. Coarse angle rotation mode CORDIC based single processing element QR-RLS processor. In *Signal Processing Conference, 2009 17th European*, pages 1279–1283, 2009. EUSIPCO.

[9] C.M. Rader. VLSI systolic arrays for adaptive nulling. *IEEE Signal Processing Magazine*, 13(4):29–49, 1996.

[10] Q. Gao and R. Stewart. Improved double angle complex rotation QRD-RLS. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays - FPGA '11*, pages 79, New York, USA, 2011. ACM Press.

[11] A. Maltsev, V. Pestretsov, R. Maslennikov and A. Khoryaev. Triangular systolic array with reduced latency for QR-decomposition of complex matrices. In *2006 IEEE International Symposium on Circuits and Systems*, pages 4, 2006. IEEE.

[12] J.A. Apolinário. *QRD-RLS adaptive filtering*. Springer US, Boston, MA, 2009.

[13] M. Harteneck, R.W. Stewart, J.G. Mcwhirter and I.K. Proudler. Algorithmic engineering applied to the QR-RLS adaptive algorithm. In *Proceedings of 4 th International Conference on Mathematics in Signal Processing*, pages 1–11, 1996.

[14] M. Shoaib, S. Werner, J.A. Apolinário and T.I. Laakso. Solution to the weight extraction problem in fast QR-decomposition RLS algorithms. In *2006 IEEE International Conference on Acoustics Speed and Signal Processing Proceedings*, pages III–572–III–575, 2006. IEEE.

[15] M. Shoaib, S. Werner, J.A. Apolinário and T.I. Laakso. Multichannel fast QR-decomposition RLS algorithms with explicit weight extraction. In *European Signal Processing Conference*, 2006. EUSIPCO.

[16] E. Kile. *Matriseinvertering påFPGA ved hjelp av QR-dekomponering*. Master's thesis, University of Oslo, 2007.

[17] D. Marković and R.W. Brodersen. *DSP Architecture Design Essentials*. Springer US, Boston, MA, 2012.

[18] I. Kollár. MATLAB files for the book "Quantization Noise". Budapest University of Technology and Economics, Dept. of Measurement and Information Systems, 2006 http://oldweb.mit.bme.hu/books/quantization/Matlab-files.html.

[19] C. Ward, P. Hargrave and J. McWhirter. A novel algorithm and architecture for adaptive digital beamforming. *IEEE Transactions on Antennas and Propagation*, 34(3):338–346, 1986.

[20] J. Ma. Annihilation-reordering look-ahead pipelined CORDIC-based RLS adaptive filters and their application to adaptive beamforming. *IEEE Transactions on Signal Processing*, 48(8):2414–2431, 2000.

[21] L. Gao and K.K. Parhi. Hierarchical pipelining and folding of QRD-RLS adaptive filters. In *2000 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.00CH37100)*, pages 3283–3286, 2000. IEEE.

[22] L. Gao and K.K. Parhi. Hierarchical pipelining and folding of QRD-RLS adaptive filters and its application to digital beamforming. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 47(12):1503–1519, 2000.

[23] K.R. Liu, S.F. Hsieh, K. Yao and C.T. Chiu. Dynamic range, stability, and fault-tolerant capability of finite-precision RLS systolic array based on Givens rotations. *IEEE transactions on circuits and systems*, 38(6):625–636, 1991.

[24] Xilinx Inc.. DS816 LogiCORE IP Floating-Point Operator v6.0. Xilinx Inc., 2012 http://www.xilinx.com/support/documentation/ip_documentation/floating_point/v6_0/ds816_floating_point.pdf.

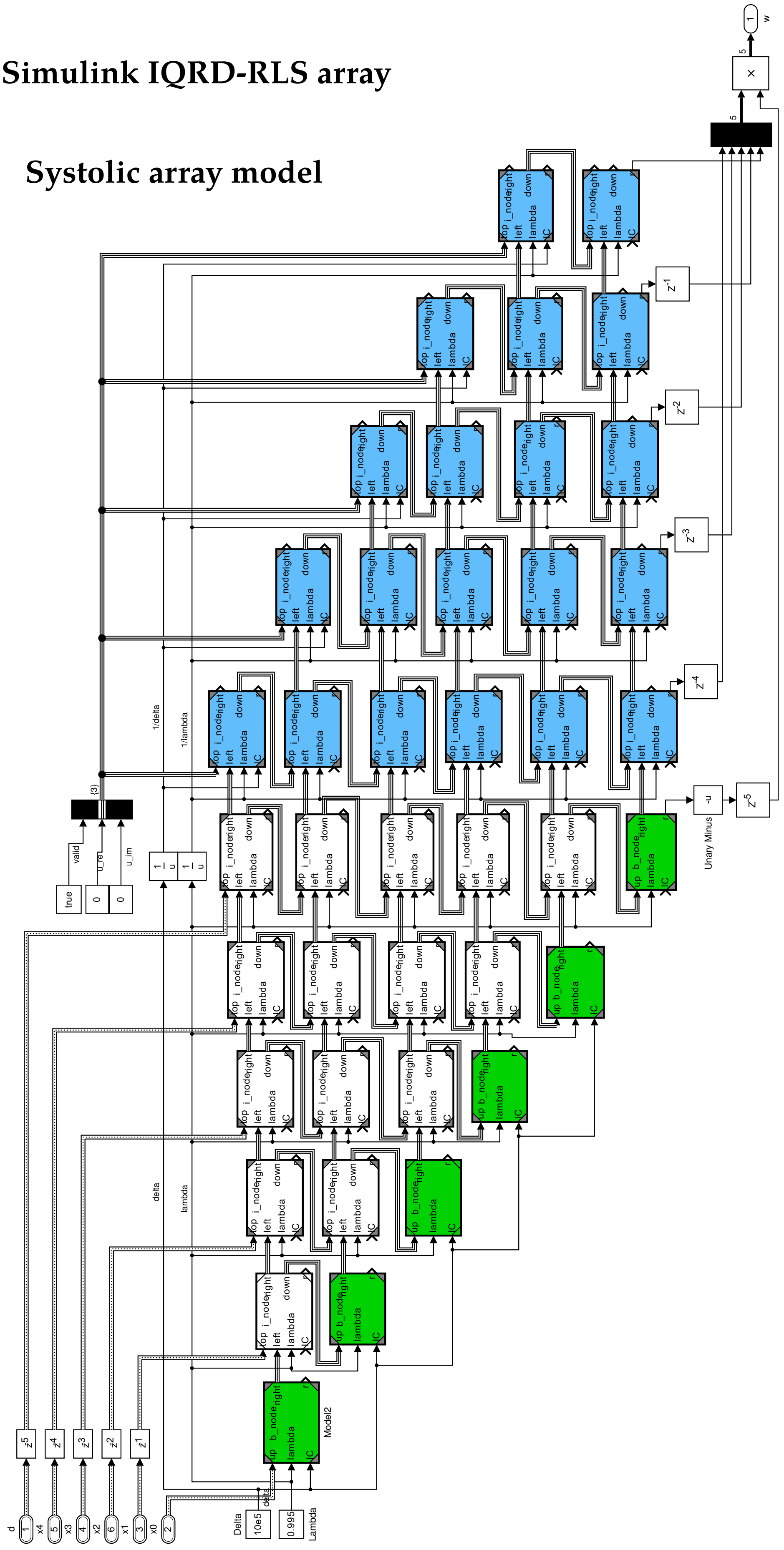
[25] IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pages c1–626, 2009.

[26] Xilinx Inc.. DS249 LogiCORE IP CORDIC v4.0. Xilinx Inc., 2011 http://www.xilinx.com/support/documentation/ip_documentation/cordic_ds249.pdf.

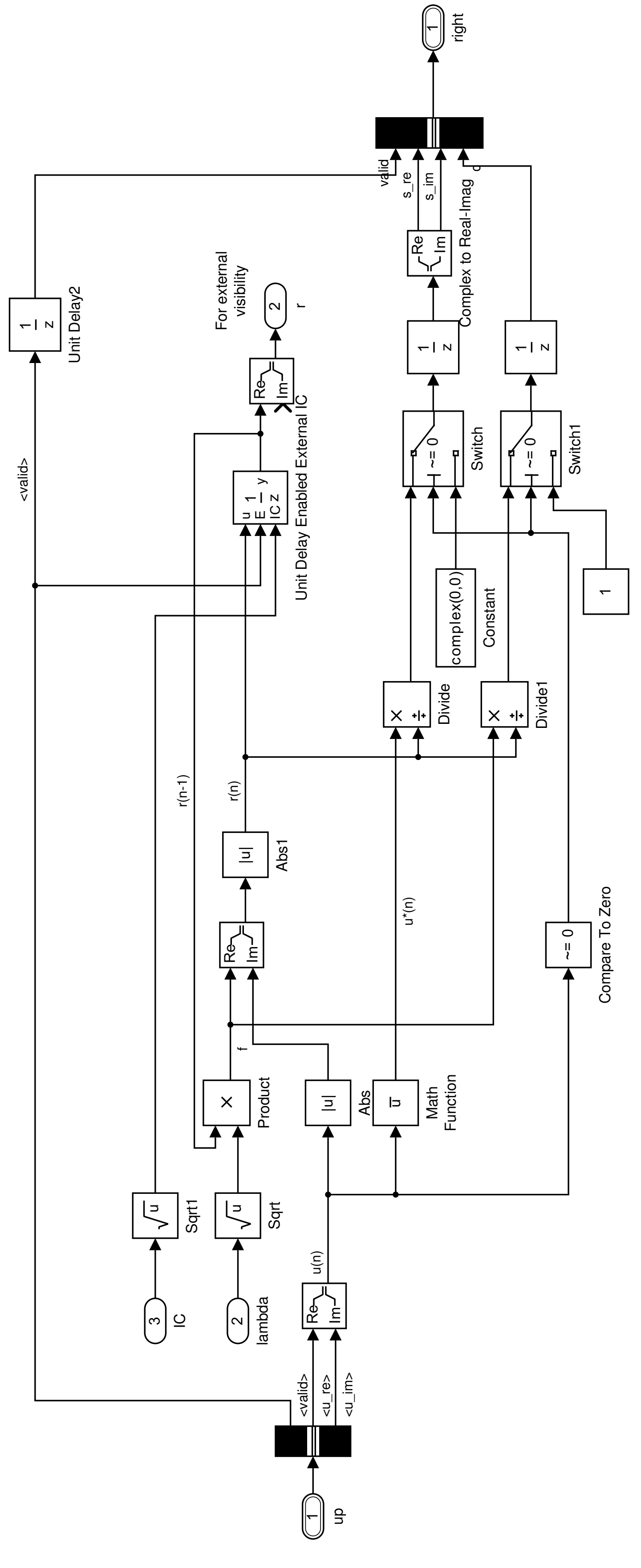
A Simulink models

A.1 Simulink IQRD-RLS array

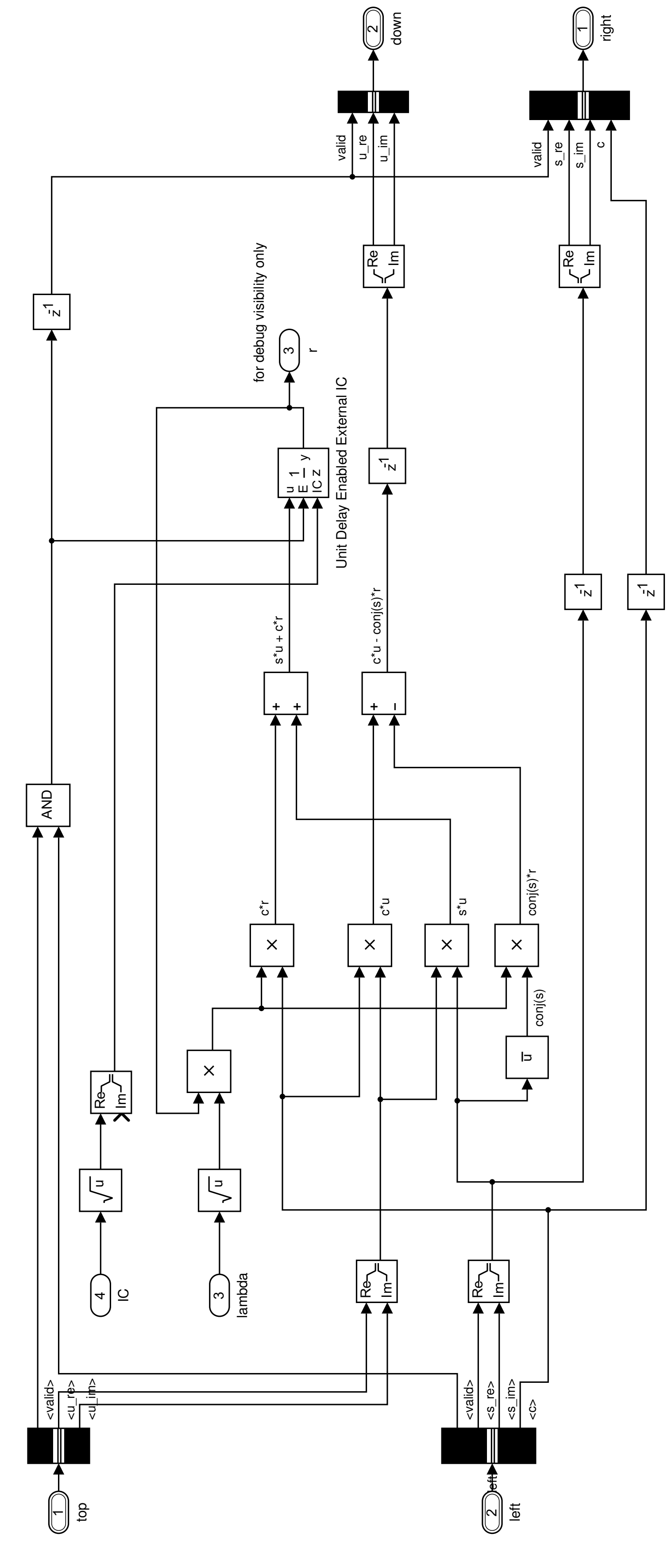
A.1.1 Systolic array model



A.1.2 Boundary node model (b_node)

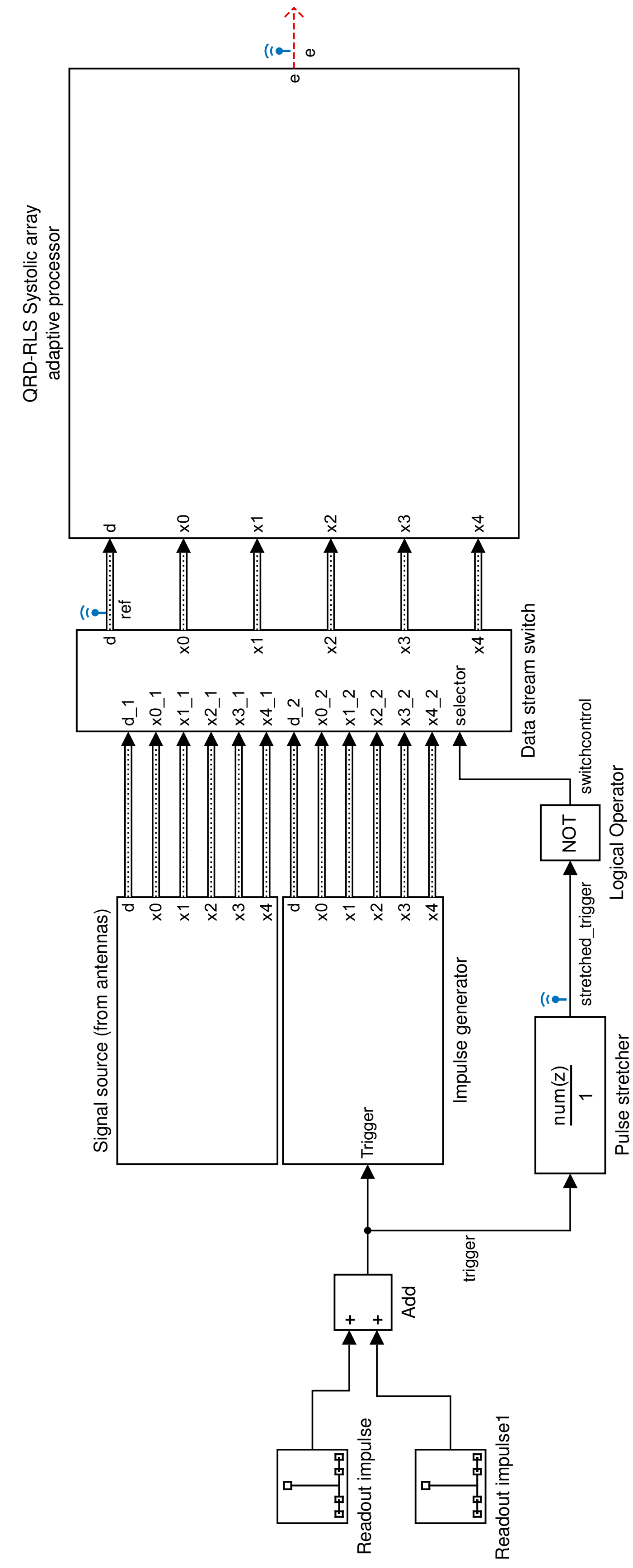


A.1.3 Internal node model (i_node)

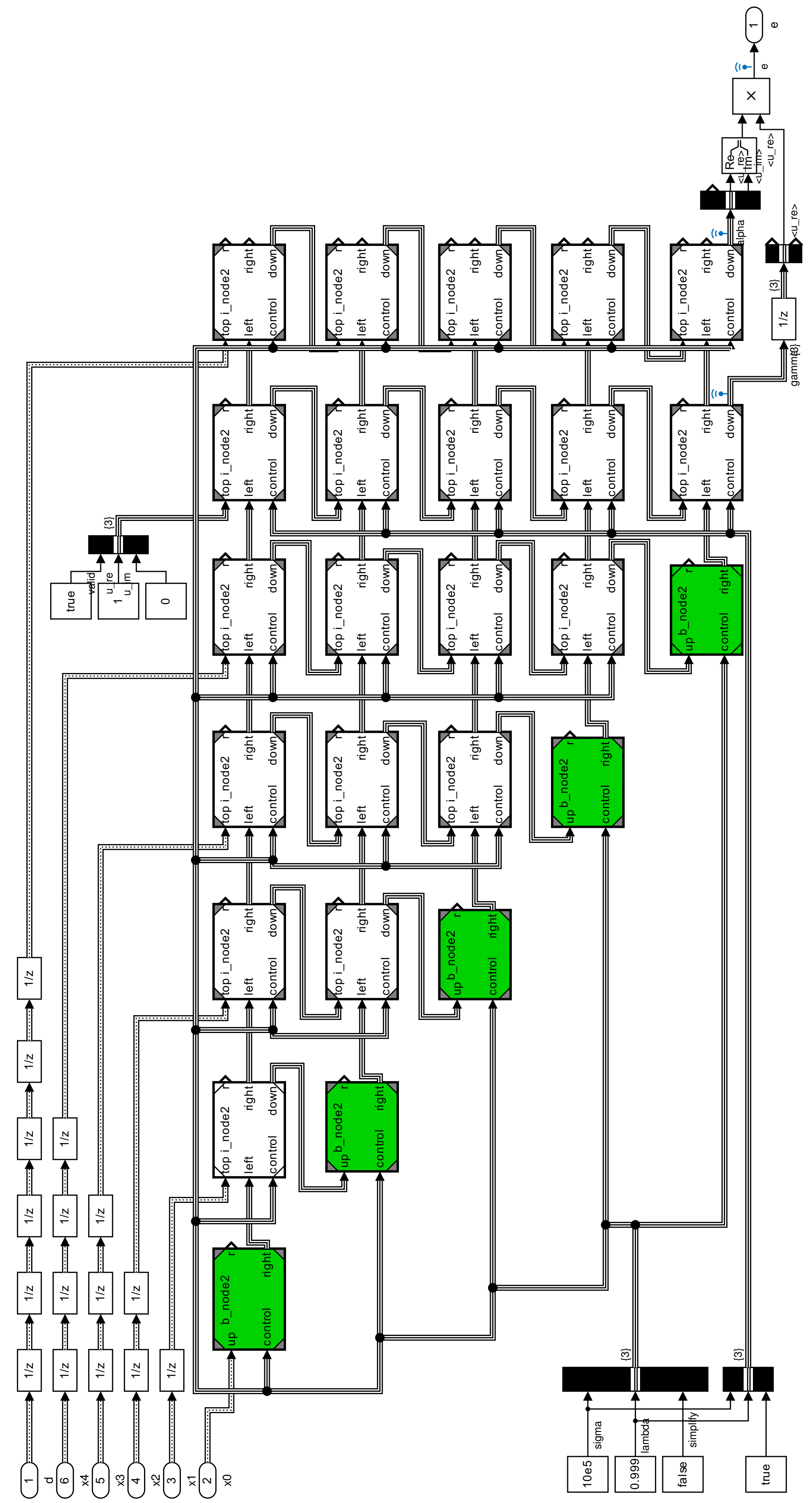


A.2 Simulink QRD-RLS array with weight flushing

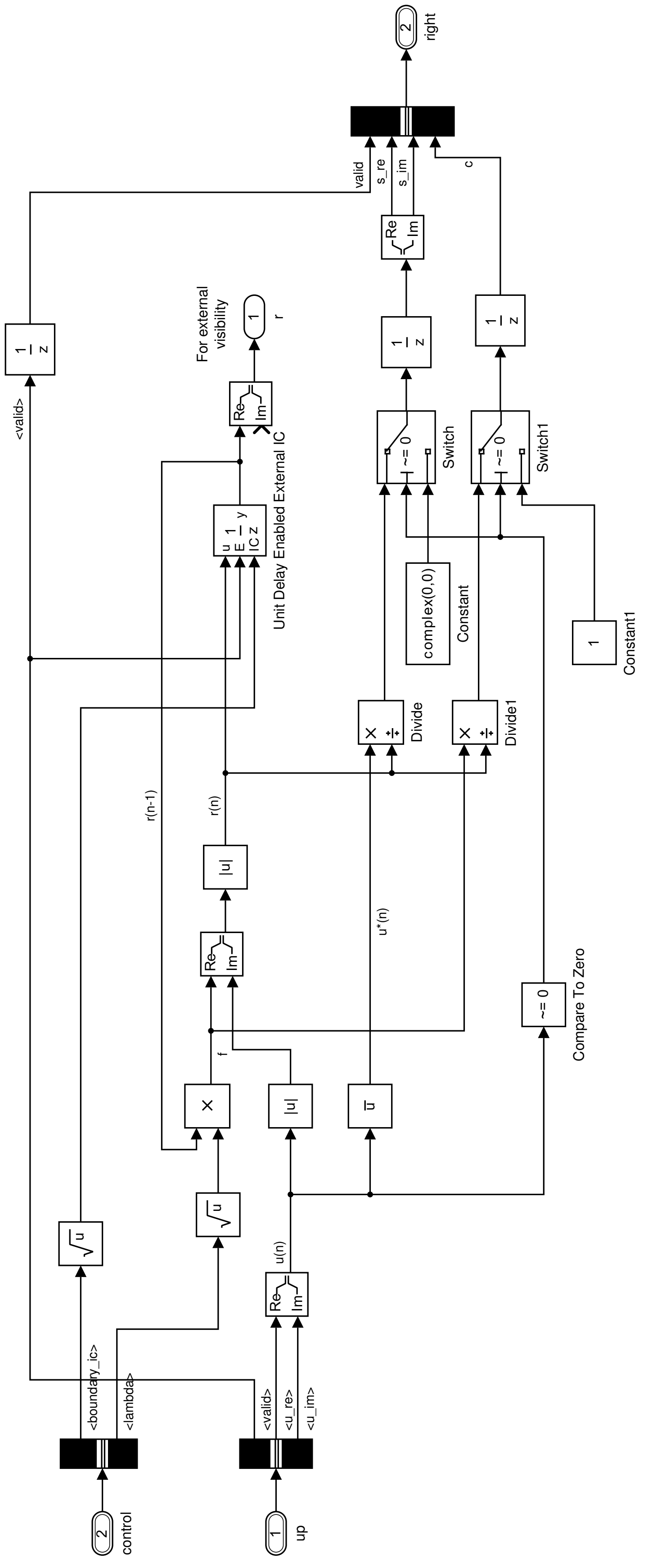
A.2.1 Top level model



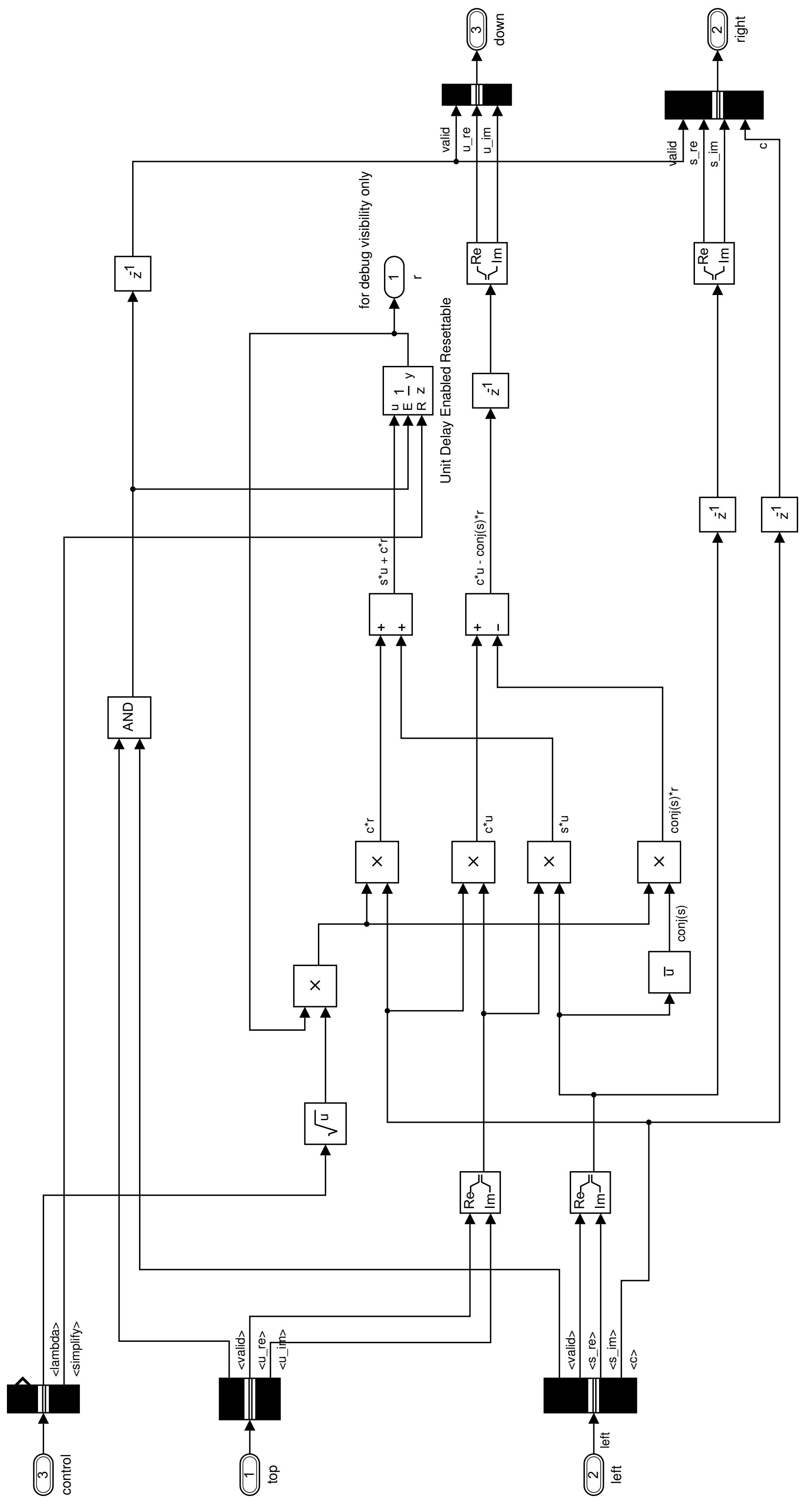
A.2.2 Systolic array model



A.2.3 Boundary node model (b_node2)



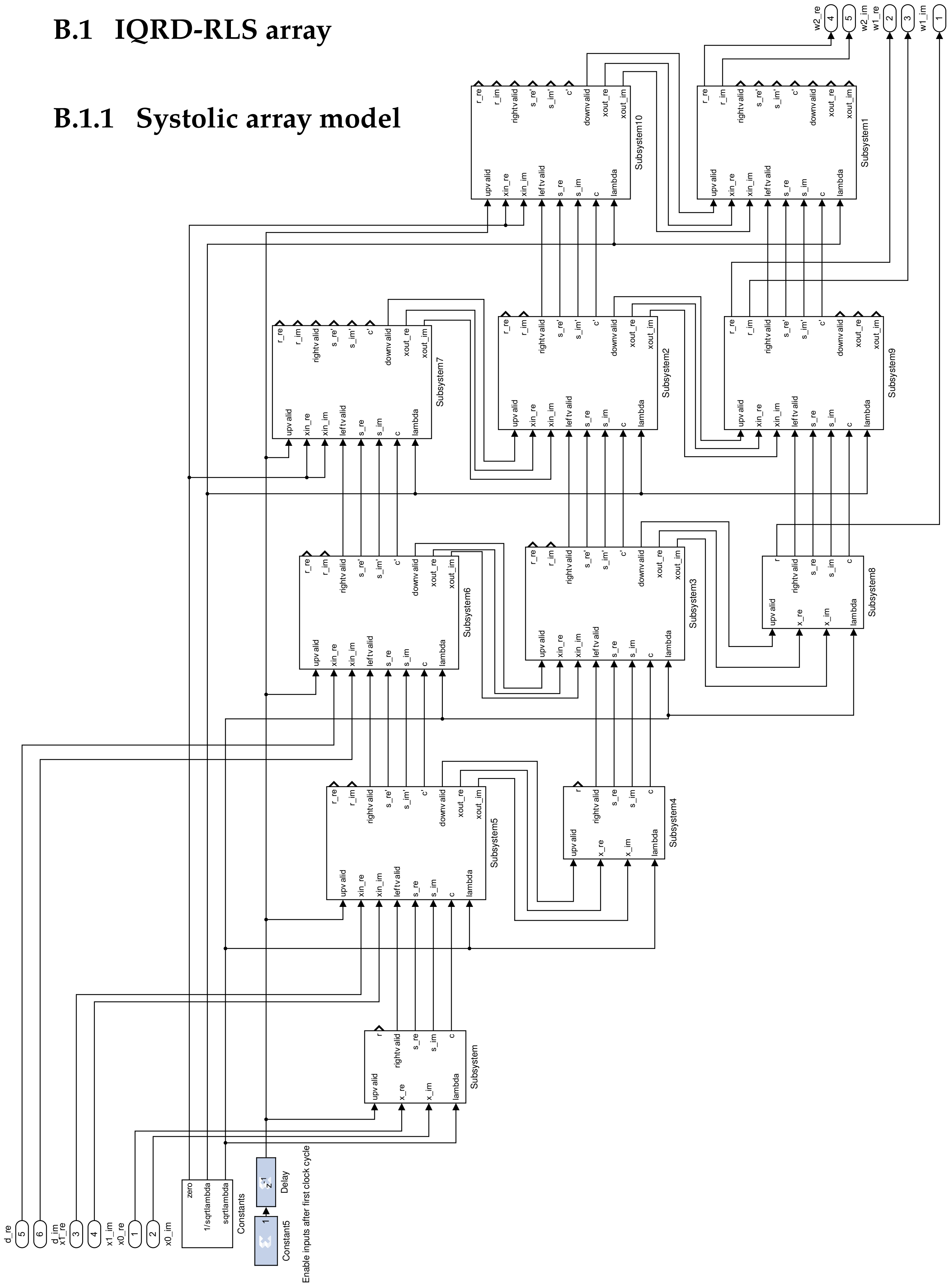
A.2.4 Internal node model (i_node2)



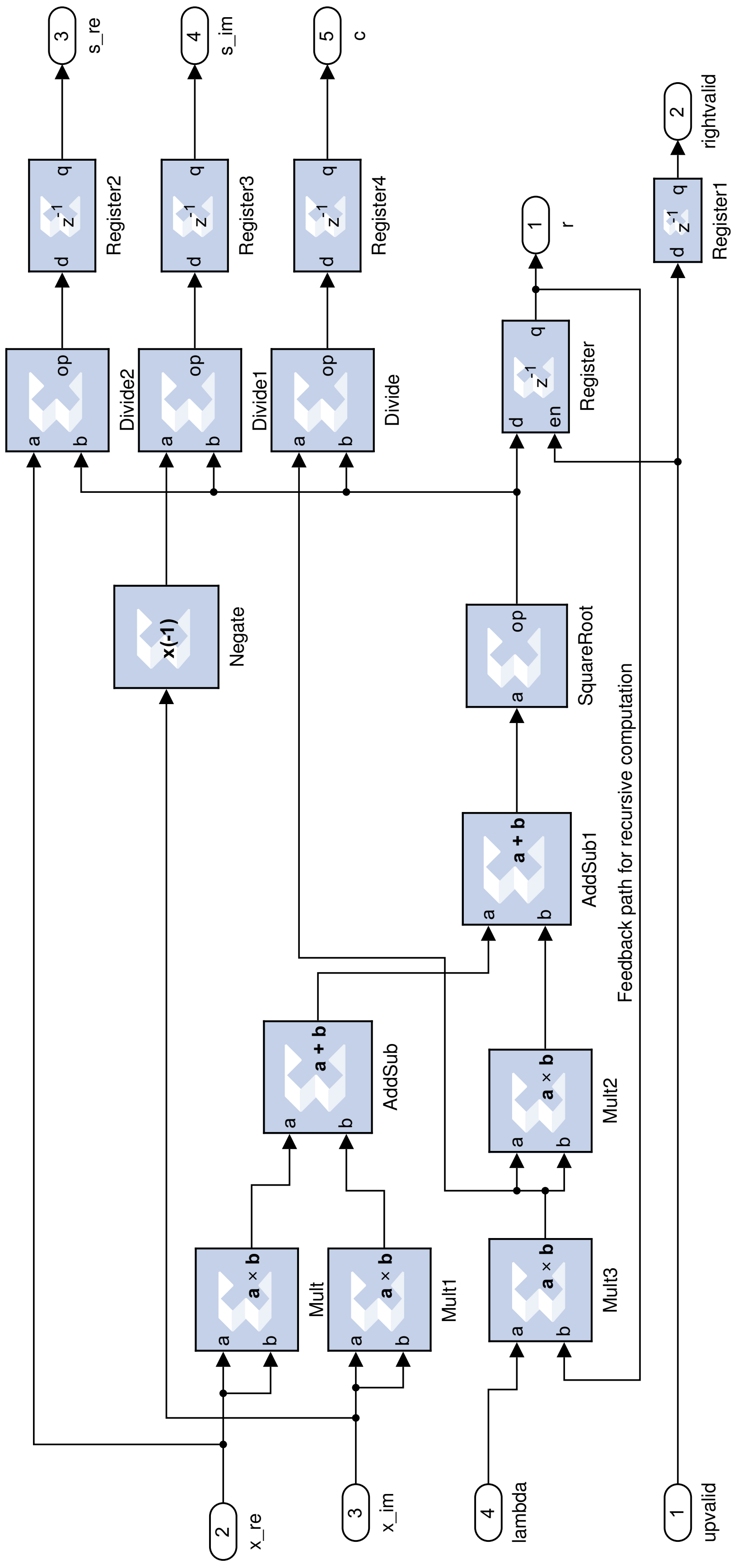
B System Generator (hardware) models

B.1 IQRD-RLS array

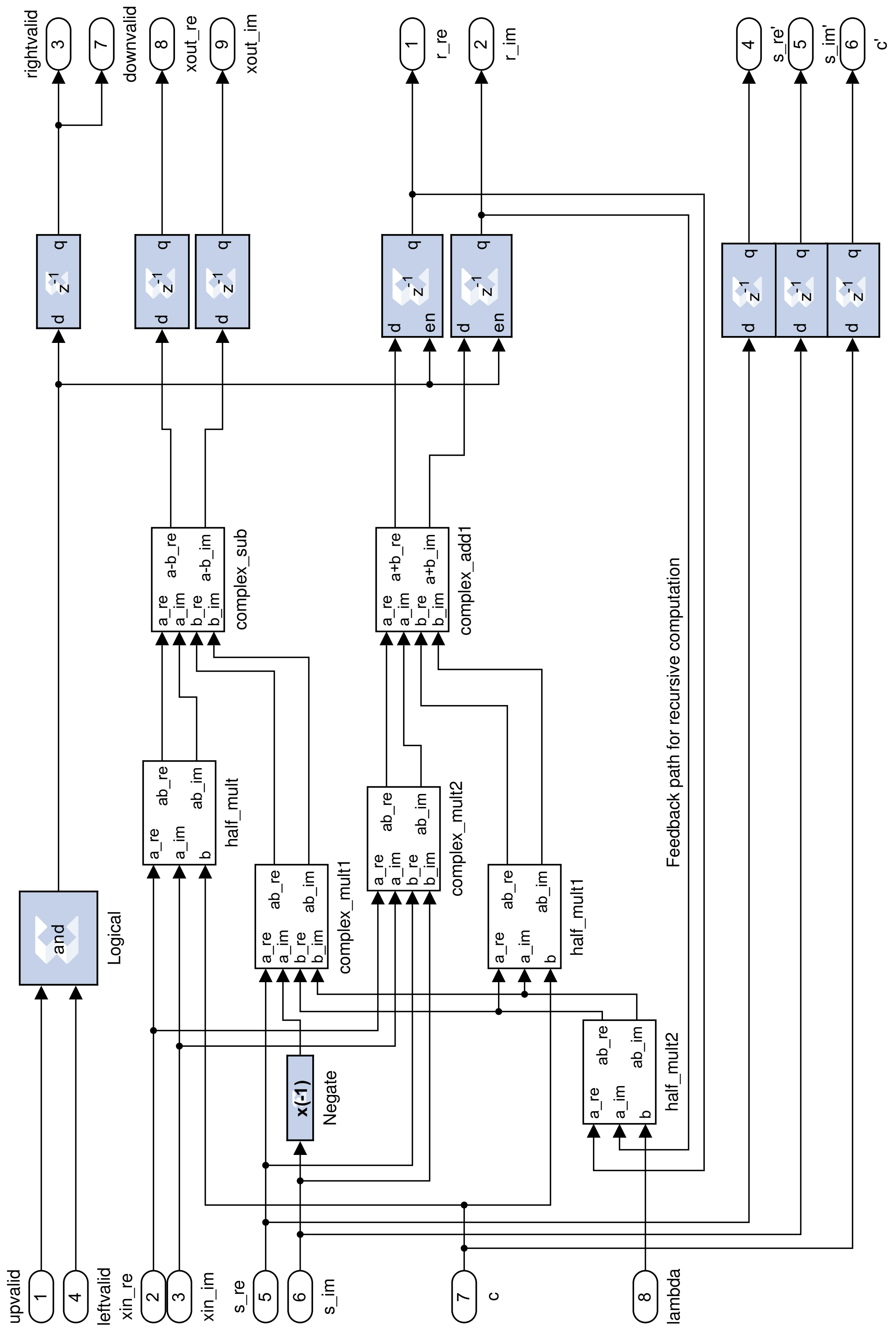
B.1.1 Systolic array model



B.1.2 Boundary node model

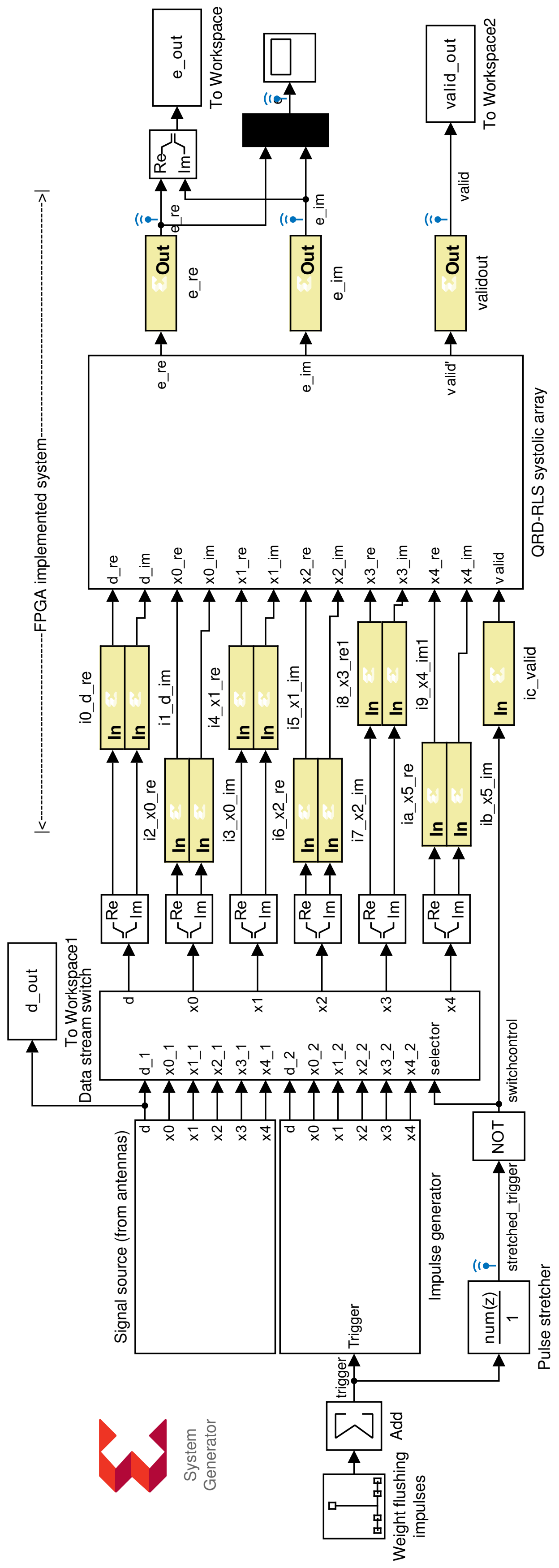


B.1.3 Internal node model

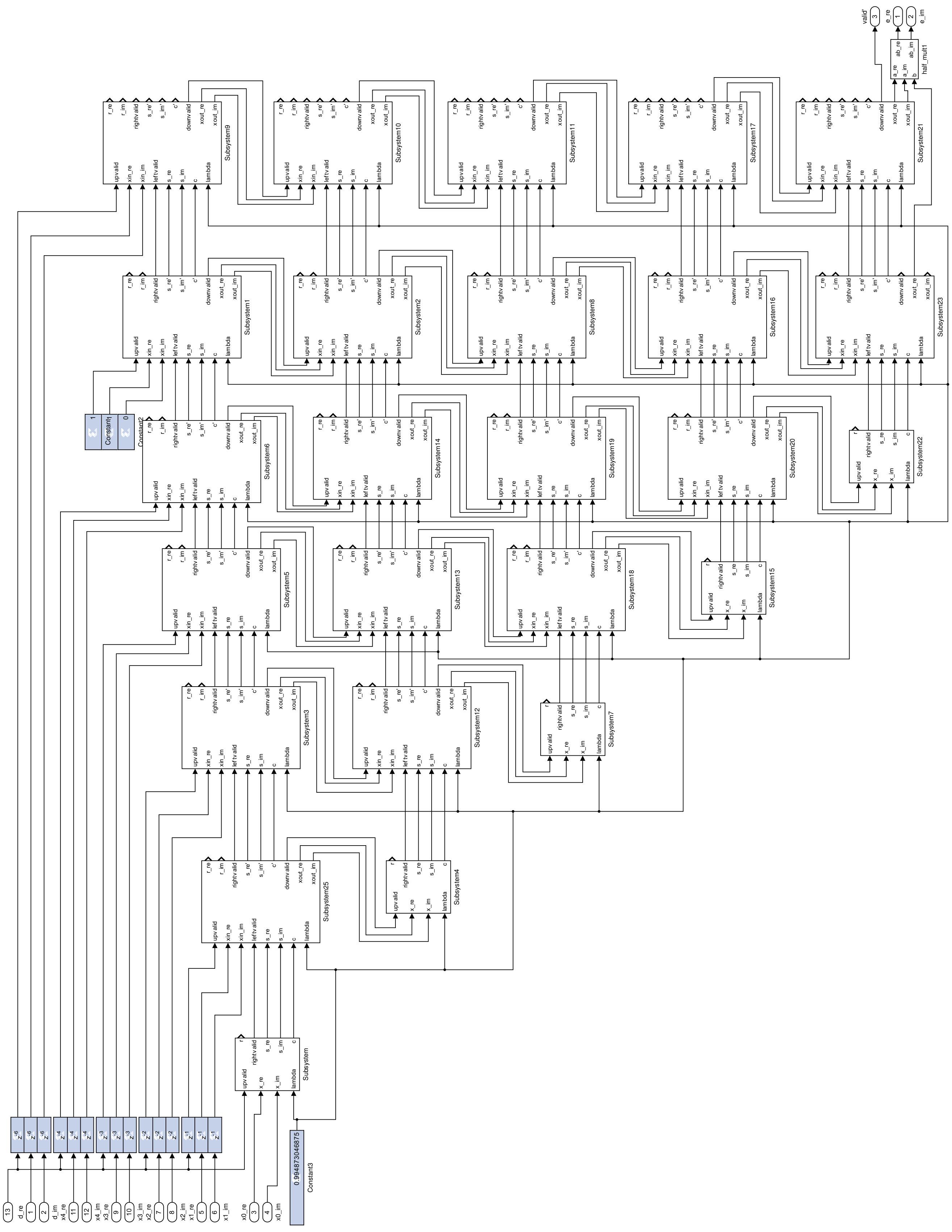


B.2 QRD-RLS array with weight flushing

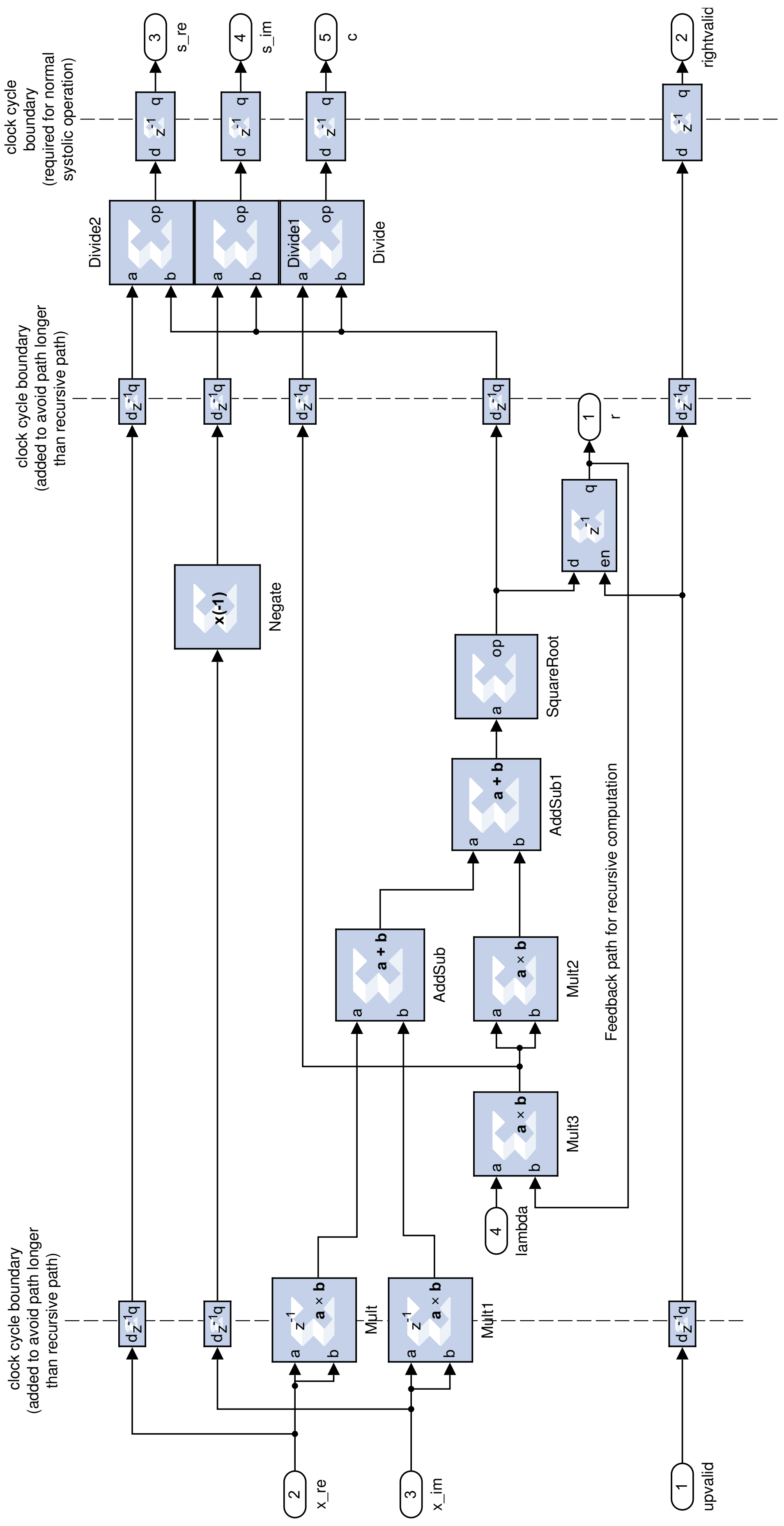
B.2.1 Top level model



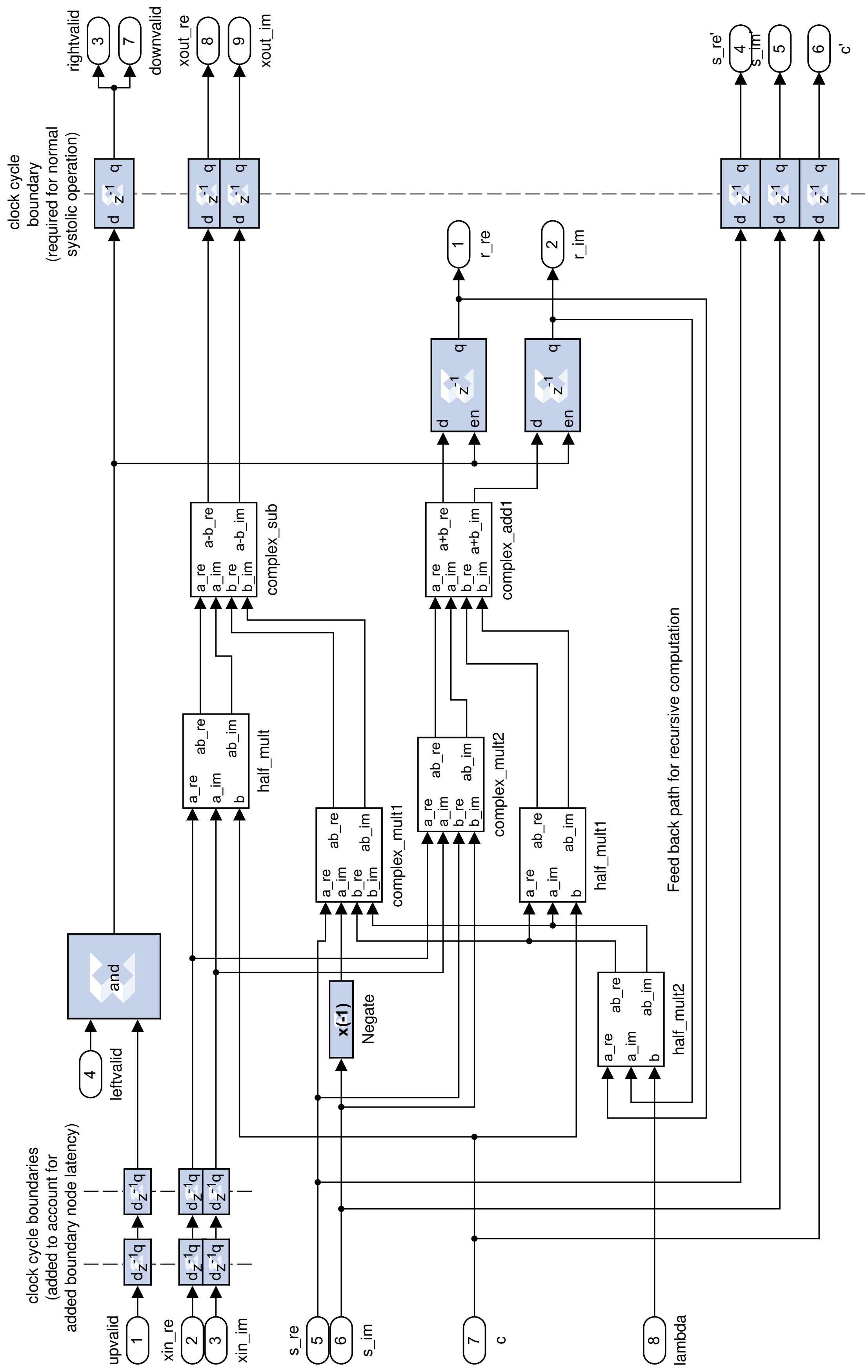
B.2.2 Systolic array model



B.2.3 Boundary node model, pipelined


clock cycle boundary (added to avoid path longer than recursive path)
clock cycle boundary (added to avoid path longer than recursive path)
clock cycle boundary (required for normal systolic operation)
2 x_re
3 x_im
1 upvalid
4 lambda
Mult
Mult1
Mult2
Mult3
AddSub
AddSub1
SquareRoot
Negate
x(-1)
Divide
Divide1
Divide2
Feedback path for recursive computation
1 r
3 s_re
4 s_im
5 c
2 rightvalid

B.2.4 Internal node model, pipelined



rightvalid
3
7
downvalid
xout_re
8
9
xout_im
s_re'
4
s_im'
5
6
c'
clock cycle boundary (required for normal systolic operation)
d z-1 q
1
r_re
2
r_im
d z-1 q en
a_re a_im b_re b_im a-b_re a-b_im
complex_sub
a_re a_im b_re b_im a+b_re a+b_im
complex_add1
Feed back path for recursive computation
a_re a_im b ab_re ab_im
half_mult
a_re a_im b_re b_im ab_re ab_im
complex_mult1
complex_mult2
half_mult1
half_mult2
and
4
leftvalid
x(-1)
Negate
clock cycle boundaries (added to account for added boundary node latency)
d z-1 q
upvalid
1
xin_re
2
3
xin_im
s_re
5
6
s_im
7
c
8
lambda

Note: Internal nodes in array column fed by constants use constant 0 instead of feedback path.

C MATLAB source code

C.1 STAP expansion processing code

```
classdef STAPProcessor < Processor
  methods(Static)
    function [collection] = Process(input, N)
      % PROCESS Expand a collection of M channels into MN channels
      % where additional channels represent temporal taps.
      % Input should be a time series collection with the reference
      % channel named 'REF', and any extra channels named 'CH1',
      % 'CH2' etc. The parameter N selects the number of taps.

      M = input.size(2)-1; % number of channels, excluding reference
      L = input.length();   % length of time vector

      [time, ref, x_in] = unpackcollection(input);

      x_out = zeros(M*N,L);

      for j = N:L   % for each timestep, create a snapshot of all
        for n=1:N % signals in the STAP filter, including current
          % input and N-1 preceding inputs
          x_out((1:M)+(n-1)*M, j) = x_in( 1:M, j-(n-1) );
        end
      end

      collection = tscollection();
      ref_ts = CustomSeries(ref(N:end), time(N:end));
      collection = collection.addts(ref_ts, 'REF');

      for j = 1:M*N
        chan_ts = CustomSeries(x_out(j,N:end), input.Time(N:end));
        collection =collection.addts(chan_ts, sprintf('CH%i',j));
      end
    end
  end
end
```

C.2 SMI processing code

```
classdef SMIProcessor < Processor
  methods (Static)
    function [output_ts, weight_ts] = Process(input, estimate_K, delta)
      % PROCESS Apply sample matrix inversion. Input should be a time
      % series collection with the reference channel named 'REF',
      % and any extra channels named 'CH1', 'CH2' etc.
      % Returns filtered output and weight vector at every sample.
      [time, d, x] = unpackcollection(input);

      NM = input.size(2)-1;
      L = input.length();    % length of time vector

      SampleCor=zeros(NM,NM,L);
      SampleAutoCor=zeros(NM,L);
      Cor=zeros(NM,NM,L);
      AutoCor=zeros(NM,L);

      outdata = zeros(1,L);
      weights = zeros(NM,L);

      for l=1:L % correlation matrices for each time step
        SampleAutoCor(:,l) = x(:,l)*conj(d(l));
        SampleCor(:,:,l) = x(:,l)*x(:,l)';
      end

      % Time-average the correlation matrices. At start of sequence,
      % use any previous samples available, up to K+1 samples
      span_min = -(estimate_K-1); % Desired left/right offsets for
      span_max = 0;        % averaging window
      for l = 1:L
        limited_min = max(l+span_min, 1); % find window indexes
        limited_max = min(l+span_max, L);
        avg_span = limited_min:limited_max;
        AutoCor(:,l) = mean( SampleAutoCor(:, avg_span) ,2);
        Cor(:,:,l) = mean( SampleCor(:,:, avg_span) ,3) + (1/delta)*eye(NM);
      end

      % Apply SMI filter
      for l = 1:L
        weights(:,l) = inv(Cor(:,:,l))*AutoCor(:,l); % R_xx^-1 * r_xd
        outdata(l) = d(l) - weights(:,l)'*x(:,l); % e = d - w^H * x
      end

      output_ts = CustomSeries(outdata, time);
      weight_ts = CustomSeries(weights, time);
    end
  end
end
```

C.3 Conventional RLS processing code

```
classdef RLSProcessor
  methods (Static)
    function [output_ts, weight_ts] = Process(input, lambda, delta)
      % PROCESS Apply conventional RLS filter. Input should be a time
      % series collection with the reference channel named 'REF',
      % and any extra channels named 'CH1', 'CH2' etc.
      % Returns filtered output and weight vector at every sample.
      [time, d_full, x_full] = unpackcollection(input);

      MN = input.size(2)-1; % input width, excluding reference
      L = input.length();   % length of time vector

      outdata = zeros(1,L); weights = zeros(MN,L);
      w = zeros(MN, 1);

      % Algorithm-specific initialization
      lambda1 = lambda^-1;
      P = (delta^-1)*eye(MN);
      for j = 1:L
        d = d_full(j); x = x_full(:,j);
        e = d - w'*x; % a priori error, x filtered with previous filer

        K = (P*x)/(lambda + x'*P*x);

        % Uses: P (previous iteration), x (input), lambda (constant)
        Pnew = lambda1*(P - K*x'*P);

        % Uses w og P (previous iteration), x (input), lambda (constant),
        % d (input, part of e).
        wnew = w + K*conj(e);
        P = Pnew;
        w = wnew;
        weights(:,j) = w;
        outdata(j) = d - w'*x;
      end

      output_ts = CustomSeries(outdata, time);
      weight_ts = CustomSeries(weights, time);
    end
  end
end
```

C.4 Double precision boundary and internal node models

```
function [s_n, c_n, r_n] = boundary_givens(r, xin, sqrtlambda)
  f = sqrtlambda*r;
  g = xin;
  if (g == 0)
    c_n = 1;
    s_n = 0;
    r_n = f;
  elseif (f == 0)
    c_n = 0;
    s_n = sign(conj(g));
    r_n = abs(g);
  else
    c_n = abs(f) / sqrt(abs(f)^2 + abs(g)^2);
    s_n = sign(f)*conj(g) / sqrt(abs(f)^2 + abs(g)^2);
    r_n = sign(f)*sqrt(abs(f)^2 + abs(g)^2);
  end
end


function [s_n, c_n, r_n, xout] = internal_givens(s, c, r, xin, sqrtlambda)
  xout = c * xin - conj(s) * sqrtlambda * r;
  r_n = s * xin + c * sqrtlambda * r;
  s_n = s;
  c_n = c;
end
```

C.5 Limited precision boundary and internal node models

These functions depend on `roundfloat()` by Kollár.

```
function [s_n, c_n, r_n] = boundary_float(r, xin, sqrtlambda)
  q = @(v) roundfloat(v,22); % quantizer

  xin = q(xin);
  sqrtlambda = q(sqrtlambda);
  r = q(r);

  f = q(sqrtlambda*r);
  if (xin == 0)
    c_n = 1;
    s_n = 0;
    r_n = f;
  else
    r_n = sqrt(q( q(f^2) + q( q(real(xin)^2) + q(imag(xin)^2) ) ));
    s_n = q(conj(xin) / r_n);
    c_n = q(f / r_n);
  end
  r_n = q(r_n);
end


function [s_n, c_n, r_n, xout] = internal_float(s, c, r, xin, sqrtlambda)
  q = @(v) roundfloat(v,22); % quantizer

  % s and c already quantized
  xin = q(xin);
  sqrtlambda = q(sqrtlambda);
  r = q(r);

  xout  = c * xin - conj(s) * sqrtlambda * r;
  r_n = s * xin + c * sqrtlambda * r;

  s_n = s;
  c_n = c;
  xout = q(xout);
  r_n = q(r_n);
end
```

C.6 Double precision CORDIC boundary and internal node models

```
function [phi_n, theta_n, r_n] = boundary_cordic(r, xin, sqrtlambda)
  f = sqrtlambda*r;

  % simulate two vectoring CORDICs by finding the magnitudes of the
  % inputs, and the angle between the inputs

  r_n = sqrt(f^2 + abs(xin)^2);
  phi_n = atan2(imag(xin),real(xin));
  theta_n = atan2(abs(xin), f);
end


function [phi_n, theta_n, r_n, xout] = internal_cordic(phi, theta, r, xin,
sqrtlambda)
  f = sqrtlambda*r;

  % simulates the four CORDIC rotators by treating the CORDIC inputs
  % as a complex number, and rotating it by multiplying with exp(i*theta)
  xin = xin * exp(-1i*phi); % phi-CORDIC rotator

  m = complex(real(f), real(xin))*exp(-1i*theta); % theta-CORDIC rotators
  n = complex(imag(f), imag(xin))*exp(-1i*theta); % -"-

  r_n = complex(real(m), real(n));

  xout = complex(imag(m), imag(n))*exp(1i*phi); % reverse phi-CORDIC rotator

  phi_n = phi;
  theta_n = theta;
end
```

C.7 IQRD-RLS systolic array simulator

```
classdef IQRDSystolicProcessor
  methods (Static)
    function [output_ts, weight_ts] = Process(input,lambda,delta,nodetype)
      % PROCESS Simulate IQRD-RLS systolic array. Input should be a time
      % series collection with the reference channel named ’REF’,
      % and any extra channels named ’CH1’, ’CH2’ etc.
      % Returns filtered output and weight vector at every sample.
      [time, d, x] = unpackcollection(input);

      MN = input.size(2)-1; % input width, excluding reference
      L = input.length();   % length of time vector

      state_struct = struct( ...
        ’node’,zeros(MN+1, MN+2),... % stores r values in recursion
        ’ready’,false(MN+1, MN+2),... % nodes that are ready
        ’c’,ones(MN+1),... % rotation parameters between nodes
        ’s’,zeros(MN+1),...
        ’xin’,zeros(MN+1, MN+2)... % inputs from above, for each node
      );
      state_struct.node(1:MN,1) = sqrt(delta); % init r11, r22 etc.
      state_struct.node(1:MN,end) = 1/sqrt(delta); % init rm11, rm22 etc.
      cs = state_struct;
      ns = cs; % use next state ns and current state cs

      ns.ready(1,1) = true; % mark r11 ready, only depends on x0 input
      ns.ready(MN+1,MN+2) = true; % mark 1/gamma as ready. node not used
      outputrowhist = zeros(MN+1, L);
      whist = zeros(MN, L);
      outputhist = zeros(1, L);

      sqrtlambda = sqrt(lambda); % may be precomputed
      invsqrtlambda = 1/sqrtlambda;
      boundary = str2func(strcat(’boundary_’, nodetype));
      internal = str2func(strcat(’internal_’, nodetype));

      input = [x; d];
      %%
      h = waitbar(0,");
      for l = 1:L
        if (mod(l,25) == 0)
          waitbar(l/L,h,sprintf(’IQRD-RLS Structure simulation: %02.0f%%’,100*l/L))
        end
        cs = ns;

        % feed new elements into the top of the xin matrix. We read a
        % time-shifted slice of the input matrix.
        invector = zeros(1, MN+1);
        for j = 1:MN+1
          if (l-(j-1) < 1)
      break; % during initialization, not all values are ready
    end
    invector(j) = input(j, l-(j-1));
  end
  invector = round(invector);
  cs.xin(1,1:MN+1) = invector;

  cs.xin(:,end) = 0; % zero rightmost column for clarity
  % not really needed since nothing else writes this column

  for j = 1:size(cs.node, 1) % visit each row..
    for k = 1:size(cs.node, 2) % visit each node in that row

      if cs.ready(j,k) == true

        % find type of node and evaluate the node
        if k == 1 % leftmost row, only boundary nodes.
          %fprintf('Boundary node %i,%i---\n', j,k)
          [ns.s(j,k), ns.c(j,k), ns.node(j,k)] =  ...
           boundary(cs.node(j,k),cs.xin(j,k),sqrtlambda);

          % mark node to the right as ready
          ns.ready(j,k+1) = true;
        elseif j > MN + 1 -(k-1) % inverse internal nodes
          %fprintf('Inv internal node %i,%i---\n', j,k)
          % upper left triangle, internal nodes. Produces outputs
          % that are indexed down and to the left so that that the
          % receiving node reads it at its own index.
          [ns.s(j,k), ns.c(j,k), ns.node(j,k), ns.xin(j+1,k-1)] = ...
            internal(cs.s(j,k-1), cs.c(j,k-1), cs.node(j,k),...
            cs.xin(j,k), invsqrtlambda);

          % propagate readiness, if possible
          if j+1 <= size(ns.ready,1)
            ns.ready(j+1,k-1) = true; % node below and to left
          end
          if k+1 <= size(ns.ready, 2)
            ns.ready(j,k+1) = true; % node to the right
          end
        else
          %fprintf('Internal node %i,%i---\n', j,k)
          % same as inverse nodes, just a different node
          [ns.s(j,k),ns.c(j,k),ns.node(j,k),ns.xin(j+1,k-1)] = ...
            internal(cs.s(j,k-1),cs.c(j,k-1),cs.node(j,k),...
            cs.xin(j,k), sqrtlambda);

          % propagate readiness
          ns.ready(j+1,k-1) = true; % below and to left
          ns.ready(j,k+1) = true; % to the right
        end
```

```
          end

        end
      end
      gamma = ns.node(MN+1, 1);
      wtilde = ns.node(MN+1, 2:(end-1));
      outputrowhist(:,l) = [gamma wtilde];

      w = zeros(MN,1);
      if cs.ready% only generate weigths once all nodes are initialized
        gamma = outputrowhist(1,l-MN);
        for j = 1:MN
          w(j) = -gamma*outputrowhist(1+j, l-MN+j);
        end
      end
      whist(:,l) = w;
      latency = l-(3*MN + 2 - 1);
      if ((l-latency) >= 1)
        outputhist(l) = d(l-latency) - w'*x(:,l-latency);
      else
        outputhist(l) = 0;
      end
    end
    close(h);

    % outputs are ready in following cycle, so add 1 to time
    weight_ts = CustomSeries(whist, time+1);
    output_ts = CustomSeries(outputhist, time+1);
  end
 end
end
```

C.8 QRD-RLS systolic array simulator

```
classdef QRDSystolicProcessor
  methods (Static)
    function [output_ts]=Process(input,lambda,delta,nodetype)
      % PROCESS Simulate QRD-RLS systolic array. Input should be a time
      % series collection with the reference channel named 'REF',
      % and any extra channels named 'CH1', 'CH2' etc.
      % Parameter 'nodetype' is used to choose node simulation
      % function. If 'givens' is selected, the simulator will use
      % boundary_givens() and internal_givens() functions.
      % Returns filtered output.
      [time, d, x] = unpackcollection(input);

      MN = input.size(2)-1; % input width, excluding reference
      L = input.length();   % length of time vector

      state_struct = struct( ...
        'node', zeros(MN, MN+2),... % store r values in recursion
        'ready', false(MN, MN+2), ... % nodes that are ready
        'c', zeros(MN), ... % rotation parameters between nodes
        's', zeros(MN), ...
        'xin', zeros(MN+1, MN+2) ... % inputs from above, for each node.
        ... % xin is 1 row larger because we are interested in the
        ... % xin's for the MN+1 row (i.e. outputs from the last MN row)
      );
      state_struct.node(1:MN,1) = sqrt(delta); % initialize r11, r22 etc.
      cs = state_struct;
      ns = cs; % use next state ns and current state cs

      ns.ready(1,1) = true; % r11 is ready, it only depends on x0 input

      outputhist = zeros(1, L);

      sqrtlambda = sqrt(lambda); % may be precomputed
      boundary = str2func(strcat('boundary_', nodetype));
      internal = str2func(strcat('internal_', nodetype));

      input = [x;
        ones(1, length(d));
        d];
      %%
      h = waitbar(0,'');
      for l = 1:L
        if (mod(l,25) == 0)
          waitbar(l/L,h,sprintf('IQRD-RLS Structure simulation: %02.0f%%',100*l/L))
        end
        cs = ns;

        % feed new elements into the top of the xin matrix. We read a
        % time-shifted slice of the input matrix
  invector = zeros(1, MN+2);
  for j = 1:MN+2
    if (l-(j-1) < 1)
      break; % during initialization, not all values are ready
    end
    invector(j) = input(j, l-(j-1));
  end
  invector = round(invector);
  cs.xin(1,1:MN+2) = invector;

  %fprintf('\nIteration start: %i\n',l);
  for j = 1:size(cs.node, 1) % visit each row..
    for k = 1:size(cs.node, 2) % visit each node in that row

      if cs.ready(j,k) == true

        % find type of node and evaluate the node
        if k == 1 % leftmost row, only boundary nodes.
          %fprintf('Boundary node %i,%i---\n', j,k)
          [ns.s(j,k), ns.c(j,k), ns.node(j,k)] = ...
           boundary(cs.node(j,k),cs.xin(j,k),sqrtlambda);

          % mark node to the right as ready
          ns.ready(j,k+1) = true;
        else
          %fprintf('Internal node %i,%i---\n', j,k)
          % same as inverse nodes, just a different node
          [ns.s(j,k),ns.c(j,k),ns.node(j,k),ns.xin(j+1,k-1)] = ...
            internal(cs.s(j,k-1), cs.c(j,k-1), cs.node(j,k),...
            cs.xin(j,k), sqrtlambda);

          % propagate readiness
          ns.ready(j+1,k-1) = true; % below and to left
          ns.ready(j,k+1) = true; % to the right
        end
      end

    end

    % in each row, zero the node feedback for the column
    % computing gamma
    column = (MN+1):-1:1;
    ns.node(j, column(j)) = 0;
  end % all rows and columns processed

  outputhist(:,l) = real(ns.xin(MN+1,1)).*(ns.xin(MN+1, 2));
end
close(h);

% outputs are ready in following cycle, so add 1 to time.
```

```
      % this structure does not create a weight output
      output_ts = CustomSeries(outputhist, time);
    end
  end
end
```

C.9 Miscellaneous utility functions, dependencies

```
function [time, ref, x] = unpackcollection(collection)
  % UNPACKCOLLECTION take a timeseries collection containing channels
  % 'REF', 'CH1', 'CH2' etc., and place the contents into matlab arrays
  % for the time vector, the reference signal and the M channels.
  % For input of duration L samples, the output dimensions are;
  % time = 1 row,  L columns
  % ref  = 1 row,  L columns
  %   x    = M rows, L columns
    M = collection.size(2)-1; % number of channels, excluding reference
    L = collection.length();   % length of time vector

    ref = collection.get('REF').Data(1,:);
    x = zeros(M,L);

    j = 1;
    for channel = collection.gettimeseriesnames
        if (strcmp(channel,'REF'))
            continue % skip it
        end
        x(j, :) = collection.get(channel).Data(1,:);
        j = j +1;
    end
    time = collection.Time;
end
```

D Other

D.1 Breakdown of conventional RLS complexity

Table D.1 Breakdown of conventional RLS arithmetic operations

Result	Sub-expression	Type	MUL	ADD	DIV
1	$\boldsymbol{P}[i-1]\boldsymbol{x}[i]$	$q \times 1, \mathbb{C}$	q^2	$q(q-1)$	0
2	$\lambda + \boldsymbol{x}^H[i]$ (Result 1)	$1 \times 1, \mathbb{R}$	q	q	0
3	$\boldsymbol{x}^H[i]\boldsymbol{P}[i-1]$	$1 \times q, \mathbb{C}$	hermitian of Result 1		
4	$K[i] = \frac{\text{Result1}}{\text{Result2}}$	$q \times 1, \mathbb{C}$	0	0	1
5	$K[i] \cdot$ (Result 3)	$q \times 1, \mathbb{C}$	q^2	0	0
6	$P[i] = \frac{1}{\lambda}(P[i-1] - \text{Result 5})$	$q \times q, \mathbb{C}$	q^2	q^2	0
7	$\epsilon[i] = \boldsymbol{x}^H[i]\boldsymbol{w}'[i-1] - d[i]$	$1 \times 1, \mathbb{C}$	q	q	0
8	$\boldsymbol{w}'[i] = \boldsymbol{w}'[i-1] - \boldsymbol{K}[i]\epsilon[i]$	$q \times 1, \mathbb{C}$	q	q	0

D.2 Implementation of CORDIC-based QRD-RLS

The comparison made in section 3.6.2 to initially choose between floating point and CORDIC (fixed point), was simplified in that it compared 16 bit fixed point to 16 bit floating point. At that level the implementations were predicted to be similar in the resource/performance trade-off. In this section, a CORDIC based fixed point implementation of QRD-RLS is detailed. References used are listed separately at the end of this appendix, and not in the main reference list for the thesis.

Fixed-point design is more challenging because of the reduced dynamic range compared to floating point. Numbers must be scaled to maintain precision and avoid overflow. Of particular interest is the recursive variable r in the systolic array nodes. According to algorithm 2.5, $r = \sqrt{\lambda r^2 + |x_{in}|^2}$, that is the norm of $[\lambda^{1/2} r, x_{in}]^T$. It is clear that r can grow and potentially overflow depending on the value of λ and x_{in}, because of this recursive equation. One article [1] describes several properties of the systolic array, such as a bound for the r variable in the first row of the array;

$$\lim_{k \to \infty} |r_{1j}[k]| \leq \frac{|x_{in,max}|}{\sqrt{1-\lambda^2}} \triangleq \mathbb{R} \tag{D.1}$$

Then, the worst case bound for the r variable in subsequent rows is;

$$\lim_{k\to\infty} |r_{mj}[k]| \leq (2\lambda)^{m-1} \cdot \mathbb{R} \tag{D.2}$$

Where $\mathbb{R}$ represents the maximum dynamic range for the first row. Since λ in this particular application is chosen close to 1 (between 0.99 to 0.999), it is observed from equation (D.2) that r should increase by one bit for every row to avoid overflow. The number of rows in the array is equal to MN.

If the input data with real and imaginary parts is in the range $[-2^{15}, 2^{15}-1]$ (16 bits) and $\lambda \in \{0.99, 0.995, 0.999\}$ we get the number of bits required is: $\log_2(\mathbb{R}) \in \{19, 19, 20\}$ bits for the last row.

For the implementation, the CORDIC construction from figure 3.10 was initially simulated in MATLAB using an ideal model of CORDIC (appendix C.6) in the same simulation script used for the floating point Givens implementation. It proved to be difficult to get the internal nodes to provide the expected results of a Givens rotation, so a fourth CORDIC rotating element was added as shown in the bottom right of figure D.1. It is not clear as to how the implementations of Rader[2] and Gao[3] can perform complex Givens rotation without this fourth rotation.

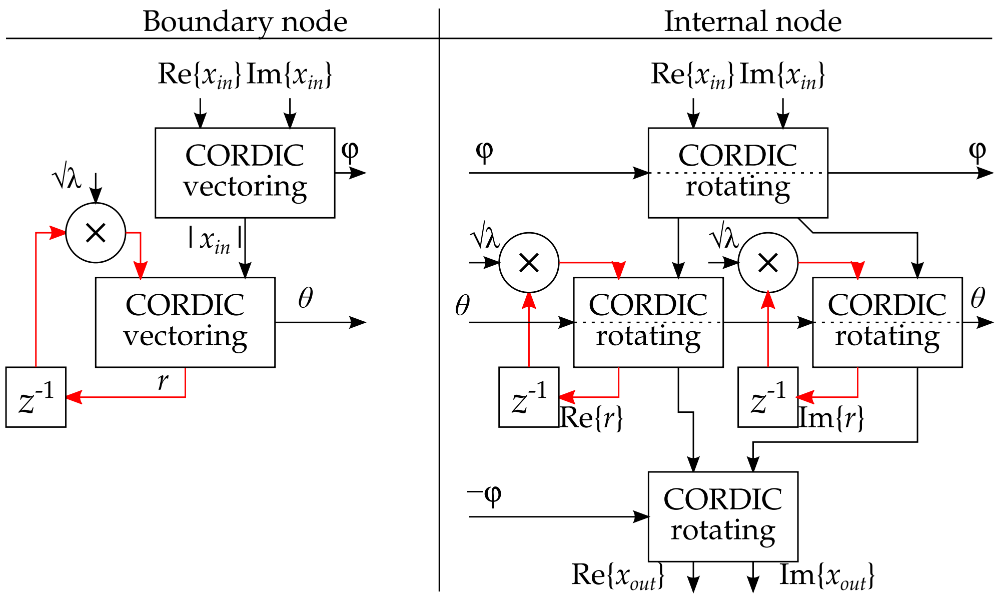


Figure D.1 Boundary and internal nodes for CORDIC based systolic array.

CORDIC IP cores provided by Xilinx were used[4]. System Generator blocks for these cores were available, but did not allow input and output registers to be disabled for purely combinatorial operation. Instead, the core had to be correctly configured using Xilinx Core Generator, and the core was imported as a “black box” in System Generator. Specific to the Xilinx CORDIC implementation is that inputs are normalized to $[-1, 1]$, phase angle is normalized to $[-\pi, \pi]$, and outputs have range $[-\sqrt{2}, \sqrt{2}]$ in rotating mode and $[0, \sqrt{2}]$ in vectoring mode.

For use with the CORDIC IP cores, the test signals are scaled from a 16 bit integer range $[-2^{15}, 2^{15}-1]$ to a fixed point range within $[-1, 1]$. The multiplication used for this purpose is referred to as the *gain*. The scale factor and the number of bits must be selected so that;

- r does not overflow for the selected λ
- Computed filter weights are accurate enough to cancel the jammer

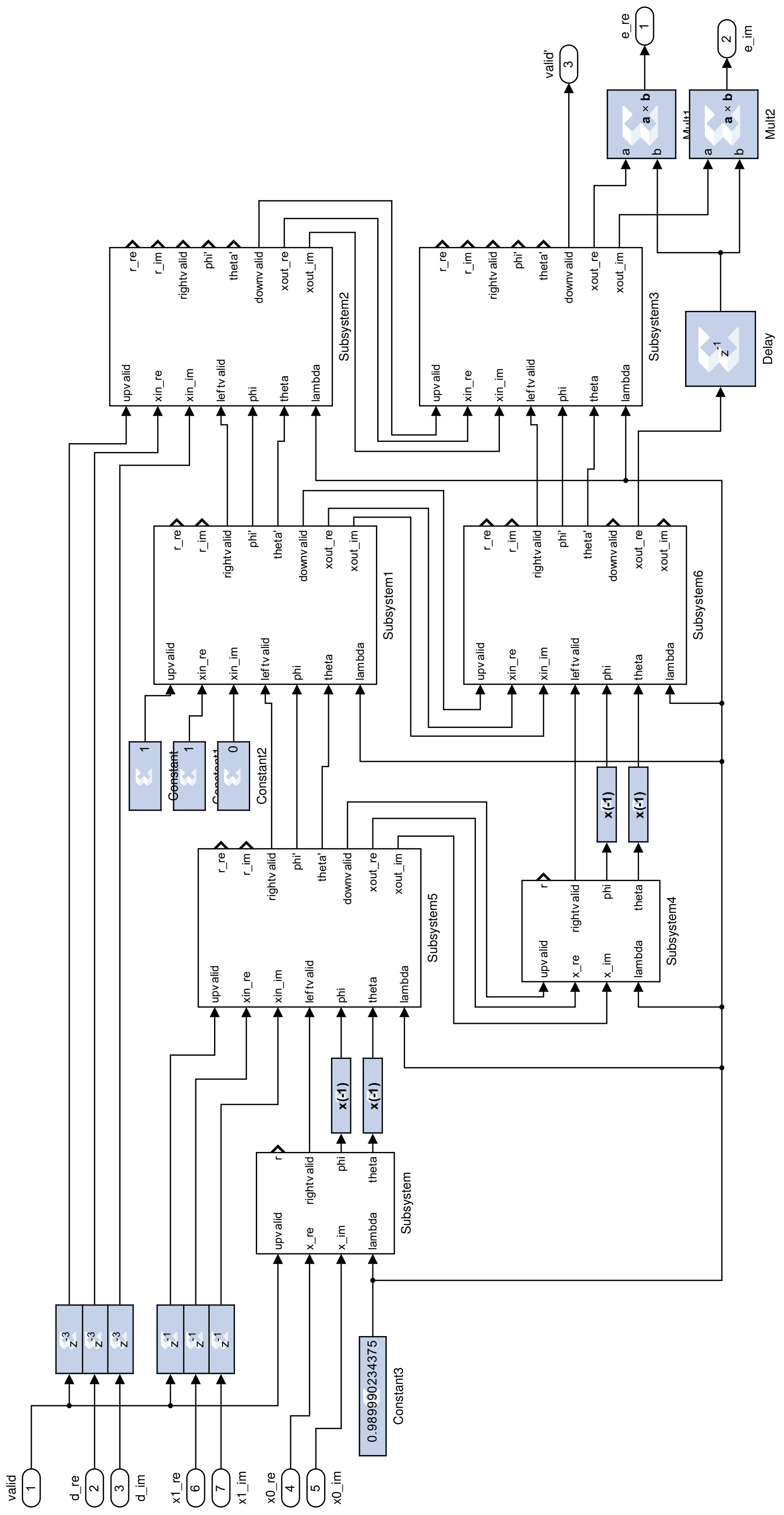
The space of possible solutions for a fixed-point design is very large, and the CORDIC cores within the same internal or boundary node could have different precisions. To reduce the solution space, it was decided to use the same precision in all CORDIC core such that every node is identical. The precision is chosen so that overflow is avoided with the test signal with largest range.

Since the test signal with largest range was the broadband signal (approximate range $[-15000, 15000]$), this was selected as the range of x_{in} instead of the wider range $[-2^{15}, 2^{15}-1]$. To get a precision of 14 bits (which proved a decent starting point for floating point mantissa precision), the gain was set to $1/60000$. The ranges for the broadband and multitone signal become approximately $[-0.25, 0.25]$ and $[-0.1, 0.1]$ respectively, at the input of the array. λ was selected to be 0.99.

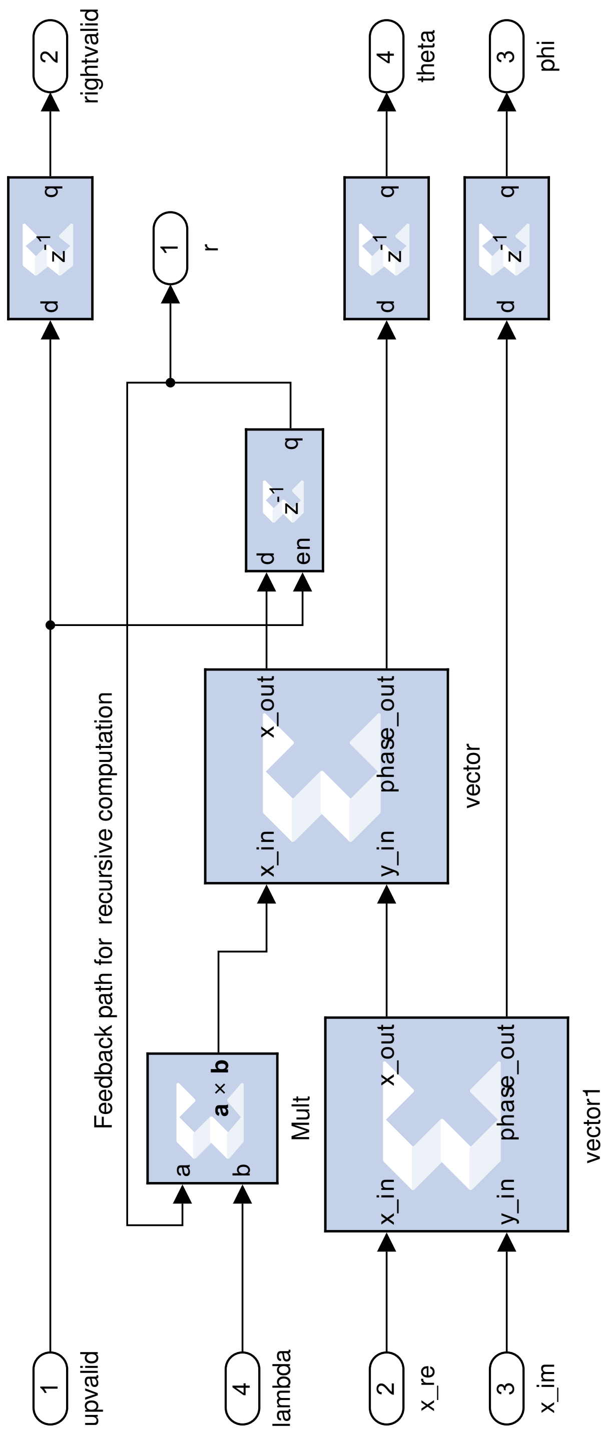
With these parameters, the implementation is straightforward and based on **figure D.1** and uses the same weight flushing method as in **figure 3.14**. In this case the amplitude of the weight flushing pulses cannot be 1 due to overflow. At the same time the pulse should be large enough to reduce quantization effects. Similarly to the regular input to the array, the pulses are scaled with a "pulse gain". This gain was by trial and error chosen to be $1/64$.

In the following **appendix D.2.1**, **D.2.2** and **D.2.3**, the System Generator hardware models are included for a $MN = 2$ system. The top level is identical to **appendix B.2.1**, with the exception of the gain, pulse gain and fewer channels passing to the QRD-RLS systolic array block. It is omitted for this reason.

D.2.1 Systolic array model

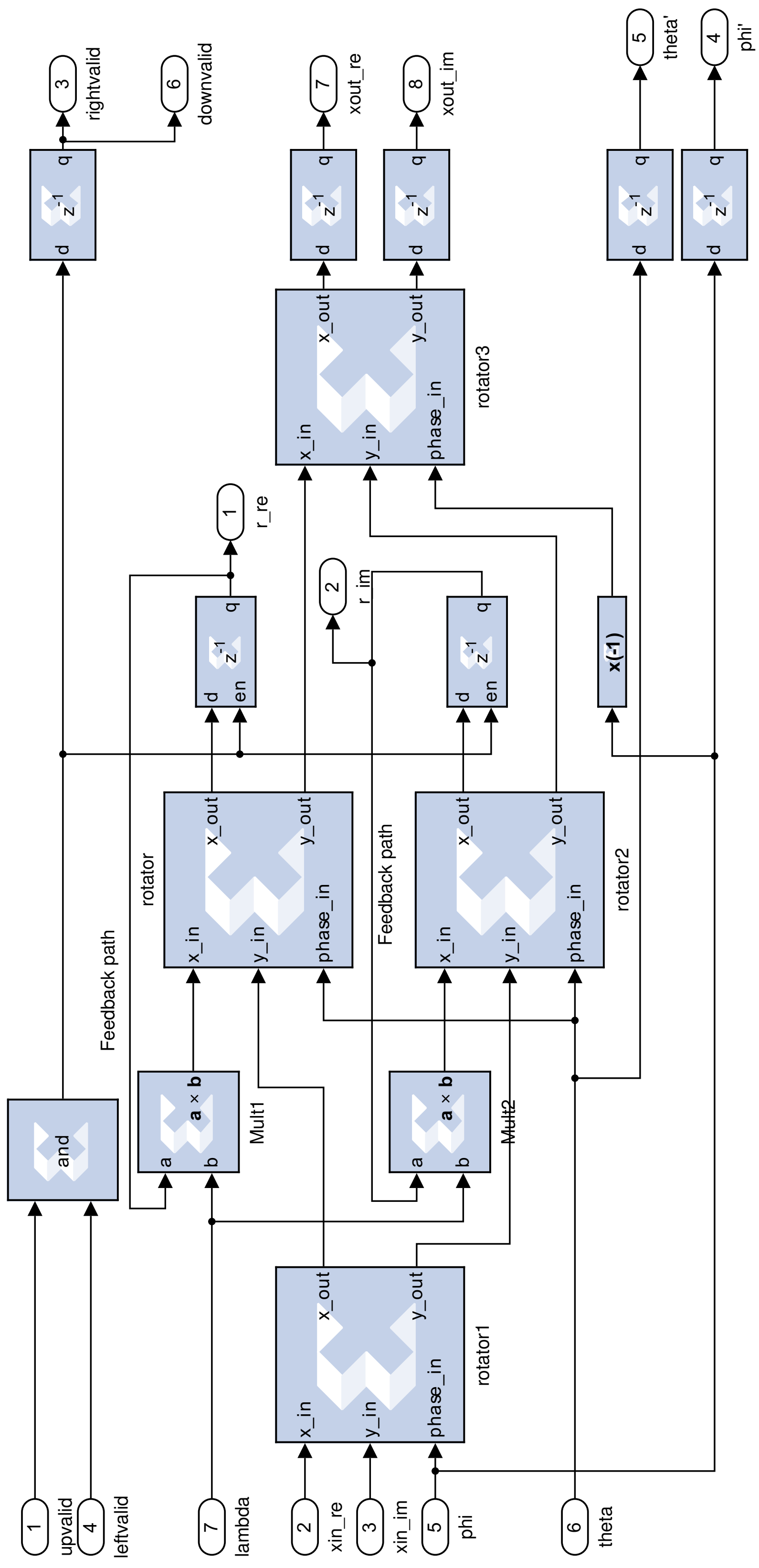


D.2.2 Boundary node model



Note: "Vector" blocks are CORDIC blocks in vectoring mode.

D.2.3 Internal node model



Note: Internal nodes in array column fed by constants use constant 0 instead of feedback path. “Rotator” blocks are CORDIC blocks in rotating mode.

D.2.4 References

[1] K.R. Liu, S.F. Hsieh, K. Yao and C.T. Chiu. Dynamic range, stability, and fault-tolerant capability of finite-precision RLS systolic array based on Givens rotations. *IEEE transactions on circuits and systems*, 38(6):625–636, 1991.

[2] C.M. Rader. VLSI systolic arrays for adaptive nulling. *IEEE Signal Processing Magazine*, 13(4):29–49, 1996.

[3] Q. Gao and R. Stewart. Improved double angle complex rotation QRD-RLS. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays - FPGA '11*, pages 79, New York, USA, 2011. ACM Press.

[4] Xilinx Inc.. DS249 LogiCORE IP CORDIC v4.0. Xilinx Inc., 2011 http://www.xilinx.com/support/documentation/ip_documentation/cordic_ds249.pdf.