ABSTRACT

This thesis report aims at introducing the background of QR decomposition and its application. QR decomposition using Givens rotations is an efficient method to prevent directly matrix inverse in solving least square minimization problem, which is a typical approach for weight calculation in adaptive beamforming. Furthermore, this thesis introduces Givens rotations algorithm and two general VLSI (very large scale integrated circuit) architectures namely triangular systolic array and linear systolic array for numerically QR decomposition. To fulfill the goal, a 4 input channels triangular systolic array with 16 bits fixed-point format and a 5 input channels linear systolic array are implemented on FPGA (Field programmable gate array). The final result shows that the estimated clock frequencies of 65 MHz and 135 MHz on post-place and route static timing report could be achieved using Xilinx Virtex 6 xc6vlx240t chip. Meanwhile, this report proposes a new method to test the dynamic range of QR-D. The dynamic range of the both architectures can be achieved around 110 dB.
ACKNOWLEDGEMENTS

I first would like to acknowledge my graduate advisor Dr. Daniel W. Bliss. His guidance and assistance on my thesis and research was invaluable. His wisdom, kindness and patient impress me. Over the two years, he has shared his ideas, time, and resource graciously with me. For this, I am most appreciative.

I would like to acknowledge Dr. Chaitali Chakrabarti for providing me the FPGA development board and give me advising on my QR-D processor design. I also like to acknowledge Dr. Lei Ying to provide me funding source on the research on USRP. For their contribution, I am most grateful.

Additionally, I would like to acknowledge my friends, the members of Dr. Bliss’s laboratory, for helping on practicing my thesis defense and courses learning over these two years. I also would like to acknowledge Mr. Min Yang, the student of Dr. Chaitali Chakrabarti, and Mr. Zhaorui Wang, my laboratory colleague, for their generous help on details in my thesis.

Finally, I would like to acknowledge my parents for supporting me studying in USA and encouraging me through the hard time. I would like to follow the family traditions that make contributions on research and serve my motherland.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>vii</td>
</tr>
</tbody>
</table>

## CHAPTER

1. INTRODUCTION

1.1 Adaptive Antenna Array ................................................................. 1

1.2 Least Square Minimization Problem ................................................... 2

1.3 Alternative Solution for LS problem: QR Decomposition ....................... 4

2. QR DECOMPOSITION

2.1 QR Algorithms Introduction ............................................................... 6

2.2 Givens Rotations .................................................................................... 7

3. QR DECOMPOSITION PROCESSOR ARRAY

3.1 Introduction ............................................................................................ 11

3.2 Related Work for the VLSI implementation of QR decomposition ............. 11

3.3 Triangular Systolic Array ....................................................................... 13

3.3.1 Boundary Node ..................................................................................... 18

3.3.2 Internal Node ....................................................................................... 20

3.4 Linear Systolic Array ............................................................................. 22
<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.1 Introduction</td>
<td>22</td>
</tr>
<tr>
<td>3.4.2 Derivation of Linear Architecture</td>
<td>24</td>
</tr>
<tr>
<td>3.4.3 Scheduling for the Linear Array</td>
<td>28</td>
</tr>
<tr>
<td>3.4.4 Retiming the Linear Array</td>
<td>32</td>
</tr>
<tr>
<td>4. QR DECOMPOSITION IMPLEMENTATION</td>
<td>37</td>
</tr>
<tr>
<td>4.1 Introduction of FPGA Design</td>
<td>37</td>
</tr>
<tr>
<td>4.1.1 FPGA Introduction</td>
<td>37</td>
</tr>
<tr>
<td>4.1.2 FPGA Design Process</td>
<td>38</td>
</tr>
<tr>
<td>4.1.3 Introduction of Modelsim and ChipScope</td>
<td>39</td>
</tr>
<tr>
<td>4.2 Implementation by the Triangular Array</td>
<td>40</td>
</tr>
<tr>
<td>4.2.1 Boundary Node</td>
<td>41</td>
</tr>
<tr>
<td>4.2.2 Square Root Inverse</td>
<td>43</td>
</tr>
<tr>
<td>4.2.3 Internal Node</td>
<td>48</td>
</tr>
<tr>
<td>4.2.4 Triangular Array Retiming</td>
<td>51</td>
</tr>
<tr>
<td>4.3 Implementation on the Linear Array</td>
<td>53</td>
</tr>
<tr>
<td>4.3.1 Boundary Node</td>
<td>54</td>
</tr>
<tr>
<td>4.3.2 Internal Node</td>
<td>56</td>
</tr>
<tr>
<td>4.3.3 Linear Array Scheduling</td>
<td>57</td>
</tr>
<tr>
<td>5. RESULT AND PERFORMANCE ANALYSIS</td>
<td>61</td>
</tr>
<tr>
<td>CHAPTER</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.1 Result Analysis on Triangular Array</td>
<td>63</td>
</tr>
<tr>
<td>5.2 Result Analysis on Linear Array</td>
<td>69</td>
</tr>
<tr>
<td>6. CONCLUSION AND FUTURE WORK</td>
<td>77</td>
</tr>
<tr>
<td>6.1 Conclusion</td>
<td>77</td>
</tr>
<tr>
<td>6.2 Future Work</td>
<td>77</td>
</tr>
<tr>
<td>6.2.1 Back-substitution</td>
<td>78</td>
</tr>
<tr>
<td>6.2.2 Weights Computation</td>
<td>82</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>84</td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 3.1 Triangular Array Computing Node Utilization</td>
<td>23</td>
</tr>
<tr>
<td>Table 3.2 Triangular Array Computing Node Utilization</td>
<td>23</td>
</tr>
<tr>
<td>Table 4.1 Schedule Of Linear Systolic Array Over 120 Clock Cycles</td>
<td>58</td>
</tr>
<tr>
<td>Table 5.1 A Simulation Result Of QR-D on A 20 By 4 Matrix</td>
<td>65</td>
</tr>
<tr>
<td>Table 5.2 FPGA Result Of QR-D On A 20 By 4 Matrix</td>
<td>65</td>
</tr>
<tr>
<td>Table 5.3 MATLAB Program Result Of QR-D On A 20 By 4 Matrix</td>
<td>66</td>
</tr>
<tr>
<td>Table 5.4 QR-D Result of 25 by 5 Matrix from Modelsim</td>
<td>74</td>
</tr>
<tr>
<td>Table 5.5 QR-D Result of 25 by 5 Matrix from MATLAB</td>
<td>74</td>
</tr>
<tr>
<td>Table 5.6 QR-D Result of 25 by 5 Matrix from FPGA</td>
<td>75</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>fig. 1.1 An Adaptive Antenna Array</td>
<td>2</td>
</tr>
<tr>
<td>Fig. 3.1 Triangular Systolic Array For QR Algorithm</td>
<td>14</td>
</tr>
<tr>
<td>Fig. 3.2 Boundary Node of Triangular Systolic Array</td>
<td>19</td>
</tr>
<tr>
<td>Fig. 3.3 Internal Node of Triangular Systolic Array</td>
<td>21</td>
</tr>
<tr>
<td>Fig. 3.4 Triangular Systolic Array For QR Algorithm</td>
<td>22</td>
</tr>
<tr>
<td>Fig. 3.5 Triangular Systolic Array For QR Algorithm (M=2)</td>
<td>24</td>
</tr>
<tr>
<td>Fig. 3.6 Array After Part B Mirroring Rotation (M=2)</td>
<td>25</td>
</tr>
<tr>
<td>Fig. 3.7 Interleaving The Rectangular Array</td>
<td>26</td>
</tr>
<tr>
<td>Fig. 3.8 Interleaved Rectangular Arrays</td>
<td>27</td>
</tr>
<tr>
<td>Fig. 3.9 QR Decomposition Linear Systolic Array Architecture For 5 Channels</td>
<td>28</td>
</tr>
<tr>
<td>Fig. 3.10 Scheduling The Array</td>
<td>29</td>
</tr>
<tr>
<td>Fig. 3.11 Interleaved QR Iteration (M=2)</td>
<td>31</td>
</tr>
<tr>
<td>Fig. 3.12 A Example Of Retimed QR Nodes When $L_n = 6$ And $T_{qr} = 5$</td>
<td>33</td>
</tr>
<tr>
<td>Fig. 3.13 Effect Latency on The Schedule (M=2)</td>
<td>34</td>
</tr>
<tr>
<td>Fig. 3.14 Retimed Scheduling for The Linear Array, $L_n=6$, $T_{qr}=5$</td>
<td>36</td>
</tr>
<tr>
<td>Fig. 4.1 Fpga Developing Flow Graph</td>
<td>39</td>
</tr>
<tr>
<td>Fig. 4.2 Pipelined Boundary Node</td>
<td>42</td>
</tr>
<tr>
<td>Fig. 4.3 Polynomial Approximation To 1/Sqrt() [5]</td>
<td>45</td>
</tr>
</tbody>
</table>
Fig. 4.4 Square Root Inverse Inside Boundary Node ........................................ 45
Fig. 4.5 RTL Schematic of The Boundary Node ............................................. 47
Fig. 4.6 Internal Node .................................................................................. 49
Fig. 4.7 RTL Schematic Of The Internal Node .............................................. 50
Fig. 4.8 A 4 by 4 QR-D Triangular Systolic Array ....................................... 51
Fig. 4.9 RTL Schematic Of Triangular Array ................................................. 52
Fig. 4.10 RTL Schematic Of A Row Of Triangular Array ............................. 53
Fig. 4.11 Boundary Node Of Linear Array ................................................... 55
Fig. 4.12 Internal Node Of Linear Array ...................................................... 56
Fig. 4.13 Linear QR Decomposition Array Architecture for 5 Input Channels .... 59
Fig. 4.14 Linear QR Decomposition Array RTL Schematic ......................... 60
Fig. 5.1 FPGA Testbench Layout .................................................................. 62
Fig. 5.2 Input Matrix Timing Graph ............................................................ 63
Fig. 5.3 Output Matrix Timing Graph ............................................................ 64
Fig. 5.4 Chipscope Captured Data from Triangular Systolic Array ............... 65
Fig. 5.5 Upper Triangular Matrix from MATLAB Computation .................... 66
Fig. 5.6 Eigen One and Eigen Two Difference vs Input Amplitude For Triangular Systolic Array ........................................................................ 68
Fig. 5.7 Processing Timing Diagram for Sixteen Matrices of 20 by 5 .............. 69
<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fig. 5.8 Timing Diagram of A Complete QR Decomposition</td>
<td>71</td>
</tr>
<tr>
<td>Fig. 5.9 Result Matrix Output Procedure (A)</td>
<td>72</td>
</tr>
<tr>
<td>Fig. 5.10 Result Matrix Output Procedure (B)</td>
<td>73</td>
</tr>
<tr>
<td>Fig. 5.11 Result Matrix from Chipscope</td>
<td>74</td>
</tr>
<tr>
<td>Fig. 5.12 Result Matrix from MATLAB</td>
<td>74</td>
</tr>
<tr>
<td>Fig. 5.13 Eigen One and Eigen Two Difference vs Input Amplitude for Linear Systolic Array</td>
<td>75</td>
</tr>
<tr>
<td>Fig. 6.1 Adaptive Beamforming System [13]</td>
<td>78</td>
</tr>
<tr>
<td>Fig. 6.2 “Frozen” Triangular Array [1]</td>
<td>79</td>
</tr>
<tr>
<td>Fig. 6.3 A QR-D Of 4 Channels with Weight Flushing (Each Node Have One Clock Cycle Delay) [1]</td>
<td>81</td>
</tr>
<tr>
<td>Fig. 6.4 Weights Computing Process</td>
<td>83</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

1.1 Adaptive Antenna Array

The goal of an adaptive antenna array is to select a set of amplitude and phase weights with which to combine the outputs from the antennas. This array is used to produce a far-field pattern that, in some sense, optimizes the reception of a desired signal [1]. The significant improvement in system anti-jam or interference performance achieved by this form of array processing becomes an essential requirement for radar and communication systems. The traditional technique for deriving the adaptive weight vector uses a closed loop gradient descent algorithm, LMS (least mean square) algorithm, where the weight updates are derived from estimation of the correlation between the signal in each channel and the summed output from antenna array. However, this process is one of poor convergence for a broad dynamic range signal environment [1].

An alternative approach is “direct solution” adaptive process which is described as a least-square minimization problem [7]. It describes accurately the external environment and provides an antenna pattern capable of suppress a wide dynamic range of jamming signal [1]. But the problem is that the direct solution method requires large amount of computation comparing with gradient descent algorithms [1] [2]. However, the use of direct solution method is now quite achievable for high bandwidth application by VLSI technique and development of dedicated parallel processing architectures such as systolic array [3].
1.2 Least Square Minimization Problem

The general block of an adaptive antenna system by direct solution approach is illustrated in Fig. 1.1. The beamforming weights are chosen by a beam-pattern controller that continuously updates them in response to antenna array output.

By setting the constraint \( c^T w = \mu \), in which \( c \) is the linear beam constraint or called steer vector, \( w \) is the beamforming weight, \( T \) is matrix transpose and \( \mu \) is a constant gain, this system can deeply null the interference signal other than the direction selected by the
steer vector $c$. Therefore, the LS problem requires the output amplitude of a combined signal [1]

$$e(t_i) = x^T(t_i)w .$$  \hfill (1.1)

To be minimized by subject to a linear beam constrain of this [1]

$$c^T w = \mu .$$  \hfill (1.2)

This is to find the $p$-element vector $w$ which satisfies (1.2) and minimizes the amplitude of a certain time length $n$ of the residual vector [1]

$$e(n) = X(n) w .$$  \hfill (1.3)

Where $X(n)$ denotes the $n$ by $p$ matrix of all data sample which enter the combiner up to time $n$. $c$ and $w$ is a vector of $p$ by 1. The analytic equation for the least squares weight vector can be obtained by equating to zero the complex gradient (with respect to $w^*$ and $\lambda$) of the quantity [1]

$$\Phi = \|e(n)\|^2 + \lambda \left( \mu - c^T w \right) ,$$  \hfill (1.4)

where $\lambda$ is a Lagrange undetermined multiplier. This leads to the well-known equation [1]

$$w(n) = \frac{\mu M^{-1}(n)c^*}{c^T M^{-1}(n)c^*}$$

$$M(n) = X^H(n)X(n) .$$  \hfill (1.5)

$M(n)$ is the estimated $p$ by $p$ covariance matrix of the all samples entering the combiner up to time $n$. The weight vector $w(n)$ can be obtained by directly solving (1.5). However, directly solving the matrix inverse can be very poor and hence numerical unstable. It occurs if the matrix has a very small determinant, in which case the true solution can be
subject to large perturbations and satisfy the equation quite accurately. This will lead to
very complicated circuit architecture for numerical computation. This is definitely not
suitable for VLSI implementation.

1.3 Alternative Solution for LS Problem: QR Decomposition

An alternative approach to the least-squares estimation problem which is the most
suitable for the numerical sense is the orthogonal triangularization [1]. This method is
known as QR decomposition [4].

An $n$ by $p$ data matrix $X(n)$ can be decomposed as [1]

$$ X(n) = Q(n)R(n) $$

where $R(n)$ is a $n$ by $p$ upper triangular matrix and $Q(n)$ is a unitary matrix of $n$ by $n$. $p$ is
the number of antenna elements and $n$ is the number of samples.

Applying QR decomposition to (2.5), we can get [1]

$$ w(n) = \mu \left( (QR)^H(QR)^{-1} c^* \right) $$

where

$$ c^* = \left( c^T M^{-1}(n) c \right)^{-1} c^T M^{-1}(n) c $$

The denominator is some constant number. Thus, we can ignore the denominator and $\mu$
since they will not change the phase of the weights.

$$ w = (R^H R)^{-1} c^* $$

Since the matrix $R(n)$ is upper triangular, (1.8) can be easily solved. The weight vector
$w(n)$ may be derived simply by double back-substitution [5].

Of course, it does not matter to use which of the algorithm for matrix triangularization
when the LS problem is not constrained by speed and throughput, for example simply using computer to do QR decomposition. However, when the application scenario is an adaptive antennas array, for example in MIMO (multiple input multiple output) communication system or phased array radar, there is no choice but the only method to use a proper VLSI (very large scale circuit) system to fulfill the application criteria.
2. QR DECOMPOSITION

2.1 QR Algorithms Introduction

The QR decomposition of a m-by-n matrix \( A \) is given by [4]

\[
A = QR
\]

(2.1)

where \( Q \in \mathbb{C}^{m \times m} \) is orthogonal matrix and \( R \in \mathbb{C}^{m \times n} \) is upper triangular matrix. If \( m > n \), \( R \) consists of an \( n \) by \( n \) right upper triangular matrix and a \((m-n)\) by \( n \) zeroes matrix below the triangular matrix.

For solving the QR decomposition, there are several different methods available to use including Gram-Schmidt algorithm [4], Householder transformation algorithm [4] and Givens rotations algorithm [4] [6]. Gram-Schmidt algorithm utilizes the Gram-Schmidt orthogonalization to compute the upper triangular matrix. Householder transformation algorithm is to eliminate certain lower elements on each row by multiplier each Householder matrices. Givens rotations is to get compute a rotation factor from the adjacent elements to eliminate the lower element in each rotation matrix. This Givens rotations triangularization processing is recursively updated as each new row of data entering the array. The recursive updating only involves two adjacent rows from the input data matrix which is particularly suitable for systolic array structure of VLSI implementation.
2.2 Givens Rotations

A complex Givens rotations [6] is presented by the following elementary transformation of this form

\[
\begin{pmatrix}
    c & s^* \\
    -s & c
\end{pmatrix}
\begin{pmatrix}
    r_{n-1} \\
x_n
\end{pmatrix}
= \begin{pmatrix}
y \\
0
\end{pmatrix},
\]  

(2.2)

where \( n \) presents the different time. \( r_{n-1} \) arrives earlier than \( x_n \) one unit time. We want to eliminate the lower element \( x_n \). Thus the rotation coefficients \( c \) and \( s \) need to satisfy the following equations

\[
\begin{align*}
    r_n &= \sqrt{|x_n|^2 + |r_{n-1}|^2}, \\
    c &= \frac{r_{n-1}}{r_n} \quad &s = \frac{x_n}{r_n}, \\
    s &= \frac{cx_n}{r_{n-1}} \quad &s^* = \frac{cx_n^*}{r_{n-1}}.
\end{align*}
\]  

(2.3)

Bring (2.5) and (2.6) back to (2.7), we can find

\[
\begin{align*}
    -sr_{n-1} + cx_n &= -r_{n-1} \frac{cx_n}{r_{n-1}} + cx_n = cx_n \times (-1 + 1) = 0, \\
    c r_{n-1}^2 + s^* x_n &= \frac{r_{n-1}^2}{r_n} + \frac{x_n^* x_n}{r_n} = r_n = y.
\end{align*}
\]  

(2.8)

Here \( c \) and \( s \) stands for cosine and sine parameters. \( |c|^2 + |s|^2 = 1 \).

Therefore a process of such elimination operation can be used to triangularize the matrix in a recursive manner. Assuming a 4 by 4 matrix, we multiply a rotation matrix where the elementary matrix moves along with column direction as equation (2.11). First, we calculate the value of \( c_1 \) and \( s_1 \) from \( x_{31} \) and \( x_{41} \) by equation (2.9) and (2.10) with \( r_{n-1} \)
equal to \( x_{41} \) and \( x_n \) equal to \( x_{31} \). Then we form the rotation matrix and do multiplication.

The second step is almost the same; the only difference is the following: \( r_{n-1} \) to be \( x_{31} \) and \( x_n \) to be \( x_{21} \). After doing all the processing as below, finally we can get the correct upper triangular matrix.

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & c_1 & s_1 \\
0 & 0 & -s_1 & c_1
\end{bmatrix}
\begin{pmatrix}
x_{11} & x_{12} & x_{13} & x_{14} \\
x_{21} & x_{22} & x_{23} & x_{24} \\
x_{31} & x_{32} & x_{33} & x_{34} \\
x_{41} & x_{42} & x_{43} & x_{44}
\end{pmatrix}
= \begin{pmatrix}
x_{11} & x_{12} & x_{13} & x_{14} \\
x_{21} & x_{22} & x_{23} & x_{24} \\
x_{31} & x_{32} & x_{33} & x_{34} \\
0 & x_{42} & x_{43} & x_{44}
\end{pmatrix}
\]

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{pmatrix}
x_{11} & x_{12} & x_{13} & x_{14} \\
x_{21} & x_{22} & x_{23} & x_{24} \\
x_{31} & x_{32} & x_{33} & x_{34} \\
0 & x_{42} & x_{43} & x_{44}
\end{pmatrix}
= \begin{pmatrix}
x_{11} & x_{12} & x_{13} & x_{14} \\
x_{21} & x_{22} & x_{23} & x_{24} \\
x_{31} & x_{32} & x_{33} & x_{34} \\
0 & x_{42} & x_{43} & x_{44}
\end{pmatrix}
\]

\[
\begin{bmatrix}
c_3 & s_3^* & 0 & 0 \\
-s_3 & c_3 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{pmatrix}
x_{11} & x_{12} & x_{13} & x_{14} \\
x_{21} & x_{22} & x_{23} & x_{24} \\
x_{31} & x_{32} & x_{33} & x_{34} \\
0 & x_{42} & x_{43} & x_{44}
\end{pmatrix}
= \begin{pmatrix}
x_{11} & x_{12} & x_{13} & x_{14} \\
x_{21} & x_{22} & x_{23} & x_{24} \\
x_{31} & x_{32} & x_{33} & x_{34} \\
0 & x_{42} & x_{43} & x_{44}
\end{pmatrix}
\]

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & c_4 & s_4 \\
0 & 0 & -s_4 & c_4
\end{bmatrix}
\begin{pmatrix}
x_{11} & x_{12} & x_{13} & x_{14} \\
x_{21} & x_{22} & x_{23} & x_{24} \\
x_{31} & x_{32} & x_{33} & x_{34} \\
0 & x_{42} & x_{43} & x_{44}
\end{pmatrix}
= \begin{pmatrix}
x_{11} & x_{12} & x_{13} & x_{14} \\
x_{21} & x_{22} & x_{23} & x_{24} \\
x_{31} & x_{32} & x_{33} & x_{34} \\
0 & x_{42} & x_{43} & x_{44}
\end{pmatrix}
\]

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & c_5 & s_5^* & 0 \\
0 & -s_5 & c_5 & 0 \\
0 & 0 & 1 & 0
\end{bmatrix}
\begin{pmatrix}
x_{11} & x_{12} & x_{13} & x_{14} \\
x_{21} & x_{22} & x_{23} & x_{24} \\
x_{31} & x_{32} & x_{33} & x_{34} \\
0 & x_{43} & x_{44}
\end{pmatrix}
= \begin{pmatrix}
x_{11} & x_{12} & x_{13} & x_{14} \\
x_{21} & x_{22} & x_{23} & x_{24} \\
x_{31} & x_{32} & x_{33} & x_{34} \\
0 & x_{43} & x_{44}
\end{pmatrix}
\]

\[
\begin{bmatrix}
c_6 & s_6^* & 0 & 0 \\
-s_6 & c_6 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{pmatrix}
x_{11} & x_{12} & x_{13} & x_{14} \\
x_{21} & x_{22} & x_{23} & x_{24} \\
x_{31} & x_{32} & x_{33} & x_{34} \\
0 & x_{43} & x_{44}
\end{pmatrix}
= \begin{pmatrix}
x_{11} & x_{12} & x_{13} & x_{14} \\
x_{21} & x_{22} & x_{23} & x_{24} \\
x_{31} & x_{32} & x_{33} & x_{34} \\
0 & 0 & 0 & x_{44}
\end{pmatrix}
\]

\[
(2.11)
\]

8
From the above process, we can see the rotation values come from the adjacent two elements in the same column, for example $c_1$ and $s_1$ are computed from $x_{31}$ and $x_{41}$. Then $c_1$ and $s_1$ will involve the multiplication on matrix $\begin{pmatrix} x_{32} & x_{33} & x_{34} \\ x_{42} & x_{43} & x_{44} \end{pmatrix}$. Thus, we can conclude there are two kinds of computation. First one is to get $c$ and $s$ values and second is to do the matrix multiplication. Meanwhile, if we regard the computation involving two adjacent elements on the same column as one computation node, for example, on matrix $\begin{pmatrix} x_{31} & x_{32} & x_{33} & x_{34} \\ x_{41} & x_{42} & x_{43} & x_{44} \end{pmatrix}$, we can get four computation nodes (N1, N2, N3, N4). N1 computes $c$ and $s$ values from $x_{31}$ and $x_{41}$ by equation (2.3) (2.4) then pass them to the N2. N2 will do some multiplication as equation (2.8) and pass same $c$ and $s$ to its next adjacent node. The data passing among these nodes will looks like a heart beating. Thus, for VLSI implementation, systolic array architecture is the most efficient method for this Givens rotation.

For a more general case $m>n$

$$\begin{pmatrix} 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 & 0 \\ 0 & 0 & \cdots & 0 & c_i & s_i \end{pmatrix} \times \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1(n-2)} & x_{1(n-1)} & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2(n-2)} & x_{2(n-1)} & x_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ x_{(m-2)1} & x_{(m-2)2} & \cdots & x_{(m-2)(n-2)} & x_{(m-2)(n-1)} & x_{(m-2)n} \\ x_{(m-1)1} & x_{(m-1)2} & \cdots & x_{(m-1)(n-2)} & x_{(m-1)(n-1)} & x_{(m-1)n} \\ x_{m1} & x_{m2} & \cdots & x_{m(n-2)} & x_{m(n-1)} & x_{mn} \end{pmatrix}$$

We form the above rotation matrix, do the matrix multiplication then shift the small rotation matrix along the diagonal by one position, and continue to do the next rotation
computation as the followings

\[
\begin{pmatrix}
1 & 0 & \cdots & 0 & 0 \\
0 & 1 & \cdots & 0 & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \cdots & c_2 & s_2^* \\
0 & 0 & \cdots & -s_2 & c_2 \\
0 & 0 & \cdots & 0 & 0 & 1
\end{pmatrix}
\times
\begin{pmatrix}
x_{11} & x_{12} & \cdots & x_{1(n-2)} & x_{1(n-1)} & x_{1n} \\
x_{21} & x_{22} & \cdots & x_{2(n-2)} & x_{2(n-1)} & x_{2n} \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
x_{(m-2)1} & x_{(m-2)2} & \cdots & x_{(m-2)(n-2)} & x_{(m-2)(n-1)} & x_{(m-2)n} \\
x_{(m-1)1} & x_{(m-1)2} & \cdots & x_{(m-1)(n-2)} & x_{(m-1)(n-1)} & x_{(m-1)n} \\
0 & x_{m2} & \cdots & x_{m(n-2)} & x_{m(n-1)} & x_{mn}
\end{pmatrix},
\]

Until the small rotation matrix reaches the left top as below

\[
\begin{pmatrix}
c_m & s_m^* & \cdots & 0 & 0 \\
-s_m & c_m & \cdots & 0 & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \cdots & 1 & 0 \\
0 & 0 & \cdots & 0 & 1 \\
0 & 0 & \cdots & 0 & 0 & 1
\end{pmatrix}\times
\begin{pmatrix}
x_{11} & x_{12} & \cdots & x_{1(n-2)} & x_{1(n-1)} & x_{1n} \\
x_{21} & x_{22} & \cdots & x_{2(n-2)} & x_{2(n-1)} & x_{2n} \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
x_{(m-2)1} & x_{(m-2)2} & \cdots & x_{(m-2)(n-2)} & x_{(m-2)(n-1)} & x_{(m-2)n} \\
x_{(m-1)1} & x_{(m-1)2} & \cdots & x_{(m-1)(n-2)} & x_{(m-1)(n-1)} & x_{(m-1)n} \\
0 & x_{m2} & \cdots & x_{m(n-2)} & x_{m(n-1)} & x_{mn}
\end{pmatrix},
\]

Put the small rotation matrix back to the right bottom as below, and start the new round of rotations to eliminate the elements on the second column.

\[
\begin{pmatrix}
1 & 0 & \cdots & 0 & 0 \\
0 & 1 & \cdots & 0 & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \cdots & 1 & 0 \\
0 & 0 & \cdots & 0 & c_{m+1} \\
0 & 0 & \cdots & 0 & -s_{m+1}
\end{pmatrix}\times
\begin{pmatrix}
\dot{x}_{11} & \dot{x}_{12} & \cdots & \dot{x}_{1(n-2)} & \dot{x}_{1(n-1)} & \dot{x}_{1n} \\
0 & \dot{x}_{22} & \cdots & \dot{x}_{2(n-2)} & \dot{x}_{2(n-1)} & \dot{x}_{2n} \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & x_{(m-2)2} & \cdots & x_{(m-2)(n-2)} & x_{(m-2)(n-1)} & x_{(m-2)n} \\
0 & x_{(m-1)2} & \cdots & x_{(m-1)(n-2)} & x_{(m-1)(n-1)} & x_{(m-1)n} \\
0 & x_{m2} & \cdots & x_{m(n-2)} & x_{m(n-1)} & x_{mn}
\end{pmatrix},
\]

Of course, the \( c \) and \( s \) are different on each rotation.
3. QR DECOMPOSITION PROCESSOR ARRAY

3.1 Introduction

In order to fulfill the high speed and high throughput requirement for the scenario of antenna array beamforming, a highly parallel systolic array can be mapped from the Givens rotations algorithm. Input the matrix, need triangularization, to the processing array. Then it will output the each element of the triangular matrix or store each element in corresponding nodes and continue for beamforming weight computation. VLSI is the best choice to fulfill the goal. However, it is impossible for all these QRD algorithms to be implemented into a VLSI system. The best choice is to use Givens rotations [1][8][9]. Using Givens rotations, QRD can be represented as a highly parallel array consists of computation nodes. This leads to systolic array architecture.

3.2 Related Work for the VLSI implementation of QR decomposition

The earliest work of VLSI architecture of QRD is proposed by H. T Kung and W. M Gentleman in 1981[10]. Two years later, in 1983 J. G. McWhirter [9] also proposed the similar method of matrix triangulation for solving least squares minimization using systolic array. But McWhirter’s work has the antenna array as the background and also proposed the proper method for compute beamforming weight. The following published works are all based on their fundamental architecture, which mainly focus on improving
computation speed and throughput. These works fall into three categories [11]. One classic of implementation is square root free using the Squared Givens rotation algorithm [12 W. M Gentleman 1973]. Another type of implementation is based on Logarithmic Number system arithmetic. The last category of implementation uses CORDIC algorithm [22 Volder 1959].

In 1999, a novel linear QR architecture was developed by R. Walke[14]. It is not the first work to propose a linear array for least square problem. The earlier works come from different authors, namely Yang and Bohme [24], Chen and Yao [25], Rader [26]. However, this work is quite significant different from other kinds of linear array. It is directly mapped from the conventional triangular array ([9] McWhirter array) based on the same recursive algorithm. Due to its easy to understand and build, more recently public works [5][19][23] and even commercial QR-D IP cores use this linear systolic array. The linear systolic array helps to save huge computation nodes and improve the utilization of each computation node to 100 percent compared with the conventional QR architecture [9][10].

On FPGA side, [R. Walke et al.] first use QRD for weight computation for adaptive beamforming application. A Squared Givens Rotation (W. M Gentleman 1973) algorithm is used to avoid the square root operation. The follow up work [R. Walke et al. 2000] uses the SGR algorithm and implements a linear array on Xilinx Virtex-E FPGA. A maximum of 9 processors can fit in, achieving a clock rare of 150MHz and throughput of
20GFLOPS with floating point operation [13]. R. L. Walke et al also first implements a linear array architecture [2][14] using CORDIC algorithm for rotation computing [16]. From R. Walke’s work, a 26-bits fixed-point data format is used to guarantee sufficient interference suppression. There are also many commercial QR-D IP (Intelligence Patent) cores using CORDIC algorithm. Altera published a QRD-RLS design based on CORDIC core [18].

3.3 Triangular Systolic Array

This kind of systolic array for QR decomposition is proposed by H. T Kung and W. M Gentleman [10]. J. G. McWhirter first introduces this method to the scenario of antenna array [9].

This systolic array consists of two types of computational nodes: boundary node on diagonal of the triangular matrix and internal node off the diagonal. The boundary nodes are used to calculate the Givens rotation that is applied over a particular row in the input matrix. An orthogonal rotation matrix as in equation (2.2) that can eliminate one lower triangular element of the decomposed matrix is computed in the boundary node, while the rotation parameters $c$ and $s$ are output to the internal nodes. The generated rotation parameters are sequentially broadcasted to the internal nodes in the same row as the boundary node from left to right by a certain clock rate. The internal nodes apply this Givens rotation parameters received from the boundary node on the same row as well as
its own input values to compute new values as outputs. Using the rotation values \( c \) and \( s \) from the boundary node, the internal nodes update the remaining elements at the two adjacent rows of the matrix involved in one rotation as specified in equation (2.2) and (2.3).

\[
\begin{align*}
    r_{ij}(n) &= c_{ij}(n-1) + s_{ij}(n-1) \\
    s_{ij}(n) &= -s_{ij}(n-1) + c_{ij}(n-1) \\
    c(n) &= r_i(n-1) \left/ r_e(n-1) \right. \\
    s(n) &= x_i(n) \left/ r_e(n-1) \right. \\
    r_e(n-1) &\text{ initialized by 0}
\end{align*}
\]

\[
\begin{array}{ccccccc}
    t_{m-1} & 0 & 0 & 0 & 0 & a_{m5} \\
    t_{m-2} & 0 & 0 & a_{m4} & a_{m5} & a_{m5} \\
    t_{m-3} & 0 & a_{m3} & a_{m4} & a_{m5} & a_{m5} \\
    t_{m-4} & a_{m2} & a_{m3} & a_{m4} & a_{m5} & a_{m5} \\
    t_{m-5} & a_{m1} & a_{m2} & a_{m3} & a_{m4} & a_{m5} \\
    t_7 & a_{71} & a_{72} & a_{73} & a_{74} & a_{75} \\
    t_6 & a_{61} & a_{62} & a_{63} & a_{64} & a_{65} \\
    t_5 & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \\
    t_4 & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\
    t_3 & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\
    t_2 & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\
    t_1 & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\
    t_0 & a_{0} & a_{0} & a_{0} & a_{0} & a_{0} \\
\end{array}
\]

Fig. 3.1 Triangular systolic array for QR algorithm [5]

First let us assume computation for each node takes 1 clock period. It means that the calculated result values will show up 1 clock period later than the input values. The rows of the input matrix are fed as inputs to the systolic array from the top. As the schedule
graph on Fig. 3.1, after receiving the first input value $a_{11}$ at $t_0$, the boundary node $N_{11}$ starts computing. The boundary node $N_{11}$ will update its stored value $r_{11}$ and output $c$ and $s$ at the time $t_1$. Meanwhile, internal node $N_{12}$ receives $c$, $s$ and $a_{12}$. It will update its stored value $r_{12}$ and pass the result to the lower level node $N_{22}$. It also passes the same $c$ and $s$ to its right adjacent node $N_{13}$ at time $t_2$. At $t_2$, internal node $N_{13}$ receives $c$, $s$ and $a_{13}$. After one clock period computation, it will update its stored value $r_{13}$ and pass the result to the lower level node $N_{23}$ and the same $c$ and $s$ to its right adjacent node $N_{14}$ at the time $t_3$. At $t_3$, internal node $N_{14}$ receives $c$, $s$ and $a_{14}$. After one clock period computation, it will update its stored value $r_{14}$ and pass the result to the lower level node $N_{24}$ and the same $c$ and $s$ to its right adjacent node $N_{15}$ at the time $t_4$. At $t_4$, internal node $N_{15}$ receives $c$, $s$ and $a_{15}$. After one clock period computation, it will only update its stored value $r_{15}$ and pass the result to the lower level node $N_{25}$ at the time $t_5$. Right now, the computation on first row of data matrix and updated row passing is done.

The updated data row passes to the next processing row $N_{22}$ $N_{23}$ $N_{24}$ $N_{25}$ at $t_2$ and finishes at $t_6$. At the same time $t_2$, the first boundary node continues to compute rotation parameters from the next row data $a_{21}$ and stored row data $r_{11}$. Then pass the $c$ and $s$ value to the internal nodes on the same row. The internal node $N_{12}$ to $N_{15}$ will also continue to compute on the row data $a_{22}$ to $a_{25}$) and stored row data ($r_{12}$ to $r_{15}$), update stored row data ($r_{12}$ to $r_{15}$) and output data to the row below them. In the same fashion, the boundary node $N_{33}$ will start processing until the output data arrive from the internal node $N_{23}$ above it. It
then passes the rotation parameters $c$ and $s$ through internal nodes of the same row sequentially. The timing detail can be seen on following Table 3.1, if the input matrix size is 5 by 5.

Table 3.1 Timing Table of QR decomposition systolic array for 5 by 5 complex matrix

<table>
<thead>
<tr>
<th>Time</th>
<th>Boundary nodes {input} $\rightarrow$ {output}</th>
<th>Internal nodes {input} $\rightarrow$ {output}</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_0$</td>
<td>$N_{11}{a_{11}} \rightarrow {c_{11}, s_{11}, \text{update}(r_{11})}$ at $t_1$</td>
<td>$N_{12}{a_{12}, c_{11}, s_{11}} \rightarrow {c_{12}, s_{11}, x^{'2}<em>{22}, \text{update}(r</em>{12})}$ at $t_2$</td>
</tr>
<tr>
<td>$t_1$</td>
<td>$N_{11}{a_{21}} \rightarrow {c_{12}, s_{12}, \text{update}(r_{11})}$ at $t_2$</td>
<td>$N_{12}{a_{22}, c_{12}, s_{12}} \rightarrow {c_{12}, s_{12}, x^{'2}<em>{22}, \text{update}(r</em>{12})}$ at $t_3$</td>
</tr>
<tr>
<td>$t_2$</td>
<td>$N_{11}{a_{31}} \rightarrow {c_{13}, s_{13}, \text{update}(r_{11})}$ at $t_3$</td>
<td>$N_{12}{a_{32}, c_{13}, s_{13}} \rightarrow {c_{13}, s_{13}, x^{'2}<em>{23}, \text{update}(r</em>{12})}$ at $t_4$</td>
</tr>
<tr>
<td></td>
<td>$N_{22}{x^{'2}<em>{22}} \rightarrow {c</em>{22}, s_{22}, \text{update}(r_{22})}$ at $t_3$</td>
<td>$N_{13}{a_{13}, c_{13}, s_{11}} \rightarrow {c_{13}, s_{11}, x^{'2}<em>{23}, \text{update}(r</em>{13})}$ at $t_4$</td>
</tr>
<tr>
<td>$t_3$</td>
<td>$N_{11}{a_{41}} \rightarrow {c_{14}, s_{14}, \text{update}(r_{11})}$ at $t_4$</td>
<td>$N_{12}{a_{32}, c_{14}, s_{14}} \rightarrow {c_{14}, s_{14}, x^{'2}<em>{22}, \text{update}(r</em>{12})}$ at $t_5$</td>
</tr>
<tr>
<td></td>
<td>$N_{22}{x^{'2}<em>{22}} \rightarrow {c</em>{22}, s_{22}, \text{update}(r_{22})}$ at $t_4$</td>
<td>$N_{13}{a_{23}, c_{12}, s_{12}} \rightarrow {c_{12}, s_{12}, x^{'2}<em>{23}, \text{update}(r</em>{13})}$ at $t_5$</td>
</tr>
<tr>
<td></td>
<td>$N_{33}{x^{'3}<em>{33}} \rightarrow {c</em>{31}, s_{31}, \text{update}(r_{33})}$ at $t_3$</td>
<td>$N_{14}{a_{32}, c_{14}, s_{12}} \rightarrow {c_{12}, s_{12}, x^{'2}<em>{24}, \text{update}(r</em>{14})}$ at $t_5$</td>
</tr>
<tr>
<td></td>
<td>$N_{33}{x^{'3}<em>{33}} \rightarrow {c</em>{31}, s_{31}, \text{update}(r_{33})}$ at $t_6$</td>
<td>$N_{15}{a_{14}, c_{11}, s_{11}} \rightarrow {c_{11}, s_{11}, x^{'2}<em>{24}, \text{update}(r</em>{15})}$ at $t_6$</td>
</tr>
<tr>
<td>$t_4$</td>
<td>$N_{11}{a_{31}} \rightarrow {c_{15}, s_{15}, \text{update}(r_{11})}$ at $t_5$</td>
<td>$N_{12}{a_{42}, c_{14}, s_{14}} \rightarrow {c_{14}, s_{14}, x^{'2}<em>{22}, \text{update}(r</em>{12})}$ at $t_6$</td>
</tr>
<tr>
<td></td>
<td>$N_{22}{x^{'2}<em>{22}} \rightarrow {c</em>{23}, s_{23}, \text{update}(r_{22})}$ at $t_5$</td>
<td>$N_{13}{a_{33}, c_{13}, s_{13}} \rightarrow {c_{13}, s_{13}, x^{'2}<em>{23}, \text{update}(r</em>{13})}$ at $t_6$</td>
</tr>
<tr>
<td></td>
<td>$N_{33}{x^{'3}<em>{33}} \rightarrow {c</em>{31}, s_{31}, \text{update}(r_{33})}$ at $t_6$</td>
<td>$N_{14}{a_{34}, c_{13}, s_{13}} \rightarrow {c_{12}, s_{12}, x^{'2}<em>{24}, \text{update}(r</em>{14})}$ at $t_6$</td>
</tr>
<tr>
<td></td>
<td>$N_{33}{x^{'3}<em>{33}} \rightarrow {c</em>{31}, s_{31}, \text{update}(r_{33})}$ at $t_6$</td>
<td>$N_{15}{a_{25}, c_{12}, s_{12}} \rightarrow {x^{'2}<em>{25}, \text{update}(r</em>{15})}$ at $t_6$</td>
</tr>
<tr>
<td>$t_5$</td>
<td>$N_{22}{x^{'2}<em>{24}} \rightarrow {c</em>{22}, s_{24}, \text{update}(r_{22})}$ at $t_6$</td>
<td>$N_{23}{x^{'2}<em>{23}, c</em>{23}, s_{23}} \rightarrow {c_{23}, s_{23}, x^{'2}<em>{23}, \text{update}(r</em>{23})}$ at $t_6$</td>
</tr>
<tr>
<td></td>
<td>$N_{33}{x^{'3}<em>{33}} \rightarrow {c</em>{32}, s_{32}, \text{update}(r_{33})}$ at $t_6$</td>
<td>$N_{24}{x^{'2}<em>{24}, c</em>{22}, s_{22}} \rightarrow {c_{22}, s_{22}, x^{'2}<em>{24}, \text{update}(r</em>{24})}$ at $t_6$</td>
</tr>
<tr>
<td></td>
<td>$N_{33}{x^{'3}<em>{33}} \rightarrow {c</em>{32}, s_{32}, \text{update}(r_{33})}$ at $t_6$</td>
<td>$N_{25}{x^{'2}<em>{25}, c</em>{21}, s_{21}} \rightarrow {x^{'2}<em>{25}, \text{update}(r</em>{25})}$ at $t_6$</td>
</tr>
<tr>
<td></td>
<td>$N_{33}{x^{'3}<em>{33}} \rightarrow {c</em>{32}, s_{32}, \text{update}(r_{33})}$ at $t_6$</td>
<td>$N_{34}{x^{'2}<em>{24}, c</em>{31}, s_{31}} \rightarrow {c_{31}, s_{31}, x^{'2}<em>{24}, \text{update}(r</em>{34})}$ at $t_6$</td>
</tr>
<tr>
<td>$t_6$</td>
<td>$N_{22}{x^{'2}<em>{22}} \rightarrow {c</em>{25}, s_{25}, \text{update}(r_{22})}$ at $t_7$</td>
<td>$N_{13}{a_{35}, c_{15}, s_{15}} \rightarrow {c_{15}, s_{15}, x^{'2}<em>{25}, \text{update}(r</em>{15})}$ at $t_7$</td>
</tr>
<tr>
<td></td>
<td>$N_{33}{x^{'3}<em>{33}} \rightarrow {c</em>{33}, s_{33}, \text{update}(r_{33})}$ at $t_7$</td>
<td>$N_{14}{a_{44}, c_{14}, s_{14}} \rightarrow {c_{12}, s_{12}, x^{'2}<em>{24}, \text{update}(r</em>{14})}$ at $t_7$</td>
</tr>
<tr>
<td></td>
<td>$N_{44}{x^{'4}<em>{44}} \rightarrow {c</em>{41}, s_{41}, \text{update}(r_{44})}$ at $t_7$</td>
<td>$N_{15}{a_{36}, c_{13}, s_{13}} \rightarrow {x^{'2}<em>{25}, \text{update}(r</em>{15})}$ at $t_7$</td>
</tr>
<tr>
<td></td>
<td>$N_{33}{x^{'3}<em>{33}} \rightarrow {c</em>{33}, s_{33}, \text{update}(r_{33})}$ at $t_7$</td>
<td>$N_{23}{x^{'2}<em>{23}, c</em>{24}, s_{24}} \rightarrow {c_{24}, s_{24}, x^{'3}<em>{33}, \text{update}(r</em>{23})}$ at $t_7$</td>
</tr>
<tr>
<td></td>
<td>$N_{44}{x^{'4}<em>{44}} \rightarrow {c</em>{41}, s_{41}, \text{update}(r_{44})}$ at $t_7$</td>
<td>$N_{24}{x^{'2}<em>{24}, c</em>{23}, s_{23}} \rightarrow {c_{23}, s_{23}, x^{'3}<em>{34}, \text{update}(r</em>{24})}$ at $t_7$</td>
</tr>
<tr>
<td></td>
<td>$N_{33}{x^{'3}<em>{33}} \rightarrow {c</em>{33}, s_{33}, \text{update}(r_{33})}$ at $t_7$</td>
<td>$N_{25}{x^{'2}<em>{25}, c</em>{22}, s_{22}} \rightarrow {x^{'2}<em>{25}, \text{update}(r</em>{25})}$ at $t_7$</td>
</tr>
<tr>
<td></td>
<td>$N_{44}{x^{'4}<em>{44}} \rightarrow {c</em>{41}, s_{41}, \text{update}(r_{44})}$ at $t_7$</td>
<td>$N_{34}{x^{'2}<em>{24}, c</em>{32}, s_{32}} \rightarrow {c_{32}, s_{32}, x^{'2}<em>{44}, \text{update}(r</em>{34})}$ at $t_7$</td>
</tr>
<tr>
<td></td>
<td>$N_{33}{x^{'3}<em>{33}} \rightarrow {c</em>{33}, s_{33}, \text{update}(r_{33})}$ at $t_7$</td>
<td>$N_{35}{x^{'3}<em>{35}, c</em>{31}, s_{31}} \rightarrow {x^{'2}<em>{25}, \text{update}(r</em>{34})}$ at $t_7$</td>
</tr>
</tbody>
</table>
Data flow from top to bottom and from left to right in this structure. The calculation of
the decomposition matrix $A$ propagates through the systolic array on a diagonal wave
front. Once all inputs have passed through the systolic array, the values held in each of
the computation nodes are the output values of the decomposed $R$ matrix. This systolic
array clearly exhibits some desired properties such as regularity and local
interconnections which render it comparatively simple to implement. Also, the control
overhead is extremely low since the processing nodes operate synchronously and can be
controlled by a simple global distributed clock.
On the original J. G. McWhirter’s work, there is only one clock cycle for each computing node. Apparently, it will be very slow since all the commutating operations inside each node have to be done in 1 clock cycle. In this case, the 1 clock cycle period should be very long to meet the time consumption of the critical path inside each node. Therefore, the pipelines can be introduced into each computation node. It will shorten the critical path length and increase clock speed significantly. However, the thing needed to focus is that a proper retiming strategy has to be chosen for nodes interconnection.

3.3.1 Boundary Node

Figure 3.2 shows data flow of a boundary node, which require six multipliers, two adders, one square root, one divider and two local memories. This data path is directly implemented from equation (2.3) (2.4) (3.1). The input is the complex number “x”. The outputs are “r”, “c” and “s”. The current input value x and the previous value r (square of r\_old is r\_sq\_old) are the elements on the same column and on the neighbor rows of the input matrix. During one Givens rotation, an orthogonal transformation is applied and the new values are computed. The new value of r will be stored in memory and become the previous value used as one of the inputs of the next rotation. The difficulty of boundary node is to find an efficient method to do the square root inverse. CORDIC algorithm or logarithm can be used for this operation.
\[
\begin{align*}
    r_n(n) &= \sqrt{r_n^2(n-1) + |x_n(n)|^2} \\
    c(n) &= \frac{r_n(n-1)}{r_n(n)} \\
    s(n) &= \frac{x_n(n)}{r_n(n)}
\end{align*}
\] (3.1)

Fig. 3.2 Boundary node of triangular systolic array [5]
3.3.2 Internal Node

Compared with the boundary node, internal node is much simpler. It requires twelve multipliers, five adders, three subtractors and two local memories. It is directly implemented from equation (2.8) (3.2) in Fig. 3.3. The inputs are rotation parameters $c$, $s$ whereas $x$ is the matrix data input. Outputs are $r_{\text{real}}$, $r_{\text{im}}$, $x_{\text{out}\_\text{real}}$ and $x_{\text{out}\_\text{im}}$. Another two inputs $r_{\text{real}\_\text{old}}$ and $r_{\text{im}\_\text{old}}$ are initialed by zero but updated when the new values are input into the internal node. The previous $r_{\text{old}}$ value and current input $x$ are the elements on the same column and on the neighbor rows of the input matrix. The internal node applies the orthogonal transform to the input value and previous value, and calculates two new complex numbers.

\[
\begin{align*}
r_{ij}(n) &= c_{ij}r_{ij}(n-1) + s_{ij}^*x_{ij}(n) \\
x_{i+1,j}(n) &= -s_{ij}r_{ij}(n-1) + c_{ij}x_{ij}(n)
\end{align*}
\] (3.2)
Fig. 3.3 Internal node of triangular systolic array [5]
3.4 Linear Systolic Array

3.4.1 Introduction

From the previous discussion, we can see that the structure of QRD triangular systolic array is very simple. However, the huge size of this array is a problem. For an $m$ by $n$ decomposition matrix $A$ ($m \geq n$), the triangular systolic array needs $(n+1)n/2$ computation nodes. If $n$ is a large number, it will consume a huge circuit resource. Meanwhile, there are some nodes that are not fully utilized, for example in Fig. 3.4, we can find the bottom nodes and top row nodes will not be 100 percent utilized at the beginning and the ending.

\[
\begin{array}{ccccccc}
  t_{28} & 0 & 0 & 0 & 0 & a_{25,5} \\
  t_{27} & 0 & 0 & 0 & a_{25,4} & a_{24,5} \\
  t_{26} & 0 & a_{25,3} & a_{24,4} & a_{23,5} \\
  t_{25} & a_{25,2} & a_{24,3} & a_{23,4} & a_{22,5} \\
  t_{24} & a_{25,1} & a_{24,2} & a_{23,3} & a_{22,4} & a_{21,5} \\
  t_2 & a_{31} & a_{22} & a_{13} & 0 & 0 \\
  t_1 & a_{21} & a_{12} & 0 & 0 & 0 \\
  t_0 & a_{11} & 0 & 0 & 0 & 0 \\
\end{array}
\]

Fig. 3.4 Triangular systolic array for QR algorithm
Table 3.1 Triangular Array Computing Node Utilization

<table>
<thead>
<tr>
<th>Node</th>
<th>Used times during decomposition</th>
<th>Utilization rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_{11} N_{12} N_{13} N_{14} N_{15}</td>
<td>25</td>
<td>86%</td>
</tr>
<tr>
<td>N_{22} N_{23} N_{24} N_{25}</td>
<td>24</td>
<td>83%</td>
</tr>
<tr>
<td>N_{33} N_{34} N_{35}</td>
<td>23</td>
<td>79%</td>
</tr>
<tr>
<td>N_{44} N_{45}</td>
<td>22</td>
<td>76%</td>
</tr>
<tr>
<td>N_{15}</td>
<td>21</td>
<td>72%</td>
</tr>
</tbody>
</table>

On the Table 3.2, each node is not 100 percent utilized. If the input matrix is an m by 5 (m>>5), it will increase the utilization. However, the significant problem that still exists is that the computation resource is wasted.

Table 3.2 Triangular Array Computing Node Utilization

<table>
<thead>
<tr>
<th>Node</th>
<th>Used times during decomposition</th>
<th>Utilization rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_{11} N_{12} N_{13} N_{14} N_{15}</td>
<td>m</td>
<td>m/(m+4)</td>
</tr>
<tr>
<td>N_{22} N_{23} N_{24} N_{25}</td>
<td>m-1</td>
<td>m-1/(m+4)</td>
</tr>
<tr>
<td>N_{33} N_{34} N_{35}</td>
<td>m-2</td>
<td>m-2/(m+4)</td>
</tr>
<tr>
<td>N_{44} N_{45}</td>
<td>m-3</td>
<td>m-3/(m+4)</td>
</tr>
<tr>
<td>N_{15}</td>
<td>m-4</td>
<td>m-4/(m+4)</td>
</tr>
</tbody>
</table>

To solve the resource waste problem, R. Walke, G. Lightbody et al [2][14] propose a method, based on the same computing nodes of the conventional triangular systolic array. Moreover, it maps the triangular array into a linear systolic array.
3.4.2 Derivation of Linear Architecture

In order to map the triangular array into a linear systolic array, there are some criteria for this mapping. The first one is that triangular array should have $2m^2+3m+1$ nodes and number of input channel equals to $2m+1$ ($m \geq 1$). Then it can be mapped onto a linear architecture with $m+1$ nodes.

The initial goal of mapping this triangular array into a smaller architecture is to manipulate the interconnection of the nodes and their positions so that they form a regular rectangular array. Then partition the nodes into groups according to their roles. Each group is assigned to each individual processing node. The key point is to align nodes along a column or a row for projection down onto a specific node. In addition, sequential operations should be pay attention to so that only local interconnections are present in the final architecture. The following four stages can achieve the linear array [2].

Fig. 3.5 triangular systolic array for QR algorithm (m=2)
By a cut made after $m+1^{\text{th}}$ boundary node at right angles to the below of the upper right diagonal line nodes, the QR triangular array is divided into two smaller triangles, $A$ and $B$ as Fig. 3.5. Triangle $A$ forms the bottom part of a rectangular array, with $m+1$ columns and $m+1$ rows. It can be clearly seen after rotate Triangle $A$ clockwise by 45 degree. Triangle $B$ now needs to be manipulated so that it can form the top part of the rectangular array. This is done in two stages. First, by mirroring triangle $B$ along the $x$-axis go through node $N_{(m+1),(2m+1)}$ parallel with the boundary nodes in the triangle $A$, form a parallelogram shape as shown in Fig. 3.6. The mirrored triangle $B$ is then moved up along the $x$-axis to above $A$ forming the rectangular array (Fig. 3.7). As depicted, the boundary cell operations are aligned down two columns; however, as a result the rectangular array is still not in a suitable format for assigning operations onto a linear architecture [2].

![Fig. 3.6 Array after part B mirroring rotation (m=2)](image)
The next stage is to fold the rectangular array along its central axis so that the two columns of boundary nodes are aligned to one column. This folding interleaves the nodes so that a compact rectangular array is produced (Fig. 3.8). Some global connections still contained in the array can be removed by projecting down the diagonal to assign all the boundary nodes to one boundary node and all internal nodes to a row of $m$ internal nodes [2]. The final linear array is shown in Fig. 3.9.

![Interleaving the rectangular array](image)

Fig. 3.7 Interleaving the rectangular array
Each node has its own delay and it is assumed that it is one clock cycle. The lines drawn through each row of nodes (Fig. 3.8) represent the data flows among the nodes after each cycle of the linear array. They form the schedule for the linear architecture. They may be denoted more compactly by a schedule vector. In Fig. 3.9, multiplexers are present at the top of the array so that inputs to the linear array can be chosen to the new input value or the result value from the previous row. The multiplexers at the bottom of the figure 3.9 choose the different directions of data flow that occur between rows of the original array due to the second folding.
A valid schedule is achieved by ensuring that data required by each set of scheduled operations are available at the time of execution [2]. This implies that the data must flow across the schedule lines in the direction of the schedule vector. The rectangular array in Fig. 3.8 contains all the computing nodes required by the QR algorithm. Fig. 3.8 shows that the nodes are to be implemented on the linear architecture by folding the rectangular array along the folding vector. Therefore, this diagram can be used to derive the continuous schedule for the linear architecture.

3.4.3 Scheduling for the Linear Array

Fig. 3.9 QR decomposition linear systolic array architecture for 5 channels
An analysis of the scheduling and timing issues can now be refined. Looking at the first schedule line of Fig. 3.8, it has the nodes from two different QR updates that have been
interleaved. One QR update means that all the stored values $r$ inside each node have been updated once or all nodes have been executed once. This can be more easily visualized by considering the schedule of the unfolded array shown in Fig.3.10 (a). Figure 3.10(b) shows the array and the schedule after array rotation and moving. The colored QR nodes belong to a previous QR update. By continuing the schedule shown in Fig.3.8 for a further $2m$ cycles, we obtain the schedule diagram shown in Fig. 3.11. Effectively successive QR updates have been interleaved as depicted by differently colored nodes. To summarize the operation, on linear architecture, the first QR operation begins at cycle=1, then the next QR operation begins after $2m+1$ cycles. Likewise, after a further $2m+1$ cycles, the third QR operation is started. Totally, it takes $4m+1$ cycles of the linear architecture to complete one specific QR update.

From Fig.3.11 we can see that there are two types of $x$ inputs into the QR nodes. The first type, referred to as external $x$ inputs, come from the data forming the input $x(n)$ matrix. These inputs are fed into the linear architecture every $2m+1$ ($m=2$ in Fig.3.11) clock cycles, (i.e. there are $2m+1$ cycles between two new external input values on the same input port or channel). The second type of $x$ input, referred to as internal $x$ input, results from the transfer of $x$ values from one internal node output port to another node input port. For the analysis of the schedule, rotation parameters $c$ and $s$ inputs as well as $x$ inputs will be considered.
White node means idle
Color node means executing

Fig. 3.11 Interleaved QR iteration (m=2)
3.4.4 Retiming the Linear Array

The linear architecture in Fig. 3.9 only has a single delay on each node to maintain the flow of data between performed operations from one cycle to the next. The mapping of the linear architecture is based on this delay number inside each node; hence, there will be no conflicts of the external \(x\) inputs with the internal \(x\) ones. However, the inclusion of actual timing details inside the QR nodes will affect the valid data schedule. The arithmetic operators used to build the QR nodes, such as multiplier or divider, need to be pipelined to meet the timing performance requirements and allow a higher clock rate of hundreds MHz [2]. An example of pipelined QR nodes after retiming is shown in Fig 3.12.

![Diagram of Retimed QR nodes](image)

**Fig. 3.12 (A)** An example of Retimed QR nodes when \(L_N = 6\) and \(T_{OR} = 5\)
In Fig. 3.12, both boundary nodes and internal nodes have the recursive loops within the architecture. They are feedback loops storing the $r$ value that comes from QR iteration. In a common sense, the architecture containing recursive loop is very hard to be pipelined since it will destroy the latency dependent of the data if some arithmetic computing operators are pipelined. To maintain the regular data schedule shown in Fig. 3.11, the latencies of the QR nodes can be slightly adjusted so that the $x$ and rotation parameters are output at the same time. The latency of boundary node producing rotation parameters $c$, $s$ and $x$ is set at $L_N$ to keep the structure of the schedule.

Fig. 3.12 (B) An example of Retimed QR nodes when $L_N = 6$ and $T_{QR} = 5$

---

---
Pipeline length inside node or known as the latency has a major effect on the scheduling of the linear array and stretches out the schedule of operation for each QR update. In Fig. 3.11, it can be found that the second iteration begins \(2m+1\) clock cycles after the start of first iteration. However, the introducing node latency \(L_N\) stretches out the scheduling diagram such that the second iteration begins after \((2m+1) L_N\) clock cycles as Fig. 3.13. This is definitely not an optimum use of the linear array since it will only be used every \(L_N\) clock cycles. This is related to the clock cycles between the successive QR updates which can be represented by \(T_{QR}\). For the conventional triangular array, the data from certain number of channels (one row of the input data matrix) are parallel inputted.
to the array on every clock cycle. The recursive loops have to be updated every clock cycle. However, for the QR-D linear array, the data of each channel on one QR update have to be inputted serially. This means, for each node, that the recursive loops do not have to be updated every clock cycle but updated every $T_{QR}$ clock cycle. This can help to pipeline the recursive loop to $T_{QR}$ length. Therefore, a high data sampling rate should be achieved. The goal is to find the proper combination of $L_N$ and $T_{QR}$ that can guarantee a valid schedule and 100 percent hardware utilization.

From G. Lightbody’s work [2], the value for $T_{QR}$ is the same as the number of input channels of the original QR triangular array, which is $2m+1$. This can be seen that it equates to the level of hardware reduction that was obtained from mapping of $2m^2+3m+1$ nodes triangular array down to $m+1$ nodes linear array. Also, by setting the node latency $L_N$ to a relatively prime number of $T_{QR}$, a valid schedule can be achieved. If these two numbers are not relatively prime number, then there will be data collision at the products of $L_N$ and $T_{QR}$ with their common multiplies.

Furthermore, choosing an optimum value of $L_N$ and $T_{QR}$ is to ensure that the nodes are fully utilized. Fig 3.14 shows the example schedule over several QR updates when $L_N$ is 6 and $T_{QR}$ is 5. The different colored nodes represent different QR iterations that are interleaved with each other to fill the blank nodes left by the highlighted QR iteration. When there is no blank node, the array is under 100 percent hardware utilized. We can find this array is on 100 percent utilization after 45 cycles. The smaller $L_N$, the faster
hardware goes into 100% utilized, but smaller pipeline stages can be achieved inside computing nodes.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>34</td>
<td>25</td>
<td>31</td>
<td>11</td>
<td>34</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>44</td>
<td>12</td>
<td>35</td>
<td>32</td>
<td>44</td>
<td>12</td>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>22</td>
<td>45</td>
<td>13</td>
<td>33</td>
<td>22</td>
<td>45</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>55</td>
<td>23</td>
<td>14</td>
<td>34</td>
<td>55</td>
<td>23</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>33</td>
<td>15</td>
<td>24</td>
<td>35</td>
<td>33</td>
<td>15</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>34</td>
<td>25</td>
<td>36</td>
<td>11</td>
<td>34</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>44</td>
<td>12</td>
<td>35</td>
<td>37</td>
<td>44</td>
<td>12</td>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>22</td>
<td>45</td>
<td>13</td>
<td>38</td>
<td>22</td>
<td>45</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>55</td>
<td>23</td>
<td>14</td>
<td>39</td>
<td>55</td>
<td>23</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>33</td>
<td>15</td>
<td>24</td>
<td>40</td>
<td>33</td>
<td>15</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>34</td>
<td>25</td>
<td>41</td>
<td>11</td>
<td>34</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>44</td>
<td>12</td>
<td>35</td>
<td>42</td>
<td>44</td>
<td>12</td>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>22</td>
<td>45</td>
<td>13</td>
<td>43</td>
<td>22</td>
<td>45</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>55</td>
<td>23</td>
<td>14</td>
<td>44</td>
<td>55</td>
<td>23</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>33</td>
<td>15</td>
<td>24</td>
<td>45</td>
<td>33</td>
<td>15</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>11</td>
<td>34</td>
<td>25</td>
<td>46</td>
<td>11</td>
<td>34</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>44</td>
<td>12</td>
<td>35</td>
<td>47</td>
<td>44</td>
<td>12</td>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>22</td>
<td>45</td>
<td>13</td>
<td>48</td>
<td>22</td>
<td>45</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>55</td>
<td>23</td>
<td>14</td>
<td>49</td>
<td>55</td>
<td>23</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>33</td>
<td>15</td>
<td>24</td>
<td>50</td>
<td>33</td>
<td>15</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>11</td>
<td>34</td>
<td>25</td>
<td>51</td>
<td>11</td>
<td>34</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>44</td>
<td>12</td>
<td>35</td>
<td>52</td>
<td>44</td>
<td>12</td>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>22</td>
<td>45</td>
<td>13</td>
<td>53</td>
<td>22</td>
<td>45</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>55</td>
<td>23</td>
<td>14</td>
<td>54</td>
<td>55</td>
<td>23</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>33</td>
<td>15</td>
<td>24</td>
<td>55</td>
<td>33</td>
<td>15</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>11</td>
<td>34</td>
<td>25</td>
<td>56</td>
<td>11</td>
<td>34</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>44</td>
<td>12</td>
<td>35</td>
<td>57</td>
<td>44</td>
<td>12</td>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>22</td>
<td>45</td>
<td>13</td>
<td>58</td>
<td>22</td>
<td>45</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>55</td>
<td>23</td>
<td>14</td>
<td>59</td>
<td>55</td>
<td>23</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>33</td>
<td>15</td>
<td>24</td>
<td>60</td>
<td>33</td>
<td>15</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 3.14 Retimed scheduling for the linear array, $L_N=6$, $T_{QR}=5$
4. QR DECOMPOSITION IMPLEMENTATION

4.1 Introduction of FPGA Design

In recent years, rapid advancement of Field Programmable Gate Array (FPGA) technology is enabling highly reliable re-programmable solutions. By implementing over the logical level not the physical circuits, users can better focus on the algorithm and architecture optimization. It helps to accelerate the system design.

In this thesis, The QR decomposition based on Givens rotations algorithm is implemented on both conventional QR systolic triangular array and linear QR array on Xilinx Virtex 6 FPGA. The input and output data as well as arithmetic operation of QR processors are in fix-point format. Fix-point square root inverse is implemented by polynomial approximation. Compared with CORDIC or square-root-free algorithm, this implementation is better understood and easier to establish. Challenges include how to pipeline each computation node and how to implement linear systolic by folding triangular array. The retiming and rescheduling on linear systolic also require deep understanding of latency dependence between each node.

4.1.1 FPGA Introduction

Field Programmable Gate Array short for FPGA belongs to the programmable logical devices family. It is generated from the rapid evolution of very large scale circuit (VLSI) and computer aided design (CAD) technology. FPGA has properties of high integration,
high speed of processing, reprogrammability, and low cost. Circuit designers can do the logical programming, compiling, optimization, simulation and verification based on the limited funding and short development period. This is an efficient method for VLSI prototyping. FPGA chips also integrate massive physical IP (Intelligent patent) cores to decrease power consummation and increase reliability. It also helps to increase processing speed.

The basic logical unit of FPGA is a look-up table. By loading the pre-defined logical bits stream stored in external memory generated from CAD software, FPGA can easily establish the corresponding logical relation inside the chip. Power off the chip, the logical relation will be wiped. In this way, FPGA can be reusable or reprogrammable.

4.1.2 FPGA Design Process

The design of FPGA is based on the idea of top – down. It contains circuit functions design, design input, function simulation, synthesis optimization, synthesis post simulation, implementation design, post simulation of place & route, board level simulation and chip programming and debug. Fig 4.1 shows a general design flow graph.
4.1.3 Introduction of Modelsim and ChipScope

Modelsim is simulation software for chip logical design that provides high speed, accuracy and convenient simulation. It utilizes the optimized compiling technique, Tcl/Tk technique and single core simulations. Based on friendly graphic user interface, Modelsim provides high speed simulation, easier debug function, hardware resource consumption analysis and so on. However, it only serves for simulation and compiling but not for place & route or programming the chip. Therefore, Modelsim has to utilize with ISE design suit.

The traditional FPGA debug tool uses logic analyzer to sample and verify digital signal directly from the hardware. This kind of tool is very expensive and not convenient
for testing since too many chip probes need to be connected. Therefore, Xilinx provides ChipScope Pro logical analyzing tool. This software based tool inserts a particular IP core, connecting to the net or port that needs analyzing, into the FPGA design. Users can analysis the signal sampled by the analyzer IP core via JTAG (Joint Test Action Group) port under ChipScope Pro analyzer GUI (graphic user interface). In this method, under the system’s required clock frequency, users can directly analysis the signals inside FPGA chip without any interference on the circuit function. This software based analyzer can implement high level trigger functions to filter out and display any design problems without re-synthesis but simply by changing the soft probe.

Utilizing the advanced functions of Modelsim and ChipScope Pro, the QRD design described in this thesis is developed under Xilinx ISE FPGA design suit, simulated on Modelsim and verified under ChipScope Pro.

4.2 Implementation by the Triangular Array

In J. G. McWhirter’s published paper, the triangular array is well defined. In Chapter 3, the details of triangular array for QR decomposition are discussed specifically. However, there are still several problems left unsolved. First, we can see each computing node only has one latency or one delay. This means the result value will show up one clock cycles later than the input value. Thu, this one clock cycle should long enough to meet the least time consisting of the max length of the combination logical circuit inside each node. In
this case, the main clock frequency has to maintain at low speed and the data throughput rate will be hurt. In order to avoid this situation, in the following design, pipeline will be applied into each computing nodes, and the whole macro architecture will be slightly modified.

The second issue involves the square root inverse. To compute the rotation parameters, the square root inverse is inevitable. However, square root inverse is a very complicated operation for VLSI implementation when the speed and accuracy are concerned. So far, there are three ways to solve the square root inverse for QR decomposition, which are CORDIC algorithm, Logarithm and root square free QRD. Considering convenience, speed and latency, a Logarithm method combining look-up table will be used for implementing square root inverse in this thesis.

4.2.1 Boundary Node

Usually, for the real system, the data input into the QR decomposition array usually do not directly come from ADC (analog to digital converter). There are certain devices between them like digital filter. Thus, a signed 16 bits fixed-point integer was chosen as the input data format since 14 bits ADC is commonly used in radio system and 2 bits is reserved for redundant. Also a fixed number containing signed 19 bits integer with 3 bits decimal was used as the $r$ output data format. Signed 32 bits fix point decimal number was applied for $s_{\text{real}}$ and $s_{\text{im}}$ output. Unsigned 32 bits fix point decimal number was used for $c$ value output.
At the beginning of this boundary node, since it is simply a magnitude square computing, we can ignore the signed bits but reserve the integer part which is 37 bits. There are also some truncations on the precision of $r$, $c$, $s_{real}$ and $s_{im}$ values. Based on the reference [5], the precision of $r$, $c$, $s_{real}$ and $s_{im}$ values described on the Fig 4.2 can meet the computation requirement.

![Diagram](image)

Fig. 4.2 Pipelined boundary node
There is also a difference of the boundary node in Fig. 4.2 comparing with the conventional boundary node. The conventional triangular QRD array requires that each node has the same latency on the output rotation parameters $c$ and $s$. However, in Fig. 4.2, the pipeline length on $c$ value is 15 clock cycles but the pipeline length on $s$ value is 12 clock cycles. This is not a fault but a compromise on the reclusive loop inside the internal node. This will be discussed in following section. We can simply regard the boundary computing latency or pipeline length is 12 clock cycles for rotation parameters computing except for $r$ because $r$ value will not involve in the computation of the internal node. It is simply the result value on the diagonal of the upper triangular matrix. In a word, this method will pipeline the boundary node to 12 stages. This helps to increase the input data sampling frequency.

**4.2.2 Square Root Inverse**

As we know, square root inverse is a complicated operation for VLSI implementation. But it is still feasible. In this thesis, a polynomial approximation will help to solve square root inverse.

For common square root inverse, the input value range is over $(0, +\infty]$ and the output value range is also over $(0, +\infty]$. In our boundary node configuration, the input of square root inverse is over $[1, 2^{38}]$. Thus, the output is over $[2^{-19}, 1]$. Even though it seems the input and output range is fixed and smaller than the normal square root inverse. However, it is still a problem to do this operation, because there is no any basic operator for VLSI
circuit. It must be done by a combination of many basic operators, for example multiplier, divider, and adder. However, the latency and speed will be a big problem. CORDIC algorithm is better choice since it can really provide a high precise result but the latency or called delay is too much. The same operation done by CORDIC ip core and divider ip core from Xilinx will take almost 20 clock cycles. It provides more accuracy, but sacrifices on the latency and speed. We need an alternative way with low latency but high clock frequency.

Polynomial approximation is better method to balance the latency and precision. For any input value, it can be mapped to \([1, 2)\) simply by shifting certain positions. For a particular value \(r_{sq}\) \((r_{sq} \geq 1\), which is an unsigned number) represented by binary form, it definitely always has the most significant bit which is 1. For example \(r_{sq} = (9999)_{10} = (10011100001111)_{2}\), we can always shift the decimal point to the right of the most significant bit. Then we get \(r_{sq\_normalized} = (1.0011100001111)_{2}\). This normalized value is always on the range of \([1, 2)\). Thus, the problem is simplified to do square root inverse over range of \([1, 2)\).
$y = Ax + B$

$X = x_{37}x_{36}x_{35} \ldots x_3x_2x_1$

$X_{normalized} = 1.x_{37}x_{36}x_{35} \ldots x_3x_2x_1$

**Fig. 4.3 Polynomial Approximation to $1/sqrt()$ [5]**

**Fig. 4.4 Square root inverse inside boundary node**
In Fig 4.3, it shows how to compute square root inverse by polynomial approximation. For $y = 1/\sqrt{x}$, we first can calculate the y values corresponding to the x input values having equal small distance over [1, 2) on MATLAB. Then straight lines can be drawn between the spots consist of $(x, y)$ on the axis. The curve of $y = 1/\sqrt{x}$ can be approximated by this broken line. The values between two adjacent spots can be interpolated and approximated by the small straight line. For the FPGA implementation, we can simply store the slope and y-axis interception value of each small straight line to a look-up table. The square root inverse result can be calculated on following steps. First, map input $r_{sq}$ to a normalized value on interval [1, 2]. Second, address by the normalized input value to the particular memory location corresponding to a certain interval on the approximated broken line. Second, fetch the stored value on this location. Third, get the value square root inverse by calculate a linear equation. Fourth, realign the result back to the true range $[2^{-19}, 1]$. During first step, the shift distance should be recorded, since realignment needs this particular value. The normalized and realignment can be expressed in following mathematic equations.

\[
\text{inverse}_r = \frac{1}{\sqrt{r_{sq}}}
\]

\[
\text{inverse}_r = \frac{1}{\sqrt{r_{sq\_normalized}} \times 2^{\text{shift\_distance}}}
\]

\[
\text{inverse}_r = \left( \frac{1}{\sqrt{r_{sq\_normalized}}} \right)^{-\frac{\text{shift\_distance}}{2}}
\]

(3.3)

$\text{shift\_distance}$ could be an odd number. If this happens on realignment, we can do this
\[
\text{inverse}_r = \left( \frac{1}{\sqrt{r_{sq\_normalized}}} \right) \times 2^{\frac{\text{shift\_distance}-1}{2}} \times 2^{\frac{1}{2}} \quad (3.4)
\]

Then simply left shift the result by \((\text{shift\_distance}-1)/2\) and multiple it by \(1/\sqrt{2}\).

Based on Huy’s work\[5\], A look-up table with 4096 depth and each space store a slope value (“A” in Fig. 4.3) of 32 bits fix-point decimal and a y-axis interception value (“B” in Fig. 4.3) of 32 bits fix-point decimal is built. This square root inverse component has maximum 38 bits integer input and provides unsigned 32 bits fix-point decimal number.

The RTL schematic of the boundary node from Xilinx ISE design suit is shown in Fig 4.5

![RTL schematic of the boundary node](image-url)
4.2.3 Internal Node

The internal node is shown in Fig 4.6. All operators such as multiplier, adder and subtractor are implemented by Xilinx IP core. The multipliers are implemented by on chip DSP cell which can help to increase frequency and decrease latency. Adders and subtractors are formed by logical slices.

However, there are two recursive loops containing multiplier and adder on the path for calculating $r_{real}$ and $r_{im}$. They will affect the internal clock frequency since the recursive loop is not easily to be pipelined under the triangular systolic array configuration. Not only does the recursive loop decrease clock frequency, but it also badly affects their parallel paths, as in the left branches on calculating $r_{real}$ and $r_{im}$. If all the inputs, for example $c$, $s_{real}$, $s_{im}$, $x_{real}$, $x_{im}$, arrive at same time, the recursive loop Path 1 (Fig 4.6) must be finished in one clock cycle, Path 2 also must be done in one clock cycle. However, Path 2 is much longer than Path 1. This is not an efficient method to finish Path 2 in one clock cycle. To avoid this happening, make the $c$ arrives 3 clock cycles later than $s_{real}$ and $s_{im}$. In this way, the Path 2 can be pipelined by 3 levels and will not be affected by the recursive loop too much. Pipeline of the recursive loop can only be done under linear systolic array. This is discussed in Chapter 3.
Fig. 4.6 Internal node
The internal node is also pipelined to 12 levels. A RTL schematic of internal node from Xilinx ISE design suit is shown in Fig 4.7.

![RTL schematic of the internal node](image)

**Fig. 4.7 RTL schematic of the internal node**
4.2.4 Triangular Array Retiming

The whole triangular systolic also must be retimed after each node is pipelined for ensuring the whole timing correct. Modified from the triangular systolic array discussed in Chapter 3, a 4 by 4 retimed triangular systolic array is shown in Fig 4.8. For the boundary node the rotation parameter output port latency is 12 clock cycles. After 12 clock cycles, the rotation parameter of each QR update will output on every clock cycle. For the internal node, the rotated $x$ value will also show up every clock cycles after the first 12 clock cycles latency. And the rotation parameters $c$ and $s$ from boundary node are passed between internal nodes sequentially every clock cycle.

![Diagram of a 4 by 4 QR-D triangular systolic array](image)

Fig. 4.8 a 4 by 4 QR-D triangular systolic array
Due to the retiming, the data can be input into the triangular systolic array parallel which slightly differs from McWhirter’s work. But the results of upper triangular are still outputted sequentially if only QR decomposition is performed.

The RTL schematics are shown in Fig. 4.9 and Fig. 4.10.
4.3 Implementation on the Linear Array

Although the conventional triangular array, the highly parallel multiple inputs structure will help to increase data throughput. However, the recursive loops inside nodes will decrease clock frequency and data sampling rate while the large size of triangular array architecture will also consume too much logical resource on FPGA. As it is discussed in chapter 3, linear array is an efficient method to solve the above problem but sacrifices on the property of parallel input.
4.3.1 Boundary Node

There is not much change on the boundary design besides adding pipeline to the recursive loop and modifying some pipeline stages on particular operators. The implemented boundary node is shown in Fig 4.11. This particular boundary node is designed from conventional 5 input channels QR decomposition array. Pipeline the whole structure to 14 stages and set $T_{QR}=5$ where this means recursive loop have 5 stages pipeline. In boundary node, the conventional configuration is to input the data on every clock cycle and output the rotation parameters on every clock cycles after certain latency clock cycles caused pipeline. For this linear array configuration, due to the interleaved QR iterations, the new data is input to the boundary node every $T_{QR}$ clock cycles. For example, if the first clock cycle is to input the data $x$ from a new QR iteration, then the next input data will be the output data from other internal node and so on. The next new QR iteration will be input on the first cycle of the next $T_{QR}$ clock cycles. $T_{QR}$ is usually equal to the number of input channel numbers which is also the number of columns of the triangular array. The detail is also discussed in Chapter 3.
Fig. 4.11 Boundary node of linear array
4.3.2 Internal Node

There is also not much modification on internal besides pipelining the recursive loops. This is shown in Fig. 4.12. The internal node is pipelined to 14 stages. All input data arrive at same time every clock cycle. The result data \( x\_real\_out \) and \( x\_im\_out \) will be output on every clock cycles after the 14 clock cycles latency due to pipeline. \( r\_real \) and \( r\_im \) will be generated only after 5 clock cycles.

Fig. 4.12 Internal node of linear array
4.3.3 Linear Array Scheduling

The linear array implemented in the following consists of one boundary node and two internal nodes. As discussed in Chapter 3, this three nodes linear systolic array is folded from the 5 channel input triangular systolic array. The folding processing and rescheduling have already been shown in Chapter 3. However, the scheduling detail is different from what discussed in Chapter 3 since both boundary node and internal node are retimed to 14 stages pipeline and $T_{QR}=5$. The architecture graph is the same as Fig 3.9. The schedule graph in Table 4.1 is a little different from Fig 3.14.

In Table 4.1, for example, at “clock cycle 1  1.1(1)” means data is inputted to boundary node which performs the same function as the Node 1.1 of the triangular systolic array at clock cycle 1. The number inside the brackets following Node number represents the corresponding QR iteration. “1.1(1)” means the 1st QR iteration or QR update on Node 1.1. The blank on node means no data are inputted to this node which could be regarded as input zero data. From the Table 4.1, we can see the linear array is fully utilized and there will be one sampled data inputted to the array on each clock cycles after 109 clock cycles. Therefore, it sacrifices on the parallel input but achieves high clock frequency and less logical resource consumption.

To switch the node of linear array performing different nodes of triangular array and control the data flowing direction, the external multiplexers in Fig. 4.13 should be also very well scheduled.
Table 4.1 schedule of linear systolic array over 120 clock cycles

<table>
<thead>
<tr>
<th>cycle</th>
<th>N1</th>
<th>N2</th>
<th>N3</th>
<th>cycle</th>
<th>N1</th>
<th>N2</th>
<th>N3</th>
<th>cycle</th>
<th>N1</th>
<th>N2</th>
<th>N3</th>
<th>cycle</th>
<th>N1</th>
<th>N2</th>
<th>N3</th>
<th>cycle</th>
<th>N1</th>
<th>N2</th>
<th>N3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.1(1)</td>
<td>30</td>
<td>1.1(7)</td>
<td>60</td>
<td>1.1(13)</td>
<td>90</td>
<td>1.1(19)</td>
<td>3.4(5)</td>
<td>2.5(5)</td>
<td>1</td>
<td>1.1(9)</td>
<td>70</td>
<td>1.1(15)</td>
<td>3.4(1)</td>
<td>2.5(1)</td>
<td>100</td>
<td>1.1(21)</td>
<td>3.4(7)</td>
<td>2.5(7)</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>62</td>
<td>2.3(5)</td>
<td>92</td>
<td>2.3(11)</td>
<td>1.4(11)</td>
<td>13</td>
<td>2.2(4)</td>
<td>1.3(4)</td>
<td>73</td>
<td>2.2(10)</td>
<td>1.3(10)</td>
<td>103</td>
<td>2.2(16)</td>
<td>4.5(2)</td>
<td>1.3(16)</td>
<td>14</td>
<td>1.2(7)</td>
<td>74</td>
</tr>
<tr>
<td>3</td>
<td>33</td>
<td>63</td>
<td>2.2(8)</td>
<td>93</td>
<td>2.2(14)</td>
<td>1.3(14)</td>
<td>18</td>
<td>2.2(5)</td>
<td>1.3(5)</td>
<td>78</td>
<td>2.2(11)</td>
<td>1.3(11)</td>
<td>108</td>
<td>2.2(17)</td>
<td>4.5(3)</td>
<td>1.3(17)</td>
<td>19</td>
<td>1.2(8)</td>
<td>79</td>
</tr>
</tbody>
</table>
Fig. 4.13 Linear QR decomposition array architecture for 5 input channels
The RTL schematic of the whole linear systolic array after synthesis is shown in Fig. 4.14.

Fig. 4.14 Linear QR decomposition array RTL schematic
5. RESULT AND PERFORMANCE ANALYSIS

These two kind of systolic arrays were implemented under Xilinx ISE design suit by customized Verilog code and Xilinx IP core including Multiplier, Adder, Block memory generator, ChipScope and a Digital clock manger (DCM) IP core. The test matrices were built into ROM core within design files. All the behavioral simulation and post-route simulation were done in Modelsim. The ChipScope was used by capture the output upper triangular matrix value after the bit stream loaded into Xilinx ML605 Virtex6 xc6vlx240t development board. The hardware test bench is shown in Fig.5.1. The main clock came from on board 200MHz differential clock. DCM used 200MHz source to generate the desired clock frequency to drive the whole structure. The control keys outside FPGA were “reset” and “start” buttons which are used for starting the QR-D and resetting everything. Status LEDs were used for observing the “running”, “stop” and “idle” status of the QR-D. The software based analyzer was used for capturing the result data matrices.
Fig. 5.1 FPGA testbench
5.1 Result Analysis on Triangular Array

This 4 input channel triangular systolic array will take 103 clock cycles to finish the QR decomposition of a 20 by 4 matrix. Usually, people choose 5 times number of channels as the number of the rows for the input matrix due to the training sequence length requirement. The timing simulation is shown in Fig. 5.2 and Fig. 5.3.

![Fig. 5.2 input matrix timing graph](image)

In fig 5.2, a 20 by 4 size matrix is parallel input into this triangular systolic array in 20 clock cycles after “qr_core_en” goes high level and “reset_qr_core” goes low level. Here, we see the input data format is signed 22 bits fix point. However, this is converted from the actual input signed 16 bits fix point number by adding 2 bits on integer part and 3 bits on decimal part. This is simply a redundant protection. Thus, the actual data can be achieved by dividing 8.
In the above screenshot, it will take 2060ns which is 103 clock cycles to finish one QR matrix decomposition under 50MHz sampling frequency. Due to the triangular systolic array, the result data do not show up at the same time but emerge sequentially on the output ports from “r_out_node_11” to “r_out_node_41”. In this case, a “write_mem_en” signal indicates there is an output value when it goes high level. In Fig 5.3, in a complete QR-D process, there are 10 times that “write_mem_en” goes to high level. This is consistent with the fact that the upper triangular matrix of a 20 by 4 matrix only has 10 elements.

Fig. 5.3 output matrix timing graph
Table 5.1 a simulation result of QR-D on a 20 by 4 matrix

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>964162</td>
<td>-2597-105764i</td>
<td>-230616-84680i</td>
<td>-270605-55330i</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>821846</td>
<td>-39185-37720i</td>
<td>73906+116647i</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>941556</td>
<td>39289-162195i</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>871580</td>
<td></td>
</tr>
</tbody>
</table>

We can see the simulated output is also consistent with the capturing data by ChipScope from FPGA test. In Fig. 5.4, the diagonal value (imaginary part is zero) can be seen. Due to the ChipScope core configuration, ChipScope only captures the final result of the QR-D. However, the actual triangular array will continuously output some intermediate calculation data due to the pipeline as the simulation timing diagram Fig 5.3 showing. But this will not affect the final result.

![ChipScope captured data from triangular systolic array](image)

**Fig. 5.4 ChipScope captured data from triangular systolic array**

Table 5.2 FPGA result of QR-D on a 20 by 4 matrix

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>964162</td>
<td>-2597-105764i</td>
<td>-230616-84680i</td>
<td>-270605-55330i</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>821846</td>
<td>-39185-37720i</td>
<td>73906+116647i</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>941556</td>
<td>39289-162195i</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>871580</td>
<td></td>
</tr>
</tbody>
</table>
In fig. 5.5, the upper triangular matrix from MATLAB is shown.

![Table 5.3 MATLAB program result of QR-D on a 20 by 4 matrix](image)

Although the details of timing simulation cannot be screen shot, the matrix computed from MATLAB is consistency with the captured data from ChipScope. Surely, there are certain errors between them.

In order to evaluate the average error, 1000 matrices were sent into the internal memory build into my design. ChipScope captured the result matrices and compared with the double precision QR-D program under MATLAB. The captured result values are in fixed point format of signed 19 bits integer and 3 bits decimal. The mean square error between these two result matrixes was calculated. The mean square error is the average standard deviation on single value between FPGA result matrix and MATLAB result matrix. The relative error is the mean square error divided by maximum value.
represented by the fix-point format. The absolute error is 7.608. The relative error comparing the maximum value represented by this 22 bits data format is 2.902e-05. Even though the absolute error seems to be large due to the imperfection of the square root inverse, it still meets the requirement of the real application.

Dynamic range of an antenna array can be generalized to the value dropping between Eigenvalue one and Eigenvalue two of the covariance matrix obtained from the received data matrix. Assume a scenario of single transmitting source. The antenna array will simply produce a rank one matrix when beamforming is not used. Obtain the covariance matrix by formula $M=X^HX$ where $X$ is the received data matrix. Apply a SVD on this covariance matrix and find the Eigenvalue distribution. In this distribution, the received signal energy is mainly on the Eigen one and the other Eigenvalues should be around the noise floor. Thus, the dynamic range of the received signal can be obtained by evaluating the Eigen one value.

Therefore, the hardware on the receiver should provide high dynamic range than the received signal. Based on the reference paper [20] [21], the dynamic range requirement should be larger than 61dB. Meanwhile, a general 14 bits ADC can provide effective 12 bits which is 12×6=72dB. In this case, the dynamic range requirement should be 72dB.

To test the dynamic range of this linear systolic array, rank one complex matrices with increasing magnitude but random phase can be sent to the QR-D and then evaluate the Eigenvalue distribution. By increasing the input magnitude and finding the dropping
between Eigen one and Eigen two, the dynamic range of this QR-D can be obtained. In this testing, send 1000 rank one matrices with random phase where every 100 rank one matrices have different magnitude from $2^1$ to $2^{15}$. Input these matrices into the QR-D. Get the result of upper triangular matrix through ChipScope and then import data into MATLAB. Plot the Eigenvalues distribution and find the Eigenvalue dropping level.

![Eigen value drop vs input amplitude](image)

**Fig. 5.6 Eigen one and Eigen two difference vs input amplitude for triangular systolic array**

From fig.5.6, the maximum of Eigenvalue drop is 115dB. It only requires 72 dB for the dynamic range. Thus, this QR-D based on triangular systolic array meets the dynamic range requirement.

Finally, this triangular systolic array occupies 30% of total Slices (total 37680) and 27% of total DSP48E1 (total 768) on Virtex6 xc6vlx240t. The tested clock frequency is 50MHz. The post-place & route static timing report show the maximum of the running
clock is 65.062MHz. Apparently, the recursive paths inside both kinds of computing nodes still slow the clock frequency.

![Image](image.png)

Fig. 5.7 Processing timing diagram for sixteen matrices of 20 by 5

5.2 Result Analysis on Linear Array

The main difference between the linear array and triangular array is that the input data need to be rescheduled. This makes the linear array need a complicated controller but also makes the recursive loop can be pipelined. This is discussed in Chapter 3 and Chapter 4.

The input and output data formats are the same as triangular systolic array.

This linear systolic array mainly consists of one boundary node and two internal nodes. This is equal to a 5 by 5 triangular systolic array. For decomposing a 25 by 5 complex matrix, the linear array will take 294 clock cycles. From the following timing simulation diagram (Fig. 5.8 (A)), we can briefly see the timing relation. There are only

69
two data input port “x_real” and “x_im” (Fig. 5.8 (B)) connected to the memory stored the matrices. The controller will pass the values of these two ports to each node by rescheduling diagram in Table 4.1. The controller also keeps sending the control signal periodically for all six multiplexers. The control signal is “mux_control” in Fig. 5.8 (C). These multiplexers are used for exchanging the c s and x_out values among nodes as Table 4.1 described. In Fig. 5.8 (D), “write_mem_en” is used to tell ChipScope to capture the output data. “write_mem_en” has a width of 3 and each bit represents the outputting status on each node.
Fig. 5.8 timing diagram of a complete QR decomposition

- 249 clock cycles on a 25 by 5 complex matrix
- 100MHz sampling frequency.
The screen shots show a complete upper triangular output process on the Modelsim.

\[ R_{11} = 1037532 \]
\[ R_{12} = 86739 + 220677i \]
\[ R_{13} = 21572 + 1644i \]
\[ R_{22} = 1056742 \]
\[ R_{23} = 119328 - 16247i \]
\[ R_{23} = 50091 + 40066i \]
\[ R_{33} = 185781 + 227385i \]
\[ R_{23} = 88279 - 212084i \]

Fig. 5.9 result matrix output procedure (A)
<table>
<thead>
<tr>
<th>Node</th>
<th>Real Part</th>
<th>Imaginary Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_{33}</td>
<td>1077028</td>
<td>0</td>
</tr>
<tr>
<td>R_{34}</td>
<td>93563 + 28698i</td>
<td>93563 + 28698i</td>
</tr>
<tr>
<td>R_{25}</td>
<td>-289956 - 242146i</td>
<td>-289956 - 242146i</td>
</tr>
<tr>
<td>R_{55}</td>
<td>14878 + 276723i</td>
<td>14878 + 276723i</td>
</tr>
</tbody>
</table>

Fig. 5.10 result matrix output procedure (B)
### Table 5.4 QR-D result of 25 by 5 matrix from Modelsim

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1037532</td>
<td>-86739+220677i</td>
<td>21572+1644i</td>
<td>50091+40066i</td>
<td>185781+227385i</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1056742</td>
<td>119328-16247i</td>
<td>88279-212084i</td>
<td>-289956-242146i</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1077028</td>
<td>93563+28698i</td>
<td>-14878+276723i</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1120321</td>
<td>162312+103157i</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1001787</td>
</tr>
</tbody>
</table>

The upper triangular Matrix decomposed by MATLAB program is shown in Fig. 5.11.

This result is almost consistent with the result Table 5.2 from Modelsim.

### Table 5.5 QR-D result of 25 by 5 matrix from MATLAB

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1037532</td>
<td>-86739+220677i</td>
<td>21572+1644i</td>
<td>50091+40066i</td>
<td>185781+227385i</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1056742</td>
<td>119328-16247i</td>
<td>88279-212084i</td>
<td>-289956-242146i</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1077028</td>
<td>93563+28698i</td>
<td>-14878+276723i</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1120321</td>
<td>162312+103157i</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1001787</td>
</tr>
</tbody>
</table>

Fig. 5.11 result matrix from MATLAB

Fig. 5.12 result matrix from Chipscope
Table 5.6 QR-D result of 25 by 5 matrix from FPGA

<table>
<thead>
<tr>
<th>1037560.31091402</th>
<th>86688.9091399078+220821.948432245i</th>
<th>21643.9121232548+1737.07689957059i</th>
<th>50176.2153509302+40132.0289008728i</th>
<th>185780.778378264+227515.563728576i</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>119435.562532336-16244.2752855852i</td>
<td>88290.6886499289-220821.948432245i</td>
<td>-290046.101570470-242251.650211864i</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1077134.88546740</td>
<td>93663.3568240271+28688.9169043808i</td>
<td>-14877.4705867499+276873.092396355i</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1120520.91770641</td>
<td>162477.620951739+103288.616194769i</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1002012.62683965</td>
</tr>
</tbody>
</table>

The above two result matrices are also consistent with data captured by ChipScope in Fig. 5.12.

Based on the data of 1000 upper triangular matrices from FPGA, the absolute error is 11.173 and relative error is 4.262e-05. The Eigen drop graph is shown in fig. 5.13. We can still calculate the dynamic range about 110dB. It still meets our requirement which is 72dB.

Fig. 5.13 Eigen one and Eigen two difference vs input amplitude for linear systolic
Finally, this linear systolic array occupies 12% of total Slices (total 37680) and 7% of total DSP48E1 (total 768) on Virtex6 xc6vlx240t. The tested clock frequency is 100MHz. The post-place & route static timing report show the maximum of the running clock is 135.373MHz. The tested clock frequency is 100MHz. Compared with the triangular array, it certainly consumes less FPGA logical resources and runs at higher clock frequency, which means a higher data sample rate. But it sacrifices the parallelism of the triangular array. This linear array can provide maximum data throughput is almost 2Gbit/sec. The triangular array can provide maximum data throughput is almost 4Gbits/sec. Although the triangular systolic array seems to have higher data throughput, its recursive loops of triangular systolic cannot be pipelined and this bottleneck cannot be easily broken through. The linear array can still be added more pipeline stages to increase the clock frequency and consumes much smaller resources when the data matrix becomes larger. Therefore, linear systolic will be the best choice for the future work.
6. CONCLUSION AND FUTURE WORK

6.1 Conclusion

Based on the result analysis on Chapter 5, the dynamic range of both triangular array and linear array can meet the requirement which is 72dB. However, the sampling frequency (equal to system clock frequency in this report) is not very high. The triangular array has a maximum clock frequency of 65MHz and 4 parallel input channels. The linear array has a higher maximum clock frequency which is 135MHz but only one input channel. For the common communication system, this sampling frequency may be not a problem. But for many wideband systems like phased array radar, the QR-D implemented on this thesis still needs more modification to increase the sampling frequency. Meanwhile, more accurate floating-point arithmetic will also be part of the next goal for this QR-D.

6.2 Future Work

For the adaptive beamforming array, QR decomposition processors array does not only work on matrix triangularization but also computes the beamforming weight. The QR decomposition implemented in this thesis can also be developed furthermore to fulfill weight computation. Fig 6.1 introduces the general method for beamforming weight computation. Constraints (steering vector) should be added to the decomposed triangular matrix. Then a double back-substitution is needed.
6.2.1 Back-substitution

The back-substitution is the important step for extracting the beamforming weights from the QR-D systolic array. After a complete matrix data is decomposed in the systolic array and all the elements of upper triangular matrix are stored on the array, the triangular systolic array will be on the second mode of “frozen mode”, where the r values stored inside each node will not be updated anymore. Firstly, we must analyze the function of the triangular array under the “frozen mode”.

Fig. 6.1 adaptive beamforming system [13]
Assuming a 5 channel triangular systolic array, the first 4 channels are used but the last channel is left as Fig 6.2. On the left triangular array, each node has the elements from the upper triangular. For the single column on right, we assume each node store a \( u \) value on the same location of \( r \) value. When the whole systolic array is under frozen mode, \( c \) and \( s \) pass among nodes become \( I \) and \( x/r \). \( r \) value inside the boundary node will not be updated. For the internal node, the \( x_{\text{out}} \) equals \( x-sr \).

![Diagram](image_url)

Fig. 6.2 “Frozen” triangular array [1]
In Fig 6.2, for the triangular array, we can follow the relations, for example, [1]

\[
\begin{align*}
  s_1 &= \frac{x_1}{r_{11}} \\
  s_2 &= \frac{x_2 - r_{12}s_1}{r_{22}} \\
  s_3 &= \frac{x_3 - r_{13}s_1 - r_{23}s_2}{r_{33}}
\end{align*}
\]

(6.1)

Then

\[
\begin{align*}
  x_1 &= r_{11}s_1 \\
  x_2 &= r_{12}s_1 + r_{22}s_2 \\
  x_3 &= r_{13}s_1 + r_{23}s_2 + r_{33}s_3
\end{align*}
\]

(6.2)

Apparently [1]

\[
\begin{align*}
  R^T s &= x \\
  s &= R^{-T} x
\end{align*}
\]

(6.3)

For the column array on right in Fig 6.2, if we input the s vector into the this column array, we can get

\[
\begin{align*}
  x_b &= x_a - u_1s_1 - u_2s_2 - u_3s_3 - u_4s_4 \\
  x_b &= x_a - s^T u \\
  x_b &= x_a - x^T R^{-1} u
\end{align*}
\]

(6.4)

Furthermore, if we input a matrix \(X\) instead of vector \(x\) and a vector \(x_b\) instead of a scale \(x_b\). We can get [1]

\[
\begin{align*}
  x_b &= x_a - X^T R^{-1} u
\end{align*}
\]

(6.5)

If input an identity matrix on \(X\) and zeroes vector on \(x_a\), we get

\[
\begin{align*}
  x_b &= -R^{-1} u
\end{align*}
\]

(6.6)

In fact, this is our weights output, but that detail will be discussed in next section. Therefore, for this 5 by 5 triangular array, we can simply get our weights flushed serially by inputting a 4 by 4 identity matrix to the first 4 channels and a zeroes vector to the last
channel. Meanwhile, the last one boundary node will not change the phase of the output value. From the equation (2.4), it will simply change the magnitude of the weights. This will not affect the weights. Fig 6.3 shows how the weights flushing works.

Fig. 6.3 A QR-D of 4 channels with weight flushing (each node have one clock cycle delay) [1]
### 6.2.2 Weights Computation

Based on the conclusion from previous section, the method for obtaining the weights will continue to be discussed. From the equation (1.8), we can get

\[
R = \left( R^T R \right)^{-1} c^*
\]

\[
w = R^{-1} \left( R^{-T} c^* \right).
\]

(6.7)

As equation (6.3), we know the triangular array can perform the back-substitution under frozen mode. But, to compute the weight, we need a double back-substitution. Therefore, in this way, a double substitution is performed.

Firstly, for the simple 5 input channel QR-D array in Fig. 6.3, we simply use its four input channel to do the QR-D. This means only utilize this decomposition processor for 4 antennas array. After input 4 channel matrix data, an upper triangle matrix stored in the left first 4 column array as fig. 6.3 is obtained. Secondly, enter the frozen mode. Input a vector \([c_1 \ c_2 \ c_3 \ c_4 -I]\) on all 5 input ports. In this case the left part triangular array will work as a matrix multiplication. It will generate a column vector \(R^Tc\) and output this vector to the right column array through ports “s”. Meanwhile “-I” is inputted from the top to the right column array on port “x”. It will obey the equation as (6.8).

\[
u_i(n) = c_i u_i(n-1) + s_i^* x_i(n)
\]

\[
x_{i+1}(n) = -s_i u_i(n-1) + c_i x_i(n).
\]

(6.8)

It stores the vector “-(\(R^Tc\))” to the vector \(u\) as Fig. 6.4 (B). Thirdly, input an identity matrix and zero vector to the 5 channel as Fig 6.3 “weigh flushing”. As the equation in
(6.5) (6.6), the weight can be calculated as equation (6.7).

\[ X = I, \ u = -(R^T \cdot c)^*, \ x_a = 0 \ . \]  \hspace{1cm} (6.9)

Bringing the above variables to equation (6.5), obtain the exact same express as equation (6.7) as in Fig 6.4 (C). Then the \( w \) vector will go through the last boundary node where magnitude will be changed but phase of weights will keep the same. Finally, we extract the weight from the QR-D triangular array. For the linear systolic array, everything is the same.

---

**Fig. 6.4** Weights computing processing
REFERENCES


23. Michael Gay and Isabel Phillips, QinetiQ Ltd. “Real-Time Adaptive Beamforming:

85
FPGA implementation Using QR Decomposition” JED, The JOURNAL of ELECTRONIC DEFENSE • SEPTEMBER'05


