Accelerator for Flexible QR Decomposition and Back Substitution

by

Srimayee Kanagala

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved November 2020 by the
Graduate Supervisory Committee:

Chaitali Chakrabarti, Chair
Daniel Bliss
Yu (Kevin) Cao

ARIZONA STATE UNIVERSITY

December 2020

ABSTRACT

QR decomposition (QRD) of a matrix is one of the most common linear algebra operations used for the decomposition of a square/non-square matrix. It has a wide range of applications especially in Multiple Input-Multiple Output (MIMO) communication systems. Unfortunately it has high computation complexity – for matrix size of nxn, QRD has $O(n^3)$ complexity and back substitution, which is used to solve a system of linear equations, has O(n^2) complexity. Thus, as the matrix size increases, the hardware resource requirement for QRD and back substitution increases significantly.

This thesis presents the design and implementation of a flexible QRD and back substitution accelerator using a folded architecture. It can support matrix sizes of 4x4, 8x8, 12x12, 16x16, and 20x20 with low hardware resource requirement.

The proposed architecture is based on the systolic array implementation of the Givens algorithm for QRD. It is built with three different types of computation blocks which are connected in a 2-D array structure. These blocks are controlled by a scheduler which facilitates reusability of the blocks to perform computation for any input matrix size which is a multiple of 4. These blocks are designed using two basic programming elements which support both the forward and backward paths to compute matrix R in QRD and column-matrix X in back substitution computation.

The proposed architecture has been mapped to Xilinx Zynq Ultrascale+ FPGA (Field Programmable Gate Array), ZCU102. All inputs are complex with precision of 40 bits (38 fractional bits and 1 signed bit). The architecture can be clocked at 50 MHz. The synthesis results of the folded architecture for different matrix sizes are presented. The results show that the folded architecture can support QRD and back substitution for inputs of large sizes which otherwise cannot fit on an FPGA when implemented using a flat architecture. The memory sizes required for different matrix sizes are also presented.

ACKNOWLEDGMENTS

I would like to extend my sincere gratitude to my thesis advisor Dr. Chaitali Chakrabarti for her tremendous support,motivation and patience throughout my work. Her guidance during this research research as well as writing of this thesis during the pandemic around is greatly appreciated. I am thankful to my committee members, Dr. Daniel Bliss and Dr. Yu (Kevin) Cao, for providing me feedback and their time on my research. I would also like to acknowledge the DARPA-PFC program for partially supporting this work.

I would like to thank Shunyao Wu and Yang Li from Arizona State University for their contribution and generous help throughout my work. I am grateful to Anuraag Soorishetty for providing me guidance and critical feedback. Lastly, I would like to thank my family and friends for their unceasing encouragement, support and motivation.

TABLE OF CONTENTS

Page

CHAPTER Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Demand for wireless communication systems has increased rapidly in the recent years. With this demand, a need for higher data rates became a challenge in order to increase capacity of the communication system. Multiple Input Multiple Output (MIMO) is a technique that can significantly improve the throughput of a system by employing multiple antennas to support multiple data streams between the transmitter and receiver concurrently [26] [4]. MIMO technique is an integral part of wireless communication standards such as IEEE 802.11ac (WiFi), HSPA+ (3G), Long Term Evolution (LTE), etc. This is because such a system results in increase of system capacity and spectral efficiency. Therefore, MIMO systems require sophisticated algorithms in the receiver for signal processing and efficient implementation of such algorithms is crucial.

QR decomposition is one of the many base-band functions of the MIMO receiver. It decomposes the complex-valued channel matrix H into an orthogonal matrix Q and an upper triangular matrix R. The channel matrices in MIMO communication systems utilize QRD in many linear detection schemes [3] and also complex methods such as list sphere detection (LSD) algorithm, sphere decoder (SD), etc [24] to leverage low complexity and high performance. The size of the channel matrix depends on the number of transmitter and receiver antennas. Thus, as the number of antennas increase, the complexity of the QRD also increases.

QRD is used not only in MIMO systems but also in beam formation in smart antennas [26]. Beamforming requires multiple antennas working together to create constructive and destructive interferences among antennas which produce a specific

beam pattern. Beamforming can improve the link capacity remarkably providing wider bandwidth per user channel [26]. Conventional beamformers are limited by jamming signals at a main lobe and side lobes of the array response. This can be solved by adaptive beamformer such as MVDR (Minimum Variance Distortion Response) algorithm. It uses QRD to calculate inverse of an input matrix, and this facilitates simple solution for complex matrix inversion operation [1].

QRD involves factorizing a square or a non-square matrix A into a product of an orthogonal matrix Q and an upper triangular matrix R, given by $A = QR$. The complexity of QRD is $O(n^3)$. QRD is a powerful algorithm to stably compute the eigen values and the corresponding eigen vectors. It uses successive unitary transformations which makes it more stable than other common factorizations such as LU decomposition. It is a dominant method for eigen value computations because of the recent developments in parallelized versions of QRD [2].

Back Substitution, on the other hand, involves solving a system of linear equations by transforming them into row-echelon form. The transformation can be done using different methods such as LU decomposition, QR decomposition, Cholesky decomposition, etc. QRD is most commonly used algorithm to transform the linear equations of the form $AX = B$ to $UX = Y$ where U is an upper triangular matrix. The complexity of back substitution algorithm is $O(n^2)$.

QRD can be computed using various techniques such as Gram-Schmidt [23], Givens Rotations [11], Gentlemans Algorithm [9], Householder reflections [12], etc. Gram-Schmidt process is a method for ortho-normalizing a set of vectors. In this method, QR is derived by finding an orthogonal projection vector q_n for each column vector a_n of input matrix and then subtracting its projections onto the previous projections (q_{n-1}). The resulting vector is then divided by the length of that vector to produce a unit vector. In Givens rotation, orthogonal plane rotations are used

to eliminate the lower triangular matrix elements within a matrix i.e, $[a_{i,j}] = 0$ for $i > j$. The input matrix is pre-multipled by rotation matrices one element at a time and rotation parameters are calculated so that the sub-diagonal elements are zeroed. Gentlemans algorithm is based on Givens rotations where the algorithm is modified to eliminate square root operation. In Householder transformation, Householder reflections are are used to calculate QRD by reflecting the first column of a matrix onto a multiple of a standard basis vector, calculating the transformation matrix, multiplying it with the original matrix and then recursing down the minors of the product.

Givens Rotation is used in most of the previously published hardware implementations because of its numerical stability and accuracy. It is typically implemented using systolic array architectures consisting of two-dimensional array of programming elements. This approach takes advantage of the inherent parallelism of Givens rotation for matrix triangularization. In this thesis, we too focus on a systolic array implementation of Givens rotation based QRD.

1.1 Motivation

The goal of this work is to design an architecture for a scalable QRD and back substitution for resource constrained systems. This architecture aims to support any matrix size though here we focus on sizes 4x4, 8x8, 12x12, 16x16, and 20x20. The resource utilization is very high, that is, the programming elements are re-used as much as possible. This work makes use of the same programming elements in both forward and backward computations avoiding separate blocks for the backward flow. The architectural framework of choice is FPGA.

1.2 Prior Work

One of the initial implementations of QRD using systolic array was proposed by H.T.Kung using Givens rotations [10]. A systolic array is a regular array of programming elements or cells in which data flows in a particular direction. Systolic arrays can be two-dimensional (rectangular, triangular or hexagonal) and data flow between the cells can be at different speeds and different directions. Systolic array architectures are interesting as they can solve certain problems in linear time. Kung proposed the QRD architecture using two programming elements (PEs), boundary and internal cells. Boundary cell performs the rotations in Givens rotations and internal cell uses these rotation parameters to transform the matrix. This architecture is ideal for real-time matrix computations on VLSI hardware.

Givens rotation based on Logarithmic number system (LNS) has been implemented to avoid the multiplication and division operation. A square root free algorithm using recursive least squares on LNS has been presented in [7]. In these architectures, the division and multiplication algorithm are replaced by subtraction and addition. Givens rotation has also been implemented using CORDIC (coordinate rotation digital computer) algorithm [27]. CORDIC speeds up the computations as it uses simple shift-add operations to realize Givens rotation. Some of the instances of QRD implemented on FPGA are shown in [15], [11], [5]. FPGAs provide high performance and power efficiency for custom hardware designs. But for large matrices, resource limitation on FPGA can be a drawback.

Most of the work described above target at different challenges in hardware design. As some of the implementations improve accuracy, they require high resource utilization. Each algorithm used for QRD target different issues such as number of computations, numerical stability, support for parallel architectures, etc. This the-

sis facilitates a single design implementation to support flexible matrix sizes with minimum resource utilization.

1.3 Thesis Contributions

The main contribution of this thesis is the design of a flexible architecture for QR decomposition and back substitution targeted on an FPGA. The architecture is based on a systolic array implementation using Givens' Algorithm. The 2-dimensional systolic array consists of two fixed Programming Elements (PEs), namely, boundary cell and internal cell. These cells support both forward and backward paths utilizing the same architecture to compute the upper triangular matrix R in the forward path and column matrix X in the backward path.

The foldable architecture for QR decomposition + back substitution supports matrix sizes that are multiples of 4, though we can extend it to support any matrix size. The architecture consists of three basic computation blocks along with memory. We also describe a scheduler to define the sequence of computations through these computation blocks.

The flexible QR accelerator computes the upper triangular matrix R, and uses the matrix R to derive the column matrix X as in a linear equation solver for back substitution. To generate an upper triangular matrix R, a series of rotations needs to be computed. The PEs in the QR accelerator perform rotations on each element of the matrix A resulting in the annihilation of the lower triangular matrix. The boundary cells perform square root and reciprocal operation; the internal cells perform a basic multiplication and accumulation (MAC) operation.

At the end of the forward computation, the R values computed in the forward path are stored in each cell. These R values are used in the back substitution to compute X using the same boundary and internal PEs. A division operation is performed to

compute the X value. These X values are stored in a memory to pass as input to other internal cells belonging to same column which compute an intermediate value to pass to the boundary cell of the specific row which eventually computes the X value of that row.

The proposed folded architecture is built using three computation blocks, namely Type I, Type II and Type III blocks. The Type I block is the triangular block on the diagonal consisting of four boundary cells and six internal cells, Type II block is a 4x4 kernel of 16 internal cells and the Type III block is a column matrix of 4 internal cells. The architecture also consists of four memory units, namely, RAM I, RAM II, RAM III and RAM IV. RAM I stores the rotation parameters C and S from Type I block, RAM II stores the intermediate values between two Type II blocks, RAM III stores the matrix R elements computes at each stage in Type I and Type II blocks, and RAM IV stores the matrix X elements computed in back substitution. The QRD computation of any sized matrix can be mapped onto this folded architecture. The scheduler in the design is used to schedule the data flow in each of the computation blocks and the memory read/writes to support data flow between each of the blocks.

We illustrate the architecture using matrices of sizes of 4x4, 8x8, 12x12, 16x16 and 20x20. We evaluate the hardware performance using resource utilization for each matrix size compared to the flat architecture. We evaluate the algorithm performance by calculating the error in deciBels (dB) for the matrix R in QRD and matrix X in back-substitution. In the forward computation, an average error of -47.89dB is observed for an input matrix of size 20x20. In the back substitution, an average error of -36.88dB is observed for matrix size of 20x20.

The proposed architecture can be clocked at 50MHz. The resource utilization of this architecture shows that it uses 32% of LUT utilization, 3.82% of LUTRAMs, 9.39% of FF and 80.86% of DSPs in Xilinx ZCU102 FPGA. It is 8x times smaller

than the flat architecture implementation for a 20x20 matrix. In fact, ZCU102 cannot support flat implementation of matrices larger than 12x12. Using this architecture, the maximum matrix size supported to compute matrix R and back substitution depends on the memory size required for each matrix size.

1.4 Organization

The rest of the thesis is divided into the following chapters: Chapter 2 presents a brief introduction of the existing hardware implementations, QR decomposition algorithm, and the Givens method. Chapter 3 presents a detailed explanation of the flexible QRD architecture, the scheduler, and the memory architecture. In Chapter 4, the back substitution accelerator is presented along with the scheduler. Chapter 5 summarizes the results of the flexible and foldable architecture for different matrix sizes in terms of performance, area utilization and computation times. Chapter 6 concludes the thesis along with future work.

Chapter 2

BACKGROUND

In this chapter, we first give a brief introduction to the QRD and back substitution, the Givens algorithm for QRD, and its existing hardware implementations.

2.1 QR Decomposition and Back Substitution

QRD is a linear algebraic method to factorize any given matrix A into two matrices: 1) an orthogonal matrix called Q, and 2) an upper triangular matrix called R such that $A = QR$ as shown in equation 2.1.

$$A = \begin{bmatrix} q_1 & q_2 & . & . & . & q_n \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} & . & . & . & R_{1n} \\ 0 & R_{22} & . & . & . & R_{2n} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ 0 & 0 & . & . & . & R_{nn} \end{bmatrix} \tag{2.1}$$

One of the applications of QRD is solving the linear system of equations. Consider the following equation $AX = B$, where A is an n x n matrix. The matrices A and B are provided as inputs and the objective is to find the column matrix X. Using QRD, matrix A can be factorized into QR as shown in equation 2.2.

$$\begin{gathered} \text{Let } AX = B, \text{ then if } A = QR \\ (QR)X = B. \text{ Thus }, RX = Q^{-1}B. \\ \text{If } B' = Q^{-1}B, \text{ then } RX = B' \end{gathered} \tag{2.2}$$

The resultant equation RX=B' can be represented as

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & . & . & R_{1n} \\ 0 & R_{22} & R_{23} & . & . & R_{2n} \\ 0 & 0 & R_{33} & . & . & R_{3n} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ 0 & 0 & . & . & . & R_{nn} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ . \\ . \\ X_n \end{bmatrix} = \begin{bmatrix} B'_1 \\ B'_2 \\ B'_3 \\ . \\ . \\ B'_n \end{bmatrix}$$

The X_n value can be computed using the reciprocal of the $R_{n,n}$ value. Solving the resultant matrix in a bottom-up approach, results in the equations in 2.3.

$$\begin{aligned} R_{n,n}X_n &= B'_n \\ R_{n-1,n-1}X_{n-1} + R_{n-1,n}X_n &= B'_{n-1} \end{aligned} \tag{2.3}$$

Thus, X_n can be used to compute X_{n-1}, and X_{n-1} can be used to compute X_{n-2} and so on. In this way, all the X values in matrix X can be computed using the matrix R. This is known as the back substitution method to solve for X. Since QRD has a computation complexity of O(n^3) and back substitution has a computation complexity of O(n^2), the resource utilization increases significantly with increase in input matrix size [17].

2.2 Givens Algorithm

Givens method performs QRD by Givens rotation [11]. Here, all the elements of a row are rotated in a way that annihilates the lower triangular matrix. In this thesis, we use the systolic array implementation of Givens rotation-based algorithm for designing the scalable QRD and back substitution algorithm.

In the Givens algorithm, the elements in the lower triangular matrix of the input are annihilated at each time-step until an upper triangular matrix is formed. The

rotation performed at each step annihilates element a_{ij} in matrix A, where i represents the row and j represents the column, starting from the last row of the first column. Specifically, Givens matrix $G(i, j, c, s)^T$ is used to convert the input matrix A into an upper triangular matrix. It rotates the i^{th} and j^{th} elements of matrix A by an angle θ such that $cos\theta = C$ and $sin\theta = S$. C and S can be obtained by equations 2.4 and 2.5.

$$C = a_{i,k}/r_{i,j} \tag{2.4}$$

$$S = a_{i,j}/r_{i,j} \tag{2.5}$$

To annihilate element $a_{i,j}$, we multiply elements $a_{i,k}$ and $a_{i,j}$ with the Givens matrix.

$$\begin{bmatrix} C & -S \\ S & C \end{bmatrix} \begin{bmatrix} a_{i,k} \\ a_{i,j} \end{bmatrix} = \begin{bmatrix} r_{i,j} \\ 0 \end{bmatrix}$$

Pre-multiplying the matrix A with G affects only the rows i and j of the matrix A and the element $a_{i,j}$ is annihilated. The element $r_{i,j}$ is computed using

$$r_{i,j} = \sqrt{a_{i,k}^2 + a_{i,j}^2} \tag{2.6}$$

If more than one entry in the matrix A need to be annihilated, then a sequence of Givens rotations needs to be performed. Once the required elements are annihilated, the Q and R matrices are obtained. For example, for a 4x4 matrix, six elements of matrix A need to be annihilated, hence six Givens rotations need to be performed as shown in equation 2.2

$$\begin{bmatrix} x & x & x & x \\ x & x & x & x \\ x & x & x & x \\ x & x & x & x \end{bmatrix} \xrightarrow{G_{4,1}} \begin{bmatrix} x & x & x & x \\ x & x & x & x \\ x & x & x & x \\ 0 & x & x & x \end{bmatrix} \xrightarrow{G_{3,1}} \begin{bmatrix} x & x & x & x \\ x & x & x & x \\ 0 & x & x & x \\ 0 & x & x & x \end{bmatrix} \xrightarrow{G_{2,1}} \begin{bmatrix} x & x & x & x \\ 0 & x & x & x \\ 0 & x & x & x \\ 0 & x & x & x \end{bmatrix} \xrightarrow{G_{4,2}}$$

$$\begin{bmatrix} x & x & x & x \\ 0 & x & x & x \\ 0 & x & x & x \\ 0 & 0 & x & x \end{bmatrix} \xrightarrow{G_{3,2}} \begin{bmatrix} x & x & x & x \\ 0 & x & x & x \\ 0 & 0 & x & x \\ 0 & 0 & x & x \end{bmatrix} \xrightarrow{G_{4,3}} \begin{bmatrix} x & x & x & x \\ 0 & x & x & x \\ 0 & 0 & x & x \\ 0 & 0 & 0 & x \end{bmatrix} \quad (2.7)$$

The annihilation starts from first column and proceeds to the last column. $G_{4,1}$ is the first rotation to the input matrix to annihilate the element at 4^{th} row, 1^{st} column. Then $G_{3,1}$ rotation is performed to annihilate the $a_{3,1}$ element and $G_{2,1}$ rotation is performed to annihilate the $a_{2,1}$ element. Thus, annihilation proceeds in reverse order within a column. Consequently, $G_{4,2}$, $G_{3,2}$ are performed to annihilate the elements in the 2^{nd} column. Finally, $G_{4,3}$ is performed to annihilate the last element to generate the upper triangular matrix R.

In the Givens rotation, first column of a matrix is used to find the angle of rotation and this is used for the rest of the columns. This leads to a regular pattern and hence it can be mapped to a systolic architecture [22].

The upper triangular matrix computed in this forward path is used to calculate the X values in a linear equation solver in back substitution. The rotation parameters used in the QRD are used to compute the matrix B' which represents the $B * Q^T$ matrix. This matrix B' is used in back substitution in a bottom up approach to compute the X values.

2.3 Existing Hardware Architectures

There are several hardware implementations of QRD. The earliest implementation was proposed by H. T. Kung and W. M. Gentleman in 1981 to solve QRD using systolic arrays [10]. Several algorithms of QRD have been explored to optimize the square root operation in [10], as it is compute intensive. In [9], Gentleman proposed an algorithm to solve the QRD using systolic array without square root. [7] and [13] proposed a parameter based square-root free algorithms where different values of parameters lead to different multiplication and division operations.

These algorithms have been implemented on different hardware platforms such as ASICs (Application Specific Integrated Circuits), FPGAs, GPUs (Graphical Processing Units), etc. The ASIC implemented in [8] proposes a hierarchical pipelining and folding method. Specifically, they use a folded transformation to map a look ahead transformed pipelined array to a small square array. In [18], a QRD hardware architecture based on a parallel tiled QRD algorithm is presented. This implementation is illustrated for a fixed size 8x8 real matrix and targeted for parallel architectures where a matrix can be divided into smaller tiles such as a tile size of 2x2. A QRD implementation based on Gram-Schmidt algorithm was proposed in [25]. As Gram-Schmidt is computationally expensive, this method performs division operation with assistance of square root using CORDIC. This is targeted to an ASIC platform and supports a matrix of size 4x4. The ASIC implementations are fast and efficient but they lack reconfigurability as they target specific application.

GPUs, on the other hand, are popular hardware platforms as they make extensive use of parallelism. In [21], data and loop-level parallelism based Givens QRD kernel is presented. Efficient memory access is achieved using intelligent loading and storing techniques. In [20], a generalized Givens rotation is presented as an improvement over

classical Givens rotation using a GPGPU (General Purpose GPU) with a massively parallel reconfigurable MPSoC (Multi Processor System on a Chip) architecture.

FPGAs are also reconfigurable platforms which are used as both dedicated hardware or as an accelerator with a host architecture. They have the programmability of software and performance capability approaching custom hardware implementations. In [6], a CORDIC-based QRD using Givens algorithm is implemented on an FPGA. This supports a real input matrix of size 4x4 and achieves high hardware utilization. The squared Givens rotation implemented in [16] reduces the latency by 50% compared to Givens rotation. A folded systolic architecture is implemented in this design and targeted on Xilinx Virtex 4 FPGA running at 115 MHz. While results for a 4x4 input matrix are presented in [16], the method is extendable to other matrix sizes with changes in the control unit.

In [14], a automatic generation tool of different decomposition methods such as QRD, LUD and Cholesky Decomposition targeted on a Xilinx Virtex 4 FPGA is presented. The programmable tool supports different decompositions for input matrix size of 4x4. In [19], a vectorized algorithm is introduced to compute QRD for floating point input. This architecture computes large matrix sizes of 192x200 in 1 ms but the hardware resource requirement is extremely high. It requires 274 18x18 multipliers, 33K registers, 29K ALUTs and 5.7 MB of memory, which is not sustainable.

This thesis also presents an architecture targeted on an FPGA to utilize the reconfigurable aspects, thus reducing the design time. Though most of the existing implementation have focused on a 4x4 configuration with the assumption that they can be scaled up for larger matrix sizes, this work explicitly demonstrates how a single architecture can support different matrix sizes. CORDIC has been used in some of the hardware implementations to improve the latency in square-root operations of Givens rotation. Since CORDIC uses high resources, we have not studied it here. Instead, we

focused on a three-stage pipelined architecture to decrease the clock period, thereby increasing the design frequency.

Chapter 3

SCALABLE QRD ACCELERATOR

In the previous chapter, we analyzed different architectures used in QR decomposition accelerators. In this chapter, we give an overview of QRD for a matrix of size nxn in Section 3.1 followed by description of the programming elements in Section 3.2. Section 3.3 gives a detailed description of QRD for a 4x4 matrix and followed by description of Scalable architecture in Section 3.4. Section 3.5 presents the details of memory size required for the design.

3.1 Overview of QRD for an nxn Matrix

In this section, a detailed explanation of the triangular systolic array for QRD is presented. This implementation consists of two types of programming elements: boundary cell and internal cell [10]. The boundary cells are placed along the diagonal of the array and the rest of the elements are implemented by the internal cells. The rotation parameters C and S mentioned in Section 2.2 are computed by the boundary cell. These are broadcast to the internal cells in the same row at each time step to compute the intermediate values. These intermediate values serve as inputs to the boundary cells in the next row. In the end, R values stored in each cell are the final R values of the upper triangular matrix.

Figure 3.1 describes the two-dimensional systolic array architecture to compute matrix R. The round cells represent the boundary cells and the square cells represent the internal cells. The boundary and internal cells are connected as shown in the figure. Figure 3.1 also describes the dataflow through the architecture. The input to the first row of elements in the triangular systolic arrays are passed in a diagonal

manner. At each time step, C and S values are calculated with elements of the input matrix of the column. These C and S values are passed to the next internal cell as inputs. The internal cell takes the C and S values of the boundary cell and input matrix of the specific column to compute the R value and U value. U value is the input to the next cell in the next row. The data flow here is in vertical direction. Note that the C and S parameters generated in the boundary cell do not get modified in the internal cells but are passed on to the next internal cells at each time step.We implement this by broadcasting the C and S values to all the internal cells at the same time.

There is another column of internal cells in the far right which take the B column vector (AX=B) as input. The C and S values generated in the boundary cell are used to compute the B' values which are used in back substitution.

3.2 Programming Elements (PEs)

The two basic programming elements used in this design are the boundary and internal cells. The boundary cells occupy the diagonal elements of the matrix R and compute the rotational parameters. The internal cells compute the intermediate values using MAC operations.

3.2.1 Boundary Cell

The boundary cell takes a complex number A as input and computes C, a real parameter and S, a complex parameter. It also computes the R value which is stored in the boundary cell for the next computation. Figure 3.2 describes the ports in boundary cell.

Algorithm 1 describes the equations used in the boundary cell computations:

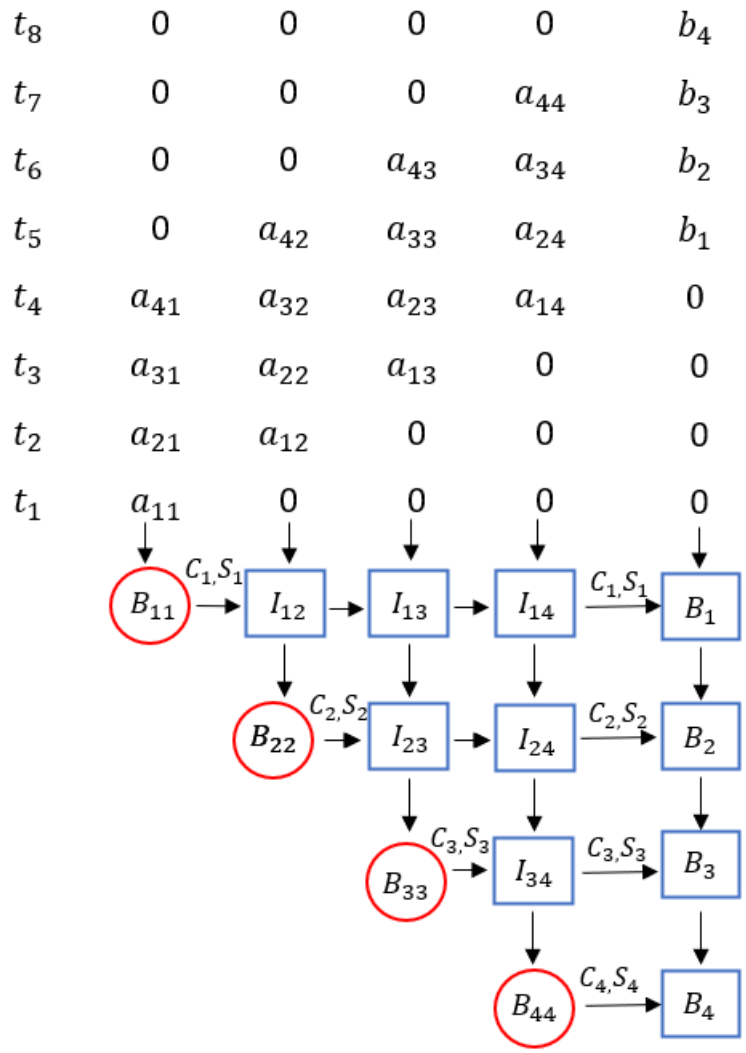

Figure 3.1: Data Flow in QR Using Systolic Array Architecture

Algorithm 1 Boundary Cell Operations

if $U_{in} \leftarrow 0$ **then**

$C \leftarrow 1$; $S \leftarrow 0$; $R \leftarrow R'$

else

$R' \leftarrow \sqrt{R^2 + |U_{in}|^2}$; $C \leftarrow R/R'$

$S \leftarrow U_{in}/R'$

$R \leftarrow R'$

end if

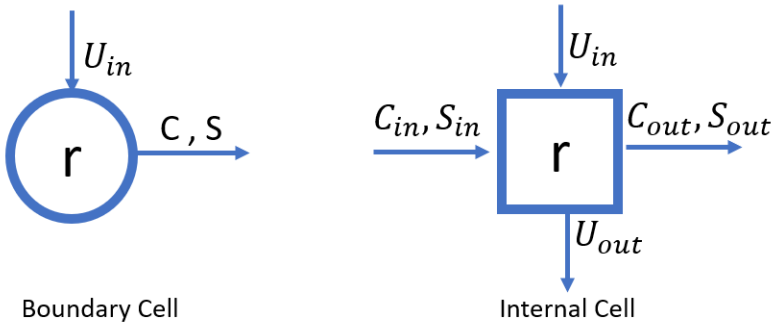

Figure 3.2: Boundary and Internal Cell Flow Diagram

Thus the boundary cells perform a reciprocal square root operation to generate the C and S parameters. This is a compute-intensive operation which requires a very long clock period if it has to be completed in one-time step. The latency of boundary cell is significantly larger than the internal cell which only have MAC operations.

To overcome this drawback, the boundary cell is redesigned to support a 3-stage pipelined architecture. Each boundary cell now takes 3 clock cycles to generate the C and S parameters. The schedule for the forward path is implemented with the new pipelined architecture.

In the Simulink design, the pipeline stages are inserted before the square root operation and before the reciprocal operation as shown in Figure 3.3. This three-stage pipeline is used to mimic the hardware pipeline stages of the boundary cell. When an RTL is generated for the boundary cell, the pipeline stages are incorporated in the square root and reciprocal block which reduce the critical path of the boundary cell. Since in Simulink, a square root or reciprocal block cannot be broken down into a three-stage pipeline, the boundary cell is designed in this way to simulate the hardware behavior. Thus both in the hardware and in the Simulink simulation, the outputs of the boundary cell are generated after three clock cycles and hence the same schedule is followed for both implementations. Furthermore, to decrease the computations

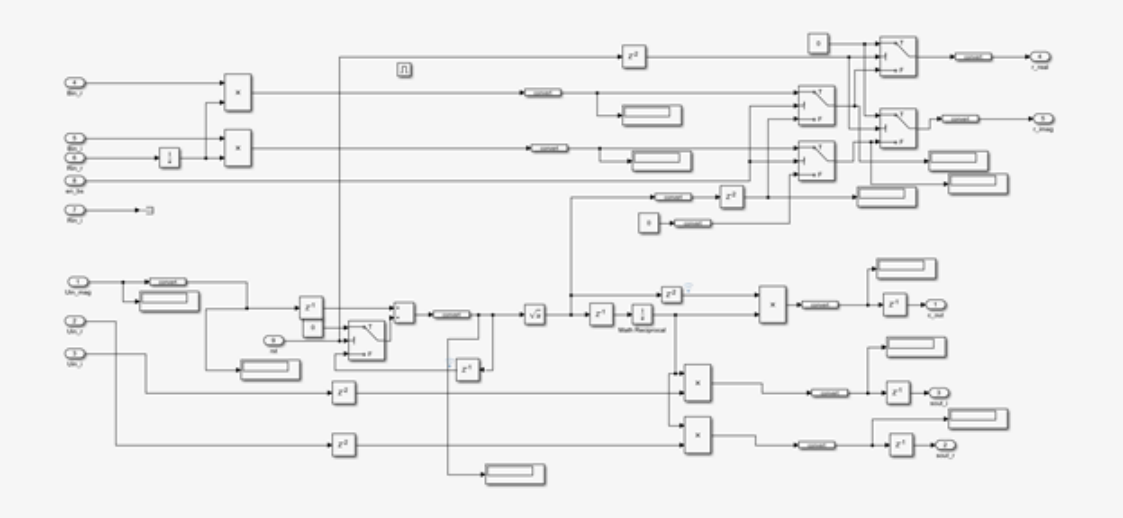

Figure 3.3: Boundary Cell Designed In Simulink

of the boundary cell, each boundary cell is provided with the magnitude as input which is the square of the real and complex value of the current input A. For the first boundary cell which calculates $R_{1,1}$, the magnitude is sent as input from the testbench. For the other boundary cells, the magnitude is calculated in the internal cell above the boundary cell and sent using vertical connections.

3.2.2 Internal Cell

The internal cells occupy the non-diagonal elements of the upper triangular matrix in the triangular systolic array. They have a significantly lower complexity than the boundary cell. Figure 3.2 describes the internal cell ports. An internal cell computes two complex multiplications, two scalar multiplications, and one addition and subtraction. Algorithm 2 describes the computations in an internal cell:

Algorithm 2 Internal Cell Operations

$U_{out} \leftarrow CU_{in} - SR$

$R' \leftarrow S^{*}U_{in} + CR$

The internal cell uses the C and S inputs generated from the boundary cell and

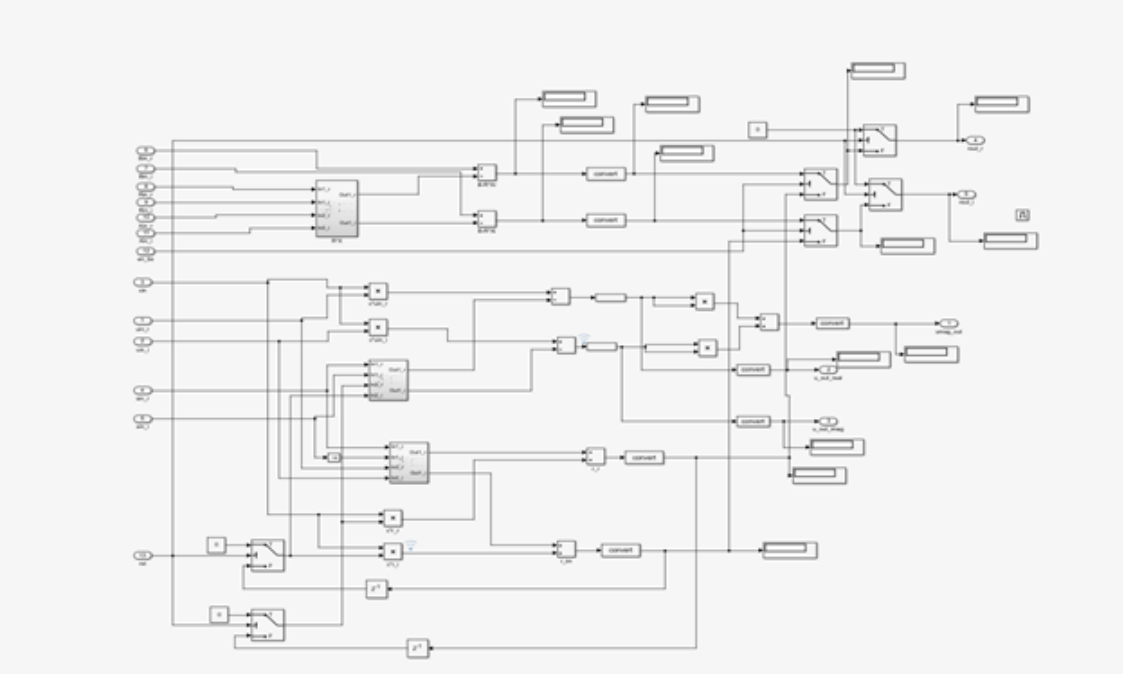

Figure 3.4: Internal Cell Designed in Simulink

U_{in} passed as input to the internal cell to generate U_{out}, which is passed as input to next boundary or internal cell present in a vertical data flow in the systolic array. The R value is computed using U_{in}, C and complex conjugate of S(S^*). The R value generated is stored in a local memory to be used in the next computation. At the end, the R value stored in the cell represents the final R value of the upper triangular matrix. Figure 3.4 describes the block diagram of the internal cell designed in Simulink:

The internal cells are used in the the last column to compute the B' values from the input column vector B. The B' values generated from these internal cells are stored and used in back substitution as inputs. In the forward path, the internal cells in the upper triangular array and the internal cells in the column matrix have same functionality. In the back substitution, internal cells in the last column become inactive since they do not compute any intermediate values.

In Figure 3.5, the round cells represent the 3-stage pipelined boundary cell and the square cells are internal cells.To synchronize the data flow between boundary and internal cells, a delay of two clocks is used in the vertical data flow i.e between two consecutive rows. Since C and S parameters from the boundary cell are broadcast in the same time steps to the internal cells of each row, the inputs to these columns are passed in the same time step.

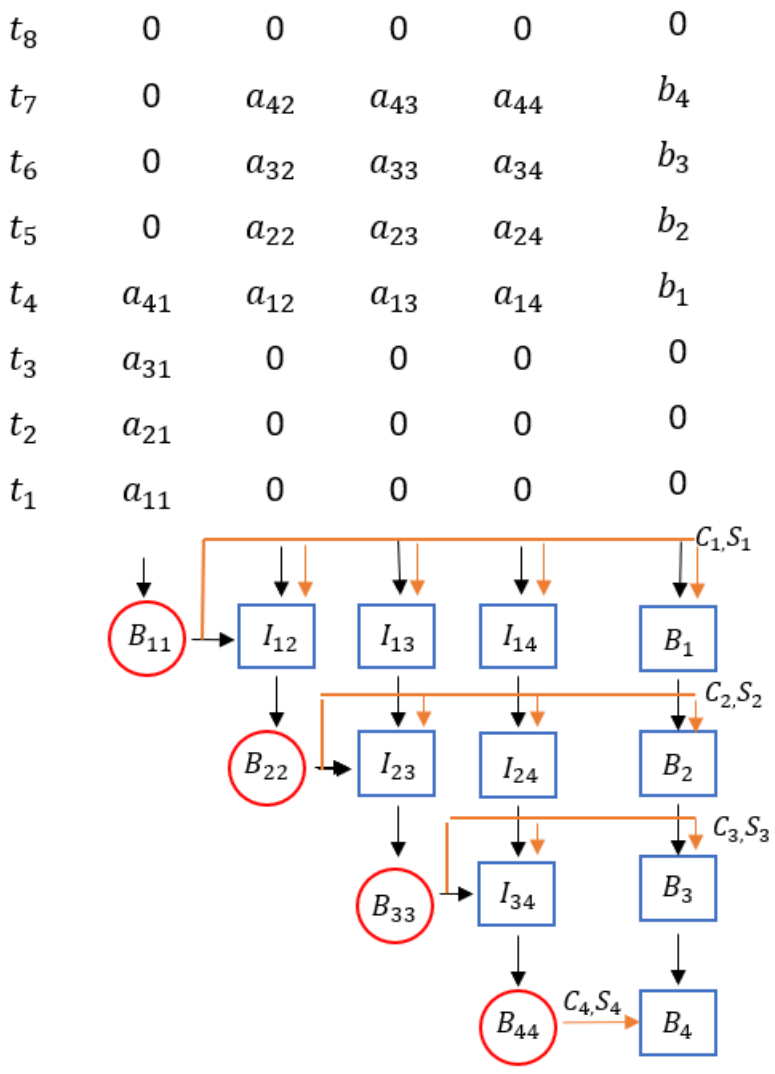

Figure 3.5: Data Flow in QR Using Pipelined Systolic Array Architecture

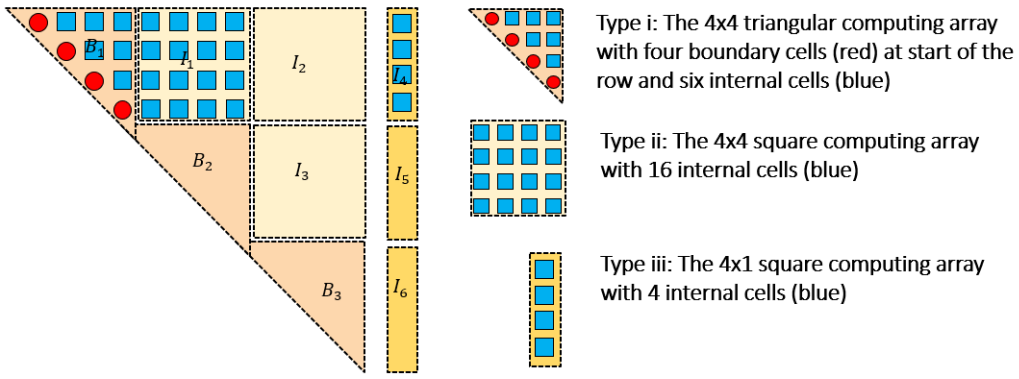

Figure 3.6: 12x12 Systolic Array Architecture

In a 4x4 triangular array, when the first boundary cell (B_{11}) receives the first input a_{11} at the first time step t_1, it computes the C and S in a three stage pipeline. Therefore, the first C and S values are generated after 3 clock cycles. The R value generated is stored in the boundary cell to be used for the next computation.

At the end of the third clock cycle t_3, C and S values are propagated to the internal cells I_{12}, I_{13}, I_{14}, in the fourth time step. The other inputs to the internal cell, (a_{12},a_{13},a_{14}), are also passed at the fourth time step. The internal cells generate the inputs to the next set of PEs in the next row. The boundary cell B_{22} receives the input at fourth time step, thus generating the rotation parameters C and S at the end of 7^{th} time step.

The inputs to the internal cells I_{23} and I_{24} are delayed for 3 clock cycles to wait for next set of C and S values. Once the C and S values are propagated from the boundary cell B_{22}, I_{23} and I_{24} calculate the next set of inputs for row 3. The boundary cell B_{33} generates the outputs in the same way and internal cell I_{34} generates inputs for the boundary cell B_{44}. At the end of 4^{th} input for the first boundary cell B_{11}, the R value stored in the PE is the final R value R_{11}. In the similar way, at the end of all computations, each PE stores the R values. The R values thus held in each PE are the values of the upper triangular matrix R.

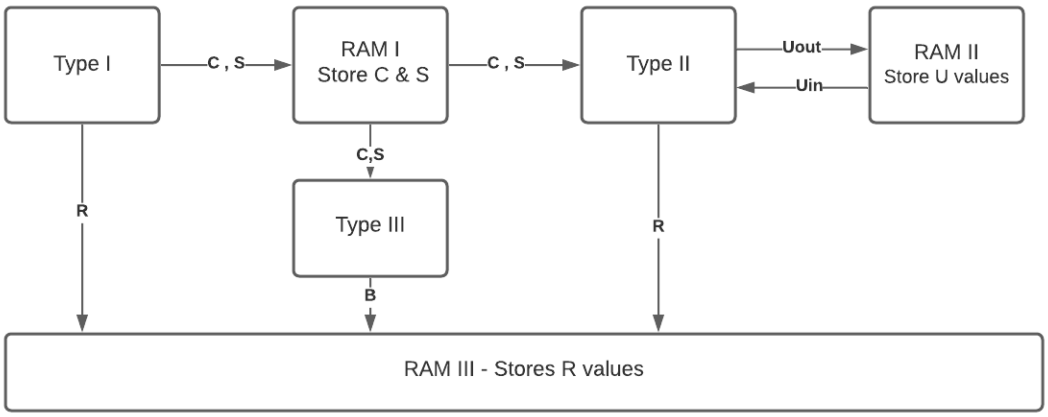

Figure 3.7: Block Diagram of the Scalable QR Accelerator

3.4 Folded Architecture Overview

In this section, we describe the proposed folded architecture and the schedule pattern used in the folded architecture. The block diagram of the proposed architecture is shown in Figure 3.7. There are three different types of computation blocks: a) Type I block: the triangular type, which consists of four boundary cells and six internal cells, b) Type II block: rectangular part which consists of sixteen internal cells, and c) Type III block: the column matrix part which consists of a column of four internal cells. These three blocks form the basic blocks of the folded architecture. The three compute blocks are connected to three memory blocks as shown in the Figure 3.7. Thus, a 12x12 matrix can be formed with three Type I blocks, three Type II blocks and three Type III blocks. The Type I block writes the C and S values to RAM I, the Type II block reads the C and S values from RAM I. RAM II reads and writes the U_{in} values to/from the Type II block. The RAM I is connected to Type III block as well to read the C and S values. RAM III is connected to all the three blocks as it stores the R values computed in each cell.

3.4.1 Schedule

The schedule used in the design follows a pattern which makes it flexible to use for any matrix size. The number of cycles for which a programming element is active depends on the input matrix size. Based on the active time of the blocks, the schedule for each of the RAM read and write also differ. The active time of the first boundary cell of a Type I cell is equal to n+3 where n is the matrix size. For a 20x20 matrix, the boundary cell is active for 23 cycles while for a 12x12 matrix, it is active for 15 cycles.

The minimum time required for a Type II block to wait to start a computation after the boundary cell of Type I block computes C and S values is 15 cycles. This time is required to synchronize the internal and boundary cell computations which take different number of cycles for their computations. This includes the read and write time of the C and S value to the memory RAM I. To elaborate, C and S values generated in each row of a Type I block have a time difference of 5 cycles whereas the time difference between two rows in a Type II block is 2 clock cycles. To maintain synchronization between these two types of blocks, the minimum time for a Type II block to start computation after a Type I block generates first set of outputs is 15 cycles.

A Type I block requires outputs from the last row of a Type II block to compute R values for the rows which are greater than 4. The minimum clock cycles required for a Type I block to wait for the inputs to be available after a Type II block is active is 12 clock cycles. This is because, the first output generated from the fourth row of a Type II block is 12 clock cycles after the Type II block is active.

In the schedule, the Type II blocks are active in a horizontal pattern i.e. all the columns of a set of 4 rows are finished first before starting computations for the next

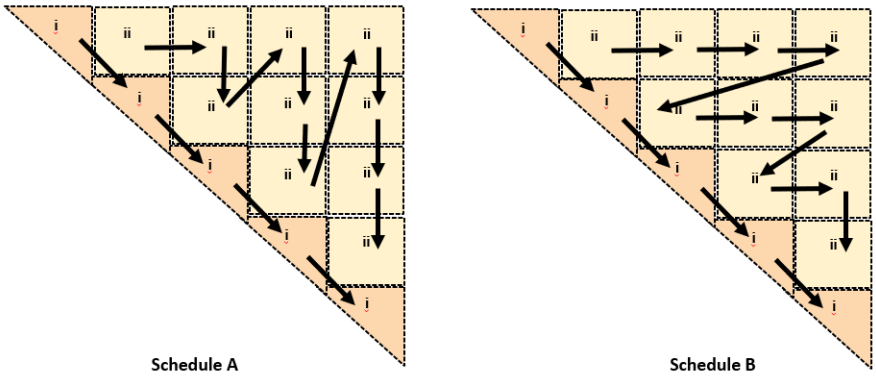

Figure 3.8: Two Possible Schedules for Computing QRD on a 20x20 matrix

set of 4 rows. An alternative pattern which can be used is a zigzag pattern where the Type II blocks are active in a vertical pattern. Figure 3.8 represents the schedules of the two patterns. Both these patterns have been analyzed in the thesis and it has been found that the schedule B with horizontal pattern is more optimized than the schedule B with vertical pattern. It takes less clock cycles and less memory since the wait time for the last Type I computation is less. Adding to this, this pattern takes less resources as the C and S values of each set of rows can be overridden in the consecutive operations. In the vertical pattern, all the C ans S values need to be saved in the RAM for all the computation time as the last column uses all the C and S values generated from the Type I blocks.

The Type III block follows the same schedule as the first Type II block of a row. The C and S values read from RAM I are passed as inputs to both the blocks. The only difference is that the Type III block is a column matrix and the matrix B' matrix values are generated earlier than the R values in Type II block.

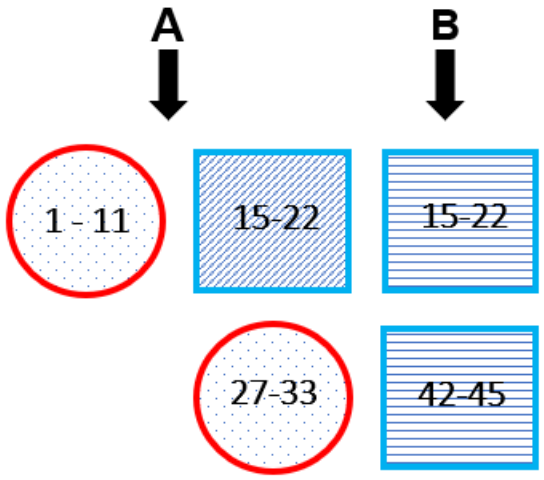

Figure 3.9: Schedule for Input Size 8x8

3.5 Schedules for Different Sized Matrices

3.5.1 Schedule for 8x8 Matrix

The schedule for a 8x8 matrix is explained in this section. 8x8 QR computation requires two Type I blocks, one Type II block and two Type III blocks. Figure 3.9 represents the active time of each block.The computation starts with passing the first four columns of matrix A as inputs at time 0 to the Type I block. The first boundary cell in the Type I block takes 11 clock cycles to compute the R_{11} value. The 8 C and S values from the boundary cell are passed to the next internal cell in the row 1 as well as to the RAM I. The RAM I stores the C and S values and use them in computing R values in Type II cell.

The first C and S value is available to the Type II block at 15^{th} clock cycle. At 15^{th} clock cycle, the last four columns of 8x8 matrix are passed as inputs to the Type II block. This block computes the $R_{1,5}$ to $R_{4,8}$ values. The $R_{1,5}$ value is computed at 22^{nd} clock cycle. Once the $R_{4,5}$ computation is started in the Type II block, the Type I block can be reused again to compute the $R_{5,5}$ value. When the Type II block

is active, at the same time, Type III block is also active to compute the B' values.To compute the $R_{5,5}$-$R_{8,8}$ values, the Type I block is active from 27^{th} to 33^{rd} clock cycle. As the Type I computations are completed and C and S values are calculated and stored in the RAM I , the Type III block is reused again to compute the last 4 values on B'. This block is active from 42^{nd} to 45^{th} clock cycle.

3.5.2 Schedule for a 12x12 Matrix

Next we describe the implementation of a 12x12 matrix on the folded architecture. The 12x12 array can be divided into blocks as show in Figure 3.6. For the ease of understanding, the Type I blocks are called as B_1,B_2,B_3 and Type II blocks are referred to as, I_1,I_2 and I_3 as shown in Figure 3.6. Type III blocks are called I_4,I_5 and I_6. To begin with, the first four columns of the 12x12 input matrix A are passed as inputs to the Type I block B_1 i,e., to the columns A_1 to A_4.

Since each boundary cell takes 3 clock cycles to compute the first C and S values, the first boundary cell B_{11} takes 15 clock cycles to complete all the computations and compute the final R_{11} value. The C and S values generated at each time step are stored in a memory, RAM I at address 1-12. These values are used to calculate the matrix R values for columns 5-8 and 9-12 in the Type II blocks I_1 and I_2.

As the first C and S values are generated from the Type I block and written to the memory, they are read from the memory and passed to the first internal cell of the Type II block I_1 to compute the matrix R for columns 5-8. At the same time, the inputs A_5 to A_8 are passed as inputs to the Type II block I_1. Since each internal block takes one-time step to calculate the U values, as described in section 3.3.2, the first internal cell I_{15} takes 12 clock cycles to compute the $R_{1,5}$ value. Parallelly, the C and S values are passed as inputs to the Type III block I_4 as well to calculate the column matrix B'. Figure 3.10 represents the schedule for 12x12 matrix.

When the $R_{1,5}$ value is computed, the Type II block I_2 can now be used to compute the matrix R for columns 9-12. Parallelly, once the last row of the Type II block I_1 generates the U_{out} values, the Type I block is active again to compute the matrix R for rows 5-8. The C and S values generated at each time step are stored in the RAM I at address space 21-28. As the Type II block I_2 finishes the computations of $R_{1,9}$-$R_{1,12}$ the same block as (I_3) is used now to compute the R values for $R_{5,9}$-$R_{8,12}$. The new C and S values generated in the second Type I computations (B_2) are used in this computation. For computing the $R_{5,9}$-$R_{5,12}$ values, U_{out} values computed in the I_2 are required as inputs. Hence these values are stored in memory unit RAM II.

When the new C and S values are read from RAM I and the U_{out} values are read from RAM II, the Type II block (I_3) computations begin. Similarly, Type III block (I_5) is active at the same time to compute the B' values for the row 5-8. Once the U_{out} values are generated from the last row of the Type II block, i,e., $I_{8,9}$-$I_{8,12}$, the Type I block (B_3) is active again to compute $R_{9,9}$-$R_{12,12}$. The C and S values generated are passed to the Type III block (I_6) to compute the B' values for the rows 9-12. Once the B'_{12} value is computed, the forward computations are marked finished in the architecture. The final R values from each Type I and Type II blocks are stored in another memory unit RAM III for back substitution.

3.5.3 Schedule for a 16x16 Matrix

In this section, schedule for 16x16 matrix is explained. Figure 3.11 represents the active time of Type I, Type II and Type III blocks. The Type I block is active for 19 clock cycles computing the $R_{1,1}$ value at the 19^{th} clock cycle using the first column of input matrix A. C and S values generated at each time step in the boundary cell are stored in RAM I. At clock cycle 14, C and S values are read from RAM I and passed to Type II block. At 15^{th} clock cycle, the columns 5 to 8 of input matrix A

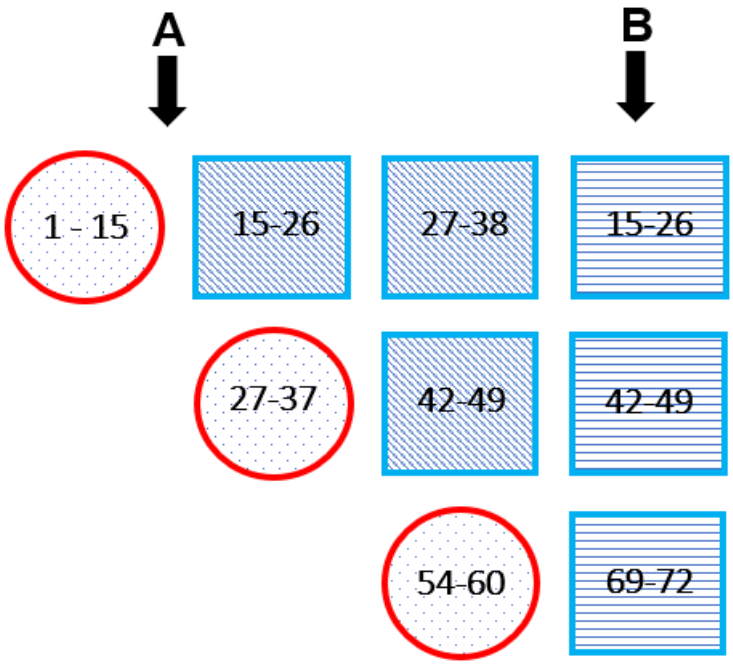

Figure 3.10: Schedule for Input Size 12x12

are passed as inputs to the Type II block. At clock cycle 30, Type II block finishes the computation of $R_{1,5}$ to $R_{4,8}$. The Type II block then starts the computation of $R_{1,9}$ to $R_{4,12}$, reading the C and S values from RAM I. Consequently, Type II block computes the $R_{1,13}$ to $R_{4,16}$ values in clock cycles 47-62.

During the computation of $R_{4,5}$ to $R_{4,8}$ values in Type II block, as the U_{out} values are computed at the internal cells, they are passed to the Type I block to compute the $R_{5,5}$ to $R_{8,8}$ values. The Type I block is active from 27^{th} to 41^{th} clock cycle. The C and S values computed in this time period are stored in RAM I. As the Type II block finishes the computations of $R_{1,13}$ to $R_{4,16}$, it reads the C and S values from RAM I to compute the values $R_{5,9}$ to $R_{8,12}$ from 63^{rd} to 74^{th} clock cycle. From clock cycle 75-86, $R_{5,13}$ to $R_{8,16}$ are computed in the Type II block.

From clock cycle 75-85, Type I block computes the R values $R_{9,9}$ to $R_{12,12}$ values. After the Type II block computations of previous row are finished, the Type II block

now computes the R values $R_{9,13}$ to $R_{12,16}$. It reads the inputs from RAM II which stores the U_{out} values from Type II block of previous row. The final set of R values are computed in the Type I block in clock period 101-107.

As the first Type II block of each row computes the R values, the Type III block computes the B' values parallely. Hence the B' values are computed in the clock periods 15-30, 63-74, and 89-96 for B'_1 to B'_4, B'_5 to B'_8 and B'_9 to B'_12. B'_13 to B'_16 are computed in the clock period 116-119 after the final computations of Type I block. This finishes all the computations in the forward path.

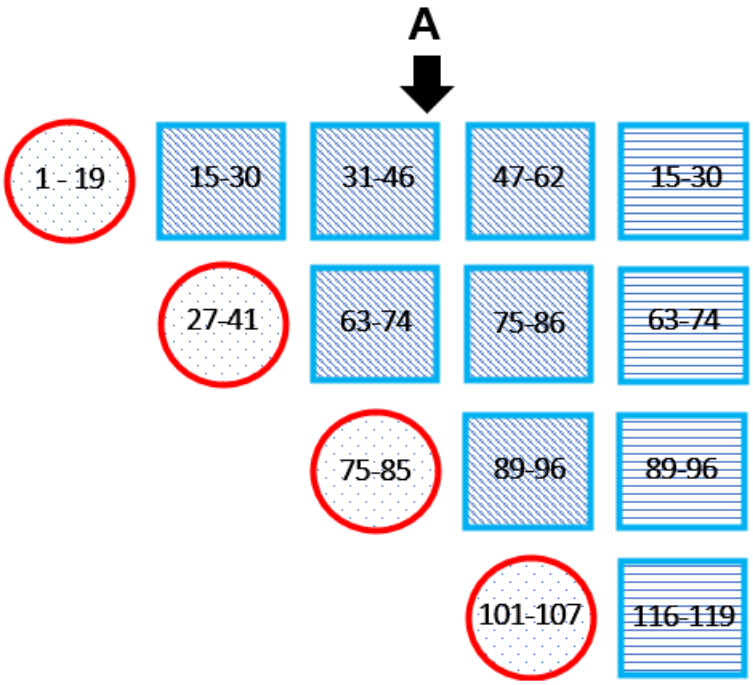

Figure 3.11: Schedule for Input Size 16x16

3.5.4 Schedule for a 20x20 Matrix

The schedule for a 20x20 matrix is explained in this section. Figure 3.12 represents the active time of each block. The Type I block is active at time 0. The first column of the input matrix A arrives at time step 0. For 20 clock cycles, the boundary cell

of Type I block computes C and S values and passes them to the internal cells and to RAM I. The last C and S value in the first row of Type I block is computed at 23^{rd} clock cycle since boundary cell takes 3 clock cycles for the computation. The $R_{1,1}$ value is generated at 23^{rd} clock cycle and $R_{1,2}$-$R_{1,4}$ are computed at 24^{th} clock cycle. Consequently, the $R_{2,2}$-$R_{4,4}$ values are computed and stored in the RAM III.

Type II block computations start at 17^{th} clock cycle. C and S values are read from RAM I at 16^{th} clock cycle and the column matrix A_5-A_8 are passed as inputs at 17_{th} clock cycle. The Type II block is active from 17_{th} clock cycle to 96^{th} clock cycle. During this time, it computes the R values $R_{1,5}$ to $R_{4,8}$. For the first four rows, the Type II block takes 20 clock cycles to compute the R values. At 17^{th} clock cycle, Type II block starts the computation of $R_{1,5}$ value and completes it at 36^{th} clock cycle. Then it starts the computation of $R_{1,9}$-$R_{4,12}$. The column matrices A_9-A_{12} and C and S values from RAM I are passed as inputs to Type II block at 37^{th} clock cycle. This repeats till 96^{th} clock cycle till $R_{1,13}$-$R_{4,16}$ values are computed and stored in RAM III.

At 29^{th} clock cycle, the Type II block computes the U_{out} values which are passed as inputs to the Type I block to compute R values from $R_{5,5}$-$R_{8,8}$. So the Type I block can start the computation at 29^{th} clock cycle. It takes 19 clock cycles to compute the $R_{5,5}$ value and hence the computation ends at 47^{th} clock cycle. Parallelly, the C and S values generated from the boundary cells are stored in RAM I.

To compute the R values $R_{5,9}$-$R_{8,12}$, the Type II block needs to wait to complete the computation of the previous 4 rows. Hence, it starts at the 97^{th} clock cycle. The $R_{5,9}$ value is computed at 112^{th} clock cycle after which the Type II block now starts the computation of $R_{5,13}$ value. To compute the $R_{5,13}$ to $R_{8,16}$ values, the Type II block requires U_{in} inputs from the previous Type II block. These values were stored in the RAM II during the computation of $R_{1,13}$ -$R_{4,16}$. The U_{in} values are read from

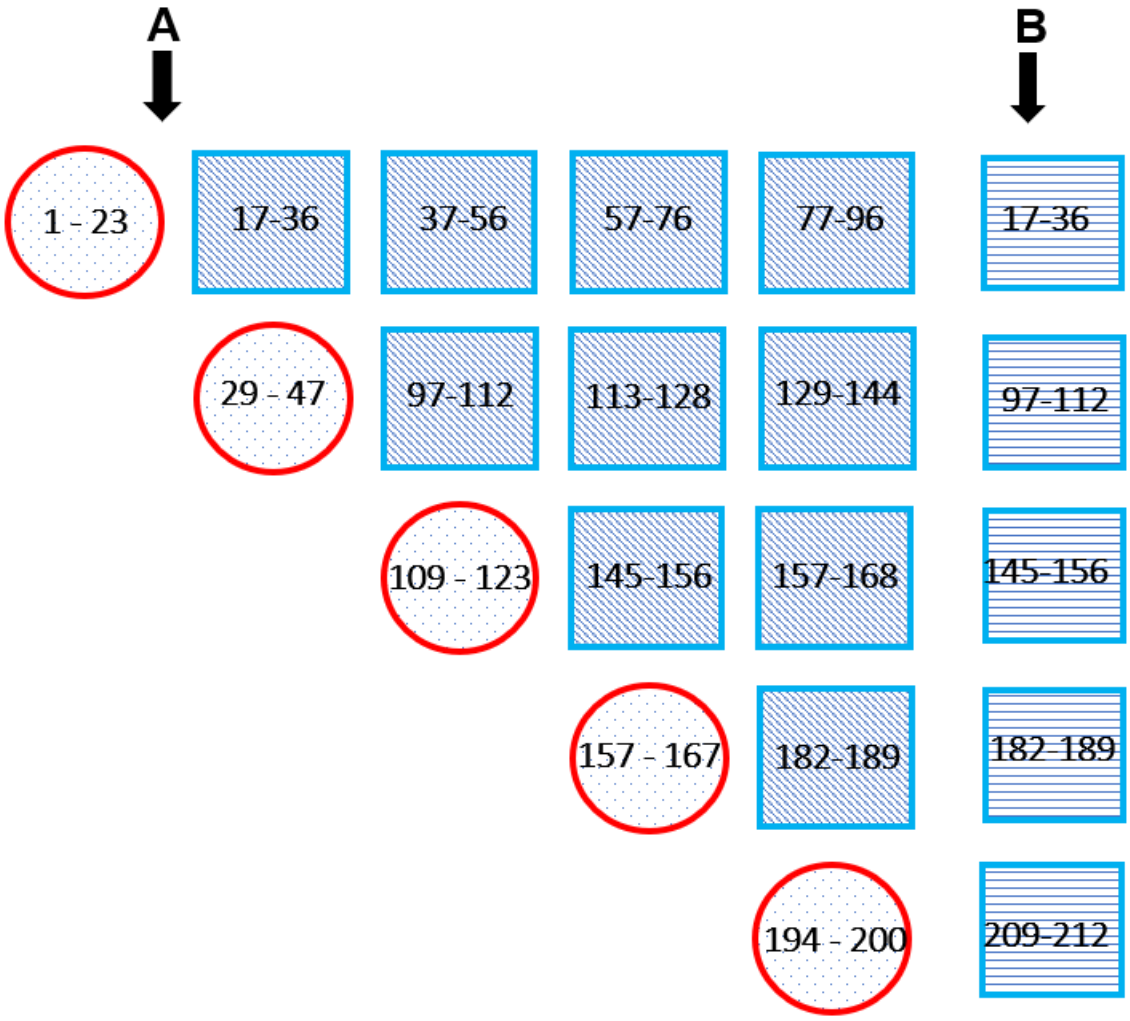

Figure 3.12: Schedule for Input Size 20x20

RAM II and passed to Type II block at 97^{th} clock cycle along with C and S values from RAM I.

The Type I block is activated at 109^{th} clock cycle to compute the values $R_{9,9}$-$R_{12,12}$. As described above, the Type II block is active from 145^{th} to 168^{th} clock cycle to compute the $R_{9,13}$-$R_{12,20}$ values.

To compute the $R_{13,13}$ to $R_{16,16}$ values, the Type I block is active from 157^{th} clock cycle to 167^{th} clock cycle. As the C and S values are generated and stored in RAM I, the Type II block is active from 182 to 189^{th} clock cyles to compute the R values

from $R_{13,17}$ to $R_{16,20}$. The final set of R values, $R_{16,16}$ to $R_{20,20}$ are computed from 194 to 200^{th} clock cycle.

The Type III block is active parallelly when first Type II computation of each row is active. The C and S values read at each row are passed as inputs to Type III block as well. The B‘ column values are stored in RAM III to be used in back substitution in backward path. The figure shows the active time of each blocks in the schedule for 20x20 input.

3.6 Memory Architecture

Recall that the scalable QR decomposition accelerator has a memory architecture with three different memories to store the rotation parameters RAM I, the data between two Type II blocks RAM II and the R values computed at each stage RAM III. The memory sizes depend on the input matrix size and increases as the input size increases.

RAM I is used to store the rotation parameters from the boundary cells. The computations in the i^{th} row require n-i+1 rotation parameters to be generated at the boundary cell. For instance, for an 8x8 input matrix, the first boundary cell generates 8 sets of C and S parameters. The boundary cell in second row generates 7 sets of C and S parameters. Since the computations are complex in nature, C is a real number but S is a complex number. Hence each set of C and S parameters consists of three values, C_{real}, S_{real} and $S_{complex}$. So a total of 3*(n-i+1) parameters needs to be saved in the RAM I for each row. With the folded architecture, there is an added advantage that all the Type II cells of each row are computed first before computing the R values in Type II cells in the next set of rows. Hence, as the clock cycles advance, the C and S values stored in the previous rows can be overridden once the parameters are no longer used.

RAM II is used to store the U_{out} values from the last row of internal cells of Type II cells. These are passed as inputs to the Type II cells in the next row. For Type II cells connected vertically, U_{out} needs to be saved to compute the R values after all the Type II computations of previous row are completed. The number of values stored depends on the size of input matrix. The number of U_{out} values at each row is equal to the row number. For example, in a 20x20 matrix, U_{out} values are stored for Type II cells computing $R_{5,9}$ to $R_{8,12}$, $R_{5,13}$ to $R_{8,16}$, $R_{5,17}$ to $R_{8,20}$. Therefore, each Type II cell computation requires 16 inputs, and since each Type II cell has four internal cells, 16x4x3 sets of values are stored in RAM II. For the next set of Type II computations, i.e., $R_{9,13}$ to $R_{12,16}$, $R_{9,17}$ to $R_{12,20}$, each Type II cell requires 12 inputs and hence 12x4x2 sets of values are stored. For the consequent computations, 8x4x1 set of values are stored. As the clock cycles advance, the RAM II memory can be overwritten to store subsequent U_{out} values, as the previously used values are no longer required. Hence, RAM II of size 528B is required to store the U_{in} values when input matrix size is 20x20.

In the QR accelerator, matrix R is computed and used for back substitution. Hence, at each stage, the values of matrix R from Type I and Type II cells are stored in RAM III. As the matrix size increases, the number of R values to store increases. For an nxn matrix, RAM III requires N(N+1)/2 values to be stored. Since, the R values are complex, the total number of values stored are n(n+1). These R values are read in back substitution to compute the column matrix X.

Chapter 4

BACK SUBSTITUTION ACCELERATOR

In the previous chapter, we have described a folded architecture for scalable QRD. In this architecture, the matrix R and the matrix B' are computed and stored in the memory unit RAM III. In back substitution, these two matrices are used to compute the column matrix X, where RX = B'. In this chapter, we describe the extensions that are needed to also compute back substitution.

The boundary and internal cell, the PEs which constitute the Type I and Type II blocks are now designed with added functionality for back substitution. The direction of computation is governed by an enable signal. The X values are computed in the boundary cells of Type I block. The internal cells of Type I and Type II block compute the intermediate values required to compute the X values.

4.1 Programming Elements

The two basic programming elements used in this design are still the boundary cell and internal cell. Recall that the boundary cells compute the rotational parameters and the internal cells compute the intermediate values using MAC operations in the forward computation. The same cells are now used in back substitution to compute the X values. The same boundary cell is used to compute X values in back substitution. A control signal is used to control the forward and backward computations in the cell. The following are the equations used in the back substitution:

$$X \leftarrow \frac{B'}{R} \tag{4.1}$$

In back substitution, the boundary cell computes the X values from inputs B_{in},

the input from column matrix B' and matrix R. The R value which was computed in the forward path is fetched from the memory during back substitution. The internal cell is used to compute intermediate values in the backward path in back substitution. The following are the equations used in the back substitution:

$$R'_{i,j} \leftarrow B'_i - R_{i,j}Xi \tag{4.2}$$

In the back substitution, the internal cell computes the R_{out} values from inputs B_{in}, X_{in} and R. The X_{in} value is fetched from memory which is computed in the boundary cell of the column where the internal cell is present. All internal cells in the same column use the same X_{in} values. B_{in} values are passed from other internal cells computed in the previous time step. The Figure 4.1 represents the data flow in the back substitution accelerator.

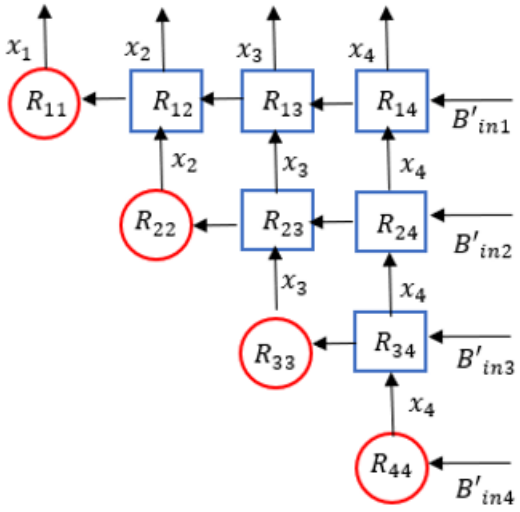

Figure 4.1: Data Flow in Back Substitution

4.2 Back Substitution Implementation

The back substitution implementation uses the same architecture as the foldable QR accelerator discussed in the previous chapter except that the direction of data flow

is reversed and the boundary and internal cells now have an additional back substitution functionality. In the algorithm used in back substitution, we use a bottom-up approach in the, i.e, X_n value is computed first followed by X_{n-1}, X_{n-2} and so on. The computation of X_1 value marks the completion of linear equation solver and both the forward and backward computation of the scalable architecture.

Figure 4.2 describes the block diagram of the proposed architecture. After the B'_n value is computed in the forward path, the en_bs signal is enabled. The $R_{n,n}$ value is read from the memory RAM III and passed as input to the last boundary cell of the Type I block. The X_n value thus computed in the boundary cell is propagated as input to all the other internal cells of the column in Type I block. Parallelly, the B' values from Type III block and R values from RAM III block are passed as inputs to compute the intermediate values. These values are passed as input to the next boundary cell in the Type I block to compute the X_{n-1} value. The X values thus computed are passed to another memory RAM IV which stores the X values to pass to the Type II cell to compute the intermediate values for the rest of the n-4 rows where nxn is the input matrix size. In this way, the X values are computed in each boundary cell and stored in the memory.

In back substitution, as the Type III cell finishes the computations in QRD accelerator, the values are stored in the RAM III memory. To start the back substitution computation, the B' values are read from the memory and passed to the Type I block. The R values are read from RAM III and passed as input to the last column of PEs in the Type I block. Every two clock cycles, the boundary cell computes the X values and stores them in RAM IV. Once the X value is computed in the last boundary cell, it is stored in RAM IV and read in subsequent clock cycles to pass as input to the Type II block.

In back substitution, the Type II block computes the intermediate values at every

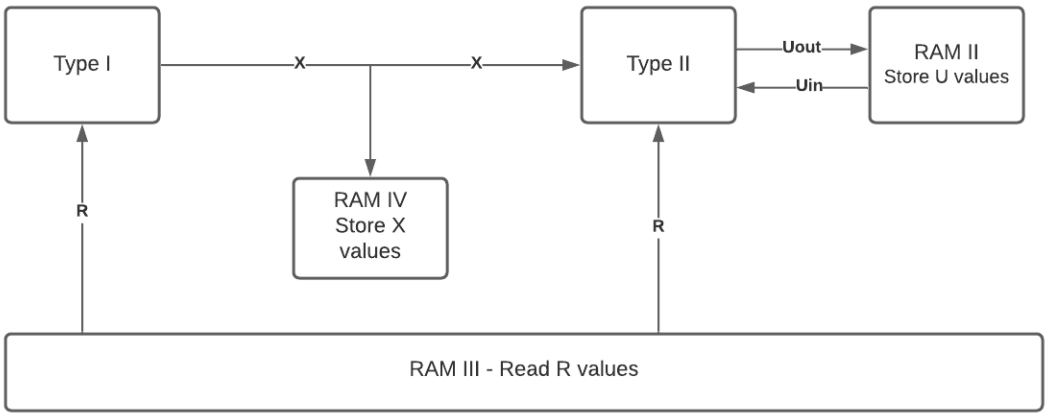

Figure 4.2: Block Diagram of Back Substitution Accelerator

clock cycle. Here, the computation starts from the last column and ends in the first column. The first column outputs are passed as inputs to the Type I or Type II blocks. Since there is no vertical dependency between two Type II cells, there is no need of RAM II in back substitution.

Once all the X values are computed, the back substitution is completed. All the X values are stored in RAM IV. The memory required for RAM IV depends on the size of the input matrix. For a matrix of size nxn, RAM IV needs to store n values.

4.3 Scalable Architecture for Back Substitution

This section gives a detailed description of schedule used in back substitution for different matrix sizes. The schedule used in the design follows a pattern which makes it flexible to use for any matrix size and it varies for each matrix size. The active time of each cell is one clock cycle since each cell boundary cell

4.4 Back Substitution for 4x4 Matrix

In this section, the back-substitution method is illustrated using a 4x4 matrix. For a 4x4 matrix as input, the foldable architecture uses only one Type I block and

one Type III block. At clock period 25, the Type III block finishes the forward computation and generates B_{in} input for the last boundary cell in the Type I block. This is because, the back substitution algorithm follows a bottom-up approach.The first X values X_4 is computed at 27^{th} clock cycle. This X value is propagated to the internal cells of the fourth column in the same time step.

The B' outputs from the Type III block are passed in the same clock cycle to these internal cells. Each internal cell calculates the intermediate value, and passes them as input to the next cell to the left. At clock cycle 29, the boundary cell in the third row computes the X_3 value and passes it as input to the internal cells in third row. These internal cells take the inputs from the boundary cell along with the inputs from the internal cells on the right to compute intermediate values. In a similar way, the boundary cell in second row computes the X_2 value at 31^{st} clock cycle and the boundary cell in first row computes the X_1 value at 33^{rd} row. At 33^{rd} clock cycle, all the X values are computed and the back substitution is completed.

4.5 Back Substitution for 8x8, 12x12, 16x16 matrices

In this section, the back substitution schedule for 8x8, 12x12 and 16x16 matrices is presented. For an input matrix of size 8x8, the forward computation ends at 54^{th} clock cycle. The schedule is shown in Figure 4.3. The boundary cell reads the B' input from RAM III and $R_{8,8}$ value from RAM III to compute the X_8 value. This value is passed as input to other internal cells in the 4^{th} column of Type I block. The third boundary cell in Type I block then computes the X_7 value from intermediate output computed in internal cell and $R_{7,7}$ value read from RAM III. The Type I computations end at 60^{th} clock cycle and the X values are stored in RAM IV. From 57^{th} to 63^{rd} clock cycle, Type II block reads the X values from RAM IV and computes the intermediate values. These are passed to the Type I cell for computing X_4 to X_1

values in 64^{th} tp 70^{th} clock cycle. In this way, the X values are computed in back substitution for an 8x8 matrix.

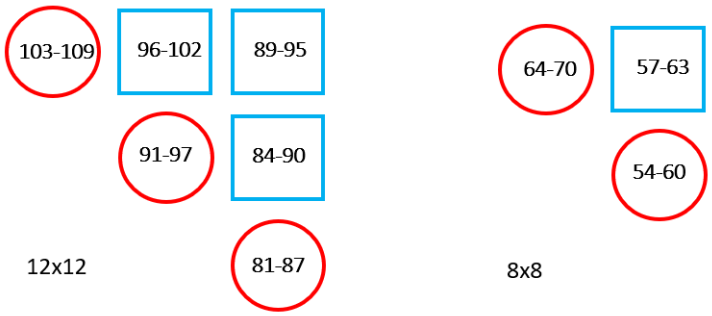

Figure 4.3: Schedule of Type I and Type II Blocks in Back Substitution for Input Size 12x12 and 8x8

For an input matrix of 12x12, the computations are similar to the 8x8 matrix. The corresponding schedule is presented in Figure 4.3. Type I block starts at 81^{st} clock cycle to compute X_12 to X_9 values. These are stored in RAM IV. Type II block then computes the intermediate values to pass as inputs to Type I in 84^{th} to 90^{th} clock cycle. Type I block then computes X_8 to X_5 values in 91^{st} to 97^{th} clock cycles. Type II block reads the X values from RAM IV and computes the intermediate values from 95^{th} to 102^{nd} clock cycles. Type I block then computes the X_4 to X_1 values in 103^{rd} to 109^{th} clock cycle. Back substitution finishes at 109^{th} clock cycle for a 12x12 matrix.

For an input matrix of size 16x16, the schedule for back substitution is presented in Figure 4.4. Type I block starts at 128^{th} clock cycle to compute X_16 to X_13 values. These are stored in RAM IV. Type II block then computes the intermediate values to pass as inputs to Type I in 131^{st} to 137^{th} clock cycle. Type I block then computes X_12 to X_9 values in 138^{th} to 144^{th} clock cycles. Type II block reads the X values from RAM IV and computes the intermediate values from 136^{th} to 149^{th} clock cycles. Type I block then computes the X_8 to X_5 values in 150^{th} to 156^{th} clock cycle. Type

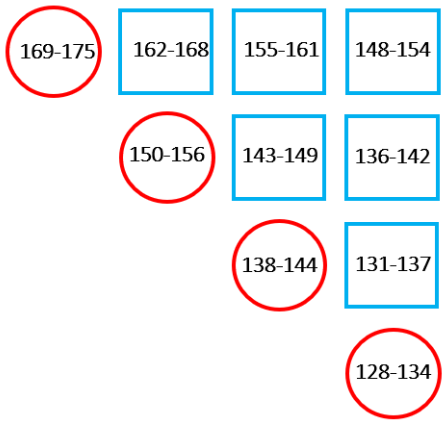

Figure 4.4: Schedule of Type I and Type II Blocks in Back Substitution for Input Size 16x16

II block reads the X values from RAM IV and computes the intermediate values from 148^{th} to 168^{th} clock cycles. Type I block then computes the X_4 to X_1 values in 169^{th} to 175^{th} clock cycle. Back substitution finishes at 175^{th} clock cycle for a 16x16 matrix.

4.6 Back Substitution for 20x20 matrix

In this section, the back-substitution method for a 20x20 matrix is described in details. Figure 4.5 represents the active time of each block during back substitution.

In a 20x20 matrix, the forward computation is completed at clock cycle 220. $B'_{20,20}$ value is computed in clock cycle 220 and passed as input to the Type I block. $R_{20,20}$ value is read from RAM III in the clock cycle 220. The boundary cell B_4 computed the X_{20} value at 221^{st} clock cycle. This value is passed as input to other internal cell in column 4 of Type I block as well as to write to memory RAM IV. The internal cells compute the intermediate values at 222^{nd} clock cycle and pass the output to the next cells connected to the left. The boundary cell in row 3 of the Type I cell computes the X_{19} value and passes it to the internal cells in the 3rd column. Using this X value and the inputs from the internal cells in 4th column, they compute the intermediate values and pass them to the cells on the left. The boundary cell in row

2 of the Type I cell computes the X_{18} value at 225^{th} clock cycle and the boundary cell in row 1 computes the X_{17} value at 227^{th} clock cycle.

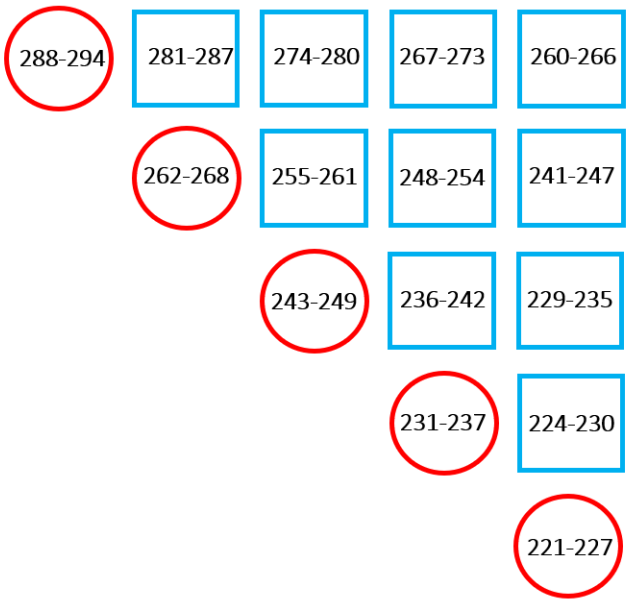

Figure 4.5: Schedule of Type I and Type II Blocks in Back Substitution for Input Size 20x20

The Type II block is used to compute intermediate values to pass to Type I cell to compute the $X_{16} - X_{13}$ values. As the X_{20} value is written to RAM IV, it is read as input to the Type II cell's last internal column cells I_{16}, I_{12}, I_8, I_4. In the next clock cycle, the X_{19} is passed as input to the 3rd column of Type II block along with $R_{13,19}$ to $R_{16,19}$ values and output from the internal cells in column 4. The Type II block completes the computation at 230th clock cycle and passes the outputs of 1st column of Type II block's internal cell to the Type I block. The next computations of the Type II block start at 229th clock cycle. A minimum delay of 5 clock cycles is required between two Type II computations for synchronization.

At 230th clock cycle, the Type I block computes the X_{16} value from inputs from Type II block and the $R_{16,16}$ value. Using the same pattern, $X_{16} - X_{13}$ values are generated and stored in RAM IV. The Type II block generates the intermediate values

following the schedule as shown in the figure. Finally, the Type I block generates the X1 value at 294th clock cycle marking the end of back substitution and linear equation solver for 20x20 matrix.

Chapter 5

RESULTS

In this chapter, the implementation results for the scalable QRD accelerator and back substitution accelerator are presented. We evaluate the architecture in terms of hardware utilization, timing summary and the error in dB for X and R values in comparison with the MATLAB results.

5.1 Architecture Configuration

The proposed folded architecture for scalable QRD+BS is built in MATLAB Simulink 2018b. Simulink library is used to build the programming elements, namely, the boundary, and the internal cell. The HDL coder library in Simulink is used to build the architecture. A separate library called QR_BW_block is created where the Type I, Type II and Type III blocks are built and saved. Three different memory blocks, the control circuitry, the selection blocks are saved in the QR_BW_block library.

A MATLAB testbench is built to drive the inputs and the scheduler. Additionally, the testbench generates the QR and linear equation solver matrices in software to compare with the Simulink results. This matrix is used as a reference to calculate the error in the generated results. The input matrix A is obtained from the correlation matrix output of receiver data generated in an inference mitigation system. All inputs are complex and represented by 40 bit signed fixed point with 38 fractional bits.

5.2 Precision Analysis

In this thesis, a signed fixed point data type is used to represent the data. With the help of precision tool designed by Shunyao Wu, the precision for this design has been calculated for an error constraint of -40dB. The MATLAB floating point values are taken as reference to calculate the error.

A software model of the architecture has been designed. The software model consists of the boundary and internal cells functionality designed in MATLAB code. These blocks are used in a loop to compute the matrix R. To calculate the right precision, the error between MATLAB and the Software model of the QR accelerator is calculate by sweeping the decimal point in the fixed point representation. The precision with the configuration which has the lowest error is taken as the final precision. Different data widths have also been tested and 40 bits has been the optimum number of bits required to represent the data.

Based on the simulation results, a precision of signed fixed point of length 40 bits is used for the ports. A fraction part of 38 bits is used in the data type with 1 bit as integer bit and 1 bit as signed bit. In the boundary cell, the reciprocal block uses a precision of 50 bits with 14 bits as fractional bits. Such a configuration ensured a -40dB error range for the R values in QR computation and X values in linear system of equations solver.

5.3 Simulation Results of the QR Accelerator

The timing results are presented in this section for matrix sizes 4x4, 8x8, 12x12, 16x16 and 20x20. For each matrix size, the time taken for the R computation is presented along with the error performance.

Matrix Size	Real	Complex
8x8	-59.81 dB	-56.59 dB
12x12	-58.66 dB	-54.75 dB
16x16	-55.50 dB	-52.50 dB
20x20	-52.62 dB	-47.89 dB

Table 5.1: Error in dB for Matrix R Computed Using QRD Accelerator for Different Matrix Sizes

5.3.1 Timing Results for Matrix Size 8x8, 12x12, 16x16, and 20x20

For the 8x8 matrix, the simulation takes 33 clock cycles. The R results are obtained at the end of each Type I and Type II computation and saved to the memory. The Type I cell takes 11 clock cycles in the first instantiation and 7 clock cycles in the second instantiation. The Type II cell takes 8 clock cycles to compute the R values.

For the 12x12 matrix, the forward computation takes 80 clock cycles to compute the upper triangular matrix. The folded architecture utilizes the Type I and Type III block thrice. Type II block is instantiated thrice for computing the R values. The first instantiation of Type I block takes 15 clock cycles, second instantiation takes 11 clock cycles and third instantiation takes 7 clock cycles.

For the 16x16 matrix, the forward computation takes 127 clock cycles to compute the matrix R. In the folded architecture, the Type I and Type III cells are instantiated four times and the Type II cell is instantiated six times. When compared with a flat architecture, the resource utilization in the foldable accelerator is minimized by 4x times with regard to Type I and Type III cells and 6x times to Type II cell.

For the 20x20 matrix, the forward computation takes 220 clock cycles to compute the R values.

R Matrix	Scalable Foldable Architecture	MATLAB
$R_{17,17}$	0.000680	0.000680
$R_{17,18}$	0.000248 + 0.000329i	0.000248 + 0.000329i
$R_{17,19}$	-0.000273 - 5.351431e-05i	-0.000273 - 5.351016e-05i
$R_{17,20}$	-3.514508e-05 - 0.000235i	-3.515331e-05 - 0.000235i
$R_{18,18}$	0.000786	0.000786
$R_{18,19}$	-0.000361 + 0.000214i	-0.000361 + 0.000214i
$R_{18,20}$	6.735430e-05 - 0.000409i	6.735457e-05 - 0.000409i
$R_{19,19}$	0.000581	0.000581
$R_{19,20}$	-5.801963e-05 + 4.599475e-05i	-5.802178e-05 + 4.598813e-05i
$R_{20,20}$	0.000634	0.000634

Table 5.2: R Values for 20x20 Matrix Obtained by fixed point Simulink implementation of proposed architecture and floating point MATLAB implementation

5.3.2 Error Performance

Table 5.1 shows the difference between the fixed point Simulink results and the floating point MATLAB results (reference) for the different matrix sizes. The error performance is represented by $10log_{10}E$, where E is the normalized absolute error between the Simulink and MATLAB results. From these results, we see that the error is very small.

Table 5.2 shows the last four rows of matrix R computed in the folded architecture. The entries are almost identical, further demonstrating that the choice of bit widths was sufficient to guarantee superior algorithm performance.

5.4 Simulation results of the Back Substitution accelerator

The simulation results for back substitution of matrix sizes 8x8, 12x12, 16x16 and 20x20 are presented in this section. The time taken for computing the matrix X in back substitution are presented along with the error performance.

Matrix Size	Real	Complex
8x8	-36.84 dB	-31.59 dB
12x12	-36.24 dB	-31.72 dB
16x16	-36.51 dB	-32.11 dB
20x20	-36.88 dB	-33.59 dB

Table 5.3: Error in dB for X Values Computed Using Back Substitution Accelerator for Different Matrix Sizes

In backward computation, X values are computed from the R values obtained by QRD. For an 8x8 matrix, 16 clock cycles are taken to compute the eight X values. It takes 28 clock cycles to compute the X values for an input size of 12x12. For a 16x16 matrix, 47 clock cycles are taken to compute the X values. For a 20x20 matrix, 73 clock cycles are taken to compute the X values.

Table 5.3 presents the error performance for each input matrix size. We see that while the errors are still very low, they are larger than that of QRD. This is to be expected since the errors in QRD are propagated to the back substitution part. Table 5.4 lists the X values computed in the folded architecture compared to the MATLAB results. We see that the X values generated from the flexible QRD and back substitution accelerator are almost equal to the floating point values generated in MATLAB.

5.5 Hardware Implementation

The flexible QRD accelerator and back substitution for a 4x4 design has been converted to hardware description and targeted to an FPGA. This design has been targeted to the Xilinx's Zynq Ultrascale+ evaluation board ZCU102. A design frequency of 50 MHz has been used for the design. The design is implemented on Vivado Design Suite 2018.3.

X	Scalable Foldable Architecture	MATLAB
X_1	6.878399e-07 + 2.695625e-06i	7.310825e-07 + 2.715244e-06i
X_2	3.417931e-05 - 8.625340e-05i	3.431856e-05 - 8.620187e-05i
X_3	2.922260e-05 - 6.648626e-05i	2.935061e-05 - 6.645610e-05i
X_4	-1.497078e-05 + 5.040541e-05i	-1.490641e-05 + 5.043102e-05i
X_5	7.387381e-05 + 1.049347e-05i	7.391894e-05 + 1.051470e-05i
X_6	0.000153 - 0.000135i	0.000153 - 0.000135i
X_7	0.000217 - 0.000164i	0.000218 - 0.000164i
X_8	4.926616e-06 - 9.052360e-05i	5.000186e-06 - 9.051137e-05i
X_9	8.415263e-05 + 6.147030e-05i	8.419702e-05 + 6.152077e-05i
X_{10}	0.000229 - 0.000103i	0.000229 - 0.000103i
X_{11}	0.000316 - 5.126465e-05i	0.000316 - 5.116090e-05i
X_{12}	0.000248 - 9.707633e-05i	0.000248 - 9.702033e-05i
X_{13}	0.000166 - 7.486414e-07i	0.000166 - 6.688314e-07i
X_{14}	0.000333 + 4.998104e-05i	0.000333 + 5.009759e-05i
X_{15}	0.000421 - 2.105430e-05i	0.000421 - 2.095375e-05i
X_{16}	0.000211 - 8.725915e-05i	0.000211 - 8.721449e-05i
X_{17}	0.000253 + 8.163505e-05i	0.000253 + 8.172965e-05i
X_{18}	0.000306 + 0.000102i	0.000306 + 0.000102i
X_{19}	0.000377 + 0.000148i	0.000377 + 0.000148i
X_{20}	0.000322 - 1.690412e-06i	0.000322 - 1.639833e-06i

Table 5.4: X Values for 20x20 Matrix

5.5.1 Programming Elements

The synthesis results of the programming elements (PE) are presented in this section. The boundary and internal cell which are the building blocks of the Type I, Type II and Type III blocks are implemented individually. Table 5.5 represents the resource utilization of these two types of cells.

Resources	Boundary Cell	Internal Cell
LUT	4109(1.5%)	3222(1.18%)
LUTRAM	242(0.17%)	80(0.06%)
FF	1534(0.28%)	2551(0.47%)
DSP	21(0.83%)	90(3.57%)

Table 5.5: Resource Utilization of Boundary and Internal Cells

The PEs operate at 50 MHz clock frequency with 1.462 ns positive slack in boundary cell and 15.098 ns positive slack in internal cell. The reciprocal and square root blocks generated in Simulink have a very high clock period. Therefore, these blocks have been modified to support a pipelined architecture which takes 3 clock cycles to compute the operations in the boundary cell. In collaboration with Yang Li, the number of iterations in the hardware reciprocal have been modified without effecting the error performance. This helped in achieving the 50MHz operating frequency.

5.5.2 4x4 Implementation

This section presents the synthesis and timing results of the 4x4 implementation. The boundary and internal cells implemented in the previous section are used as building blocks for designing the 4x4 implementation. The 4x4 implementation operates at 50 MHz with a positive slack of 1.318 ns. Table 5.6 represents the synthesis results of 4x4 implementation.

Resources	4x4 Implementation
LUT	33156(11.11%)
LUTRAM	1361(0.92%)
FF	13532(2.27%)
DSP	594(20.29%)

Table 5.6: Resource Utilization of 4x4 Implementation

5.5.3 Implementation of Type I , Type II and Type III blocks

This section represents the synthesis results of the building blocks of the design, the Type I , Type II and Type III cells. These blocks use 32.68% of the LUTs, 3.82% of the LUTRAMs, 9.39% of the FFs and 80.86% of the DSPs of the ZCU102. These three blocks can be used to compute matrix R for any matrix size. Table 5.7 shows the resource utilization of each of the blocks.

Resources	Type I	Type II	Type III
LUT	33156(11.11%)	49639(18.11%)	9495(3.46%)
LUTRAM	1361(0.92%)	3450(2.4%)	726(0.5%)
FF	13532(2.27%)	32344(5.9%)	6680(1.22%)
DSP	594(20.29%)	1280(50.79%)	240(9.52%)

Table 5.7: Resource Utilization of Type I, Type II, and Type III Blocks

The Type I cell operates at the longest clock period of 20 ns with 1.318 ns of positive set up slack time. This is the slowest block of the design since it has the boundary cell which is computationally complex.

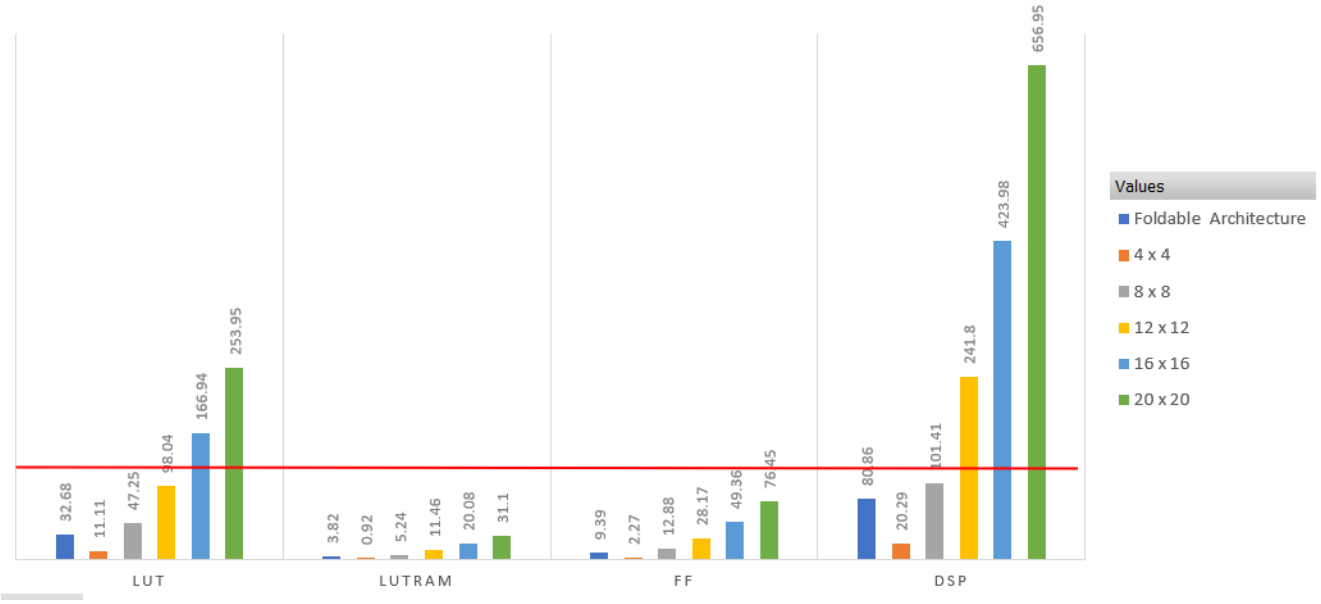

Figure 5.1: Comparision of Foldable Architecture for Different Matrix Sizes With Flat Architecture

5.5.4 Analysis of Resource Utilization

The folded architecture is designed to minimize the FPGA resource utilization for QR decomposition and linear equation solver of large matrices. The proposed architecture offers the flexibility to compute the matrix R and matrix X for any matrix size. Figure 5.1 shows the resource utilization of the foldable architecture for each matrix size in comparison with a flat architecture. The red line increases the point of 100% resource utilization of the ZCU102 FPGA.

It can be observed from Figure 5.1 that matrix size of 12 and greater has a resource utilization of greater than 100%. This shows that the flat architecture is not feasible for computing matrix R and matrix X using a single FPGA chip. The folded architecture makes optimum usage of resources supporting flexibility and scalability in matrix computation.

5.5.5 Memory Size Requirements

In this section we present the memory requirements of the proposed folded architecture. Table 5.8 lists the number of values stored in each memory unit for a matrix of size nxn. Specifically, RAM I stores the C and S values of first 8 rows. RAM II stores the data passed between two internal cells connected vertically. Hence it has to store the U_{out} values for the first 8 rows. Matrix sizes of 4x4 and 8x8 do not require vertical data flow between two Type II blocks and thus their RAM II memory requirement is zero. RAM III stores the R values computed in each PE. Therefore, it requires a memory to store n*(n+1) values. Finally RAM IV stores n values of column X. Thus the total number of values stored in the four memories for a matrix size of nxn is given by equation 5.1.

$$(5n^2 - 38n + 172) \tag{5.1}$$

Memory	Number of values stored
$RAMI$	24n - 84
$RAMII$	$4(\mathrm{n}^2 - 16n + 64)$
$RAMIII$	$\mathrm{n}^2 + n$
$RAMIV$	n

Table 5.8: Number of Values Stored in Each Memory for a Matrix of Size nxn

Table 5.9 lists the memory requirements of all matrix sizes supported by this architecture. The largest matrix size that has been considered is 20x20 and thus the folded architecture required a total of 7KB of memory.

Memory	4x4	8x8	12x12	16x16	20x20
RAMI	0.15KB	0.54KB	1.02KB	1.5KB	1.98KB
RAMII	0KB	0KB	0.32KB	0.96KB	2.64KB
RAMIII	0.14KB	0.44KB	0.84KB	1.52KB	2.2KB
RAMIV	0.04KB	0.08KB	0.12KB	0.16KB	0.20KB

Table 5.9: Memory Required for Matrices of Different Sizes

5.5.6 Maximum Matrix Size Supported

We have seen that the flexible QRD and back substitution accelerator supports any matrix size which is multiple of four. The constraint to support any matrix size is the maximum size of memory required for each matrix size. RAM II in the design is the memory which stores maximum number of intermediate values in the design and this increases as the matrix size increase. This is because, RAM II stores the intermediate outputs between Type II blocks and number of Type II blocks increase with increase in matrix size. The maximum memory size supported on ZCU102 FPGA is 4104 KB, which limits the maximum matrix size that can be supported by this architecture. The total memory required to store these values is given by $5 * (5n^2 - 38n + 172)$ since each value requires 5B. Hence the maximum matrix size supported by the proposed architecture is 412x412.

Chapter 6

CONCLUSION AND FUTURE WORK

6.1 Summary

In this thesis, a flexible and scalable implementation of QR Decomposition and back substitution is presented. It is shown that this architecture can be used to compute matrix R in QR and back substitution to compute X for any matrix size which is a multiple of 4.

Givens rotation is used to implement QRD and back substitution. It is a systolic array architecture consisting of two types of programming elements namely, boundary cell and internal cell. Boundary cell is a compute intensive block consisting of modules to compute square root and reciprocal operations. In order to decrease the latency, a pipelined architecture for boundary cell is implemented. It takes 3 clock cycles to compute the rotations in boundary cell with this optimization.

The proposed folded architecture is composed of three types of blocks, the Type I , Type II and Type III. Type I block consists of a 4x4 triangular matrix with four boundary cells and six internal cells. Type II block is a 4x4 rectangular matrix of sixteen internal cells and Type III block is a 4x1 column matrix of four internal cells which computes the $Q^{-1}B$, where Q is the orthogonal matrix in QR and B is the column input in $AX{=}B$.

A scheduler is designed which controls the programming blocks to support any matrix size input. It controls the memory read and writes from different blocks. The schedule pattern for each matrix size is presented in the thesis. This helps derive the active time of each of the basic computation blocks.

The flexible QRD accelerator requires four different types of memory to store intermediate values in the folded architecture. The memory size required for the flexible QR accelerator depends on the maximum size of the input matrix. In this work, a maximum size of 20x20 is used as input. The memory required to support 20x20 computations is 1.98KB for RAM I , 2.64 KB for RAM II, 2.2KB for RAM III and 200B for RAM IV.

It is shown that the flexible QRD architecture is used to compute back substitution using the same programming elements of QR accelerator. A control signal is used to control the data flow direction in QR and back substitution. This is illustrated using matrix sizes 4x4, 8x8, 12x12, 16x16 and 20x20. The scheduler and memory sizes required in the back substitution is also presented.

The flexible QRD and back substitution accelerator can compute QRD and back substitution for any matrix size using the same architecture, thereby improving the resource utilization significantly when compared to implementing a flat architecture for the same input matrix size. Such a design can be easily implemented on a single FPGA unlike the flat architecture. The same hardware is used in both forward and backward computations controlled by an enable signal. Thus the proposed folded architecture is useful for applications involving multiple matrix sizes where hardware resources are constrained.

6.2 Future work

The work presented in this thesis is an initial exploration to provide a unified accelerator for both QR decomposition and back substitution for any matrix size.

- This work presents the synthesis results of flexible QRD and back substitution accelerator targeted on a Xilinx Zynq Utlrascale+ FPGA. It can be implemented on the hardware to compute QRD in real time.

- The input matrix size currently supported are multiples of four such as 4x4, 8x8, 12x12, etc. This can be extended to support any input matrix size.

- The programming elements in this architecture support the QRD and back substitution. This can be made more flexible by supporting other matrix decompositon such as LU decomposition, Cholesky decomposition. The functionality for different algorithms can be added to the programming elements and enabled using the scheduler.

- As the matrix size increases, the usage of Type II blocks in the computation increases quadratically. To address this, for higher matrix sizes, the architecture can be modified by adding two Type II blocks. The scheduler would then need to be updated. Such a design would also lead to increase in memory as well.

.

REFERENCES

[1] Adaptive beamformer with QRD, `https://www.intel.cn/content/dam/www/programmable/us/en/pdfs/literature/ds/adaptive-beamformer-with-qrd.pdf`, 2020. [Online; accessed November 7, 2020].

[2] J. APOLINARIO. *QRD-RLS Adaptive Filtering*. Lecture notes in mathematics. Springer US, 2009.

[3] S. Aubert, M. Mohaisen, F. Nouvel, and K. Chang. Parallel QR decomposition in lte-a systems. In *2010 IEEE 11th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, pages 1–5, 2010.

[4] H. Bölcskei, D. Gesbert, C.B. Papadias, and A.J. van der Veen. *Space-Time Wireless Systems: From Array Processing to MIMO Communications*. Cambridge University Press, 2006.

[5] D. Boppana, K. Dhanoa, and J. Kempa. FPGA based embedded processing architecture for the QRD-RLS algorithm. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 330–331, 2004.

[6] D. Chen and M. Sima. Fixed-point CORDIC-based QR decomposition by givens rotations on FPGA. In *2011 International Conference on Reconfigurable Computing and FPGAs*, pages 327–332, 2011.

[7] E. N. Frantzeskakis and K. J. R. Liu. A class of square root and division free algorithms and architectures for QRD-based adaptive signal processing. *IEEE Transactions on Signal Processing*, 42(9):2455–2469, 1994.

[8] L. Gao and K. K. Parhi. Hierarchical pipelining and folding of QRD-RLS adaptive filters. In *2000 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.00CH37100)*, volume 6, pages 3283–3286, 2000.

[9] W. M Gentleman. Least Squares Computations by Givens Transformations Without Square Roots. *IMA Journal of Applied Mathematics*, 12(3):329–336, 12 1973.

[10] W. M. Gentleman and H. T. Kung. Matrix Triangularization By Systolic Arrays. In Tien F. Tao, editor, *Real-Time Signal Processing IV*, volume 0298, pages 19 – 26. International Society for Optics and Photonics, SPIE, 1982.

[11] W. Givens. Computation of plane unitary rotations transforming a general matrix to triangular form. *Journal of the Society for Industrial and Applied Mathematics*, 6(1):26–50, 1958.

[12] A. S. Householder. Unitary triangularization of a nonsymmetric matrix. *J. ACM*, 5(4):339–342, October 1958.

[13] S. F. Hsieh, K. J. R. Liu, and K. Yao. A unified square-root-free approach for QRD-based recursive-least-squares estimation. *IEEE Transactions on Signal Processing*, 41(3):1405–1409, 1993.

[14] A. Irturk, B. Benson, A. Arfaee, and R. Kastner. Automatic generation of decomposition based matrix inversion architectures. In *2008 International Conference on Field-Programmable Technology*, pages 373–376, 2008.

[15] R. Alvarez O. L. Gandara J O. Aguilar J. V. Castillo, A. C. Atoche. FPGA-based hardware matrix inversion architecture using hybrid piecewise polynomial approximation systolic cells. In *Approximate Computing: Design, Acceleration, Validation and Testing of Circuits, Architectures and Algorithms in Future Systems*, pages 1–6, 2020.

[16] M. Karkooti, J. R. Cavallaro, and C. Dick. FPGA implementation of matrix inversion using QRD-RLS algorithm. In *Conference Record of the Thirty-Ninth Asilomar Conference onSignals, Systems and Computers, 2005.*, pages 1625–1629, 2005.

[17] M. Langhammer and B. Pasca. High-performance QR decomposition for FPGAs. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, page 183–188, New York, NY, USA, 2018. Association for Computing Machinery.

[18] C. Liu, C. Tang, Z. Xing, L. Yuan, and Y. Zhang. Hardware architecture based on parallel tiled QRD algorithm for future MIMO systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(5):1714–1724, 2017.

[19] V. Mauer and M. Parker. Floating point STAP implementation on FPGAs. In *2011 IEEE RadarCon (RADAR)*, pages 901–904, 2011.

[20] F. Merchant, T. Vatwani, A. Chattopadhyay, S. Raha, S. K. Nandy, R. Narayan, and R. Leupers. Efficient realization of givens rotation through algorithm-architecture co-design for acceleration of QR factorization, 2018.

[21] M. Najoui, A. Hatim, and S. Belkouch. Novel parallel givens QR decomposition implementation on VLIW architecture with efficient memory access for real time image processing applications. In *Proceedings of the 2nd International Conference on Big Data, Cloud and Applications*, BDCA'17, New York, NY, USA, 2017. Association for Computing Machinery.

[22] G. R. Prabhu, B. Johnson, and J. S. Rani. FPGA based scalable fixed point QRD core using dynamic partial reconfiguration. In *2015 28th International Conference on VLSI Design*, pages 345–350, 2015.

[23] L. Pursell and S. Y. Trimble. Gram-schmidt orthogonalization by gauss elimination. *The American Mathematical Monthly*, 98(6):544–549, 1991.

[24] P. Salmela, A. Burian, H. Sorokin, and J. Takala. Complex-valued QR decomposition implementation for MIMO receivers. In *2008 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 1433–1436, 2008.

[25] D. Shin and J. Park. A low-latency and area-efficient gram–schmidt-based QRD architecture for MIMO receiver. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(8):2606–2616, 2018.

[26] P. Sirisuk, F. Morgan, T. El-Ghazawi, and H. Amano. *Reconfigurable Computing: Architectures, Tools and Applications: 6th International Symposium, ARC 2010, Bangkok, Thailand, March 17-19, 2010, Proceedings*. LNCS sublibrary: Theoretical computer science and general issues. Springer, 2010.

[27] J. Zhang, P. Chow, and H. Liu. CORDIC-based enhanced systolic array architecture for QR decomposition. *ACM Trans. Reconfigurable Technol. Syst.*, 9(2), December 2015.