A QR Decomposition Accelerator for Digital Beamforming

Vinay Singh
vi7715si-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Victor Åberg, Lund University

Industrial Supervisor:
Joakim Axmon, BeammWave AB
Samir Kumar Mishra, BeammWave AB

Examiner: Pietro Andreani

March 11, 2026

Printed in Sweden
Tryckeriet i E-huset, Lund

Statement of originality

I hereby affirm that this Master thesis was composed by myself, that the work herein is my own except where explicitly stated otherwise in the text. This work has not been submitted for any other degree or professional qualification except as specified; nor has it been published. Where other people's work has been used (either from a printed source, internet, or any other source), this has been carefully acknowledged and referenced.

During the preparation of this thesis, I have used ChatGPT and Google Gemini to assist me in the writing process to improve language, flow, and readability. After using this tool/service, I have reviewed and edited the content as needed and I take full responsibility for the content of the whole thesis.

Abstract

QR decomposition (QRD) is a computationally-intensive matrix factorization method which is widely used in signal processing. Meeting the stringent processing budgets of real-time applications necessitates dedicated hardware acceleration. This thesis presents the architecture and Register-Transfer-Level (RTL) implementation of a QRD accelerator based on Givens rotations, specifically designed to achieve an end-to-end latency of $\leq$ 50 μs for an 8 $\times$ 8 complex covariance matrix. The architecture transforms the complex input into a 16 $\times$ 16 realified representation, which is processed by a CORDIC-based datapath to compute both the **Q** and **R** matrices. To manage data dependencies, the design employs a stage-wise binary-tree elimination schedule enforced by a hard memory-visibility barrier. While the compute core is fully pipelined with a fixed latency of 16 cycles, the system throughput is governed by the on-chip memory service model, resulting in a sustained initiation interval of 2.

Functional correctness is established via bit-exact verification against a golden fixed-point C reference model using `Q1.15` arithmetic. The design was implemented and validated on a Zynq™ UltraScale+™ Field Programmable Gate Array (FPGA) development board. Operating at 245.76 MHz, the accelerator completes a single QRD in 9.83 μs, satisfying the target requirement with a significant performance margin and achieving a sustained throughput of approximately 101 kQRD/s. The results indicate that end-to-end latency is primarily dominated by system-level data movement and synchronization barriers rather than raw arithmetic computation. These findings motivate future research into relaxed consistency models and inter-stage data forwarding to further optimize scaling for higher-dimensional matrices. The novel architecture and scheduling methodology developed in this work has been filed for patent protection.

Popular Science Summary

Delivering 5G by Speeding Up the Math— How can 5G become faster and more reliable, especially in crowded places? One important piece is how quickly a base station can do the math needed to aim its radio beams.

A 5G base station does not send the same signal in every direction like a lightbulb. Instead, it can focus energy into narrow beams and points them toward different users. This is called *beamforming*. It helps users get stronger signals and reduces interference.

To aim these beams, the base station must repeatedly solve a heavy math problem called *QR Decomposition (QRD)*. The radio environment changes very quickly, so this math has to be done again and again within a very short time window. If the calculation takes too long, the beam settings become outdated and performance drops.

This thesis presents a dedicated hardware accelerator—a small specialized "engine"— built to run this QRD math much faster than a general-purpose processor. The key idea is to do more work in parallel. A simple analogy is a sports tournament: many matches happen at the same time, and only the winners move forward. In the same way, the accelerator arranges the QRD steps so that many parts can run simultaneously, instead of one after another.

With this approach, the design can finish one QRD in 9.83 μs at 245.76 MHz, well below the project target of $\leq$ 50 μs. For comparison, a human blink takes roughly 100,000 μs. At this speed, the accelerator can handle about 100,000 QRDs per second.

By reducing this processing delay, base stations can update beam directions more quickly and serve more users efficiently. The scheduling method used in this work was considered novel enough to be filed for patent protection.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Victor Åberg, for his invaluable guidance and continued encouragement.

I am also deeply thankful to my industrial supervisors at BeammWave AB, Joakim Axmon and Samir Kumar Mishra, for their technical mentorship and for the opportunity to work on such a challenging project. I would also like to thank Per-Olof Brandt for his early encouragement, which got me started in the first place. Finally, I would like to thank my family and friends for their unwavering support throughout my master's studies.

Contents

List of Figures

List of Tables

Acronyms

1R1W One Read One Write.

ABF Analog Beamforming.

ASIC Application-Specific Integrated Circuit.

BC Boundary Cell.

BS Base Station.

BT Binary-tree.

CGRAs Coarse-Grained Reconfigurable Architectures.

CORDIC Coordinate Rotation Digital Computer.

CSI Channel State Information.

CSR Configuration and Status Registers.

CV complex-valued.

DBF Digital Beamforming.

DBFA digital beamforming accelerator.

DUT Design Under Test.

FB Feed Buffer.

FP Fixed-pivot.

FPGA Field-Programmable Gate Array.

GPU Graphics Processing Units.

GR Givens rotation.

GS Gram-Schmidt process.

HBF Hybrid Beamforming.

HH Householder Transformation.

MAC Memory Access Controller.

MIMO Multiple-Input Multiple-Output.

PE Processing Element.

PULP Parallel Ultra Low Power.

Q Engine Q ENGINE.

QR ENGINE QR Engine.

QRD QR Decomposition.

R Engine R ENGINE.

RAW Read-after-Write.

RF radio-frequency.

ROB Re-Order Buffer.

RSB Rotation Sequence Broadcaster.

RTL Register-Transfer Level.

RV real-valued.

SV SystemVerilog.

UEs User Equipments.

WBB Write Back Buffer.

Notation

Symbol	Description
$\mathbb{R}^{N\times N}$	Set of real-valued matrices of dimension $N \times N$.
$\mathbb{C}^{N\times N}$	Set of complex-valued matrices of dimension $N \times N$.
$\mathbf{A}$	Input matrix (general).
$\mathbf{Q}$	Orthogonal matrix resulting from QR decomposition.
$\mathbf{R}$	Upper triangular matrix resulting from QR decomposition.
N_R	Number of receiving antennas (dimension of complex matrix).
$\mathbf{C}$	Complex-valued covariance matrix ($N_R \times N_R$).
$\mathbf{C}_r$, $\mathbf{A}_r$	Realified covariance matrix ($2N_R \times 2N_R$).
$\mathbf{G_k}(i, j)$	Givens rotation matrix eliminating element at row j, column k.
c, s	Cosine and sine parameters for Givens rotation.
L	Compute pipeline latency (cycles).
II	Initiation interval (cycles).
T_{QRD}	Total execution time for one QR decomposition (seconds).
f_{dp}	Datapath clock frequency (Hz).
$P_{j,s}$	Number of independent row-pairs in stage s.
G_s	Inter-stage gap/penalty (cycles).
W_{line}	Width of a memory line (bytes).
B_{row}	Number of memory beats required to transfer one row.

Chapter 1

Introduction

In a conventional wireless communication system, a single unit called a Base Station (BS) communicates with multiple User Equipments (UEs) over shared spectrum resources. Figure 1.1 illustrates a typical base station scenario.

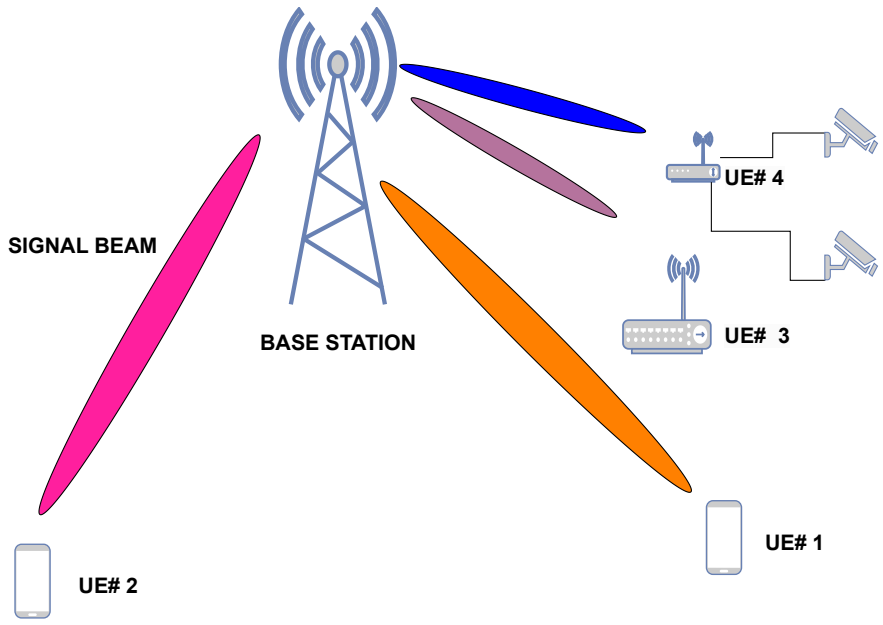


Figure 1.1: A typical base station communication structure.

With the ever-increasing demand for high-speed data [1], efficient utilization of limited spectrum resources has become essential. Beamforming [2] is one such key technology that efficiently utilizes the wireless channel by precisely controlling the amplitude, phase as well as direction of the signals.

Beamforming techniques are widely adopted and can be grouped into three categories [3, 4]:

Analog Beamforming (ABF): In ABF, a single radio-frequency (RF) mod-

ulated data stream is fed into an antenna array. Phase shifters are placed before each antenna to alter the phase of the transmitted signal and thereby control the beam direction [5].

Digital Beamforming (DBF): In DBF, the boundary between the analog and digital domains is moved to the individual antenna level. Each antenna element is equipped with a dedicated RF chain and data converters, allowing signal amplitude and phase to be manipulated entirely in the digital domain [6]. This provides the highest degree of freedom for spatial filtering, enabling the generation of multiple simultaneous beams and precise null-steering for interference cancellation.

Hybrid Beamforming (HBF): In HBF, a compromise between ABF and DBF is made. It reduces RF cost and power by using N_{RF} RF chains to drive N_{ant} antennas, typically $N_{\mathrm{s}} \leq N_{\mathrm{RF}} \ll N_{\mathrm{ant}}$, where N_{s} denotes the number of transmitted data streams [7]. Beamforming is split into a digital precoder across RF chains and an analog beamforming network that maps RF chains to antennas. Depending on the antenna connectivity, hybrid beamformers are classified as *fully connected* or *partially connected* (subarray-based) [8]: fully connected designs connect each RF chain to all antennas whereas partially connected designs connect each RF chain to a subset of antennas.

The need for higher capacity in 5G and beyond pushes systems towards serving multiple users and multiple data streams at the same time. Simply shifting to higher carrier frequencies is not sufficient to meet the capacity demands, since the usable spectrum is limited and higher bands come with propagation constraints [9]. Therefore fine-grained control of beams, robust interference suppression, and a scalable way to increase the number of independent channels as the antenna count grows is needed. To achieve this, the beamforming techniques mentioned above are used. However, these techniques come with their own trade-offs.

ABF is relatively cheap and energy efficient because it minimizes the number of data converters and RF chains. However, it does not scale well when multiple independent beams are required. Since the antenna array is driven by a single or a very small number of RF chains, ABF mainly provides *array gain* for one dominant beam. When servicing multiple users, the single beam needs to be swept in a time-multiplexed manner, thereby limiting the multi-user capacity [5].

HBF is introduced to overcome the shortcomings of ABF and DBF. It combines a digital stage with an analog beamforming network, thereby reducing the number of RF chains. However, because the number of RF chains is still smaller than the number of antennas, the number of independent streams remains limited. The analog stage also constrains the beamformer structure, which reduces flexibility for interference management. As a result, HBF typically relies on adaptive, codebook-based operation to balance performance against limited analog hardware resources [8, 10].

In DBF, each antenna element has its own RF chain. This allows precise multi-beam control and stronger interference suppression. The flexibility offered by DBF comes at the cost of immense computational complexity; since beamforming weights are applied to every digital stream, the system must perform

high-dimensional matrix factorizations, such as QR Decomposition (QRD), in real-time to keep up with rapidly varying Channel State Information (CSI). This leads to increased hardware complexity and power consumption due to one RF chain and data converter per antenna element [11].

Unlike other beamforming techniques, DBF offers finer beam control, simultaneous multi-user and multi-stream operation. This effectively translates to better channel utilization, lower latency, and higher data rates. Because of its overall flexibility and scalability, DBF is a key enabler for mmWave 5G and future wireless systems, and motivates research into efficient algorithms and hardware architectures that make fully digital solutions practical.

1.1 QR Decomposition in Digital Beamforming

In high-frequency 5G mmWave systems, the wireless channel can vary rapidly due to mobility. To maintain robust and low-latency communication, the device must frequently update its CSI at the receiver side.

BeammWave™ AB has developed scalable algorithms to perform DBF in a power-efficient way. These algorithms are deployed on the receiver side of the device and rely on repeated matrix computations using the continuously updated CSI to compute beamforming steering vectors. The computed steering vectors are then utilized for beamforming during transmission. In time-varying mmWave channels, this processing must be executed frequently and within tight real-time budgets, thereby presenting a bottleneck for scaling the algorithms in massive Multiple-Input Multiple-Output (MIMO) systems. A critical step in the formation of steering vectors is a matrix factorization method called QRD [12, Sec 5.2]. QRD factors a matrix $\mathbf{A}$ into an orthogonal matrix $\mathbf{Q}$ and an upper triangular matrix $\mathbf{R}$:

$$\mathbf{A} = \mathbf{QR}. \tag{1.1}$$

For an $N \times N$ matrix, the computational complexity of QRD scales as $\mathcal{O}(N^3)$ [12, Sec 5.2]. As a result, repeated computation of QRD under microsecond-scale deadlines can become a dominant system-level bottleneck.

1.1.1 Realification: Complex-to-real transformation and scaling impact.

In the particular case of DBF, QRD is performed on the complex-valued (CV) covariance matrix $\mathbf{C} \in \mathbb{C}^{N_R \times N_R}$, where N_R denotes the number of receiving antennas, formed after channel estimation. In order to avoid native complex arithmetic in the hardware, the complex covariance matrix is converted to an equivalent real form as shown below:

$$\mathbf{C} \in \mathbb{C}^{N_R \times N_R} \quad \longrightarrow \quad \mathbf{C_r} \in \mathbb{R}^{(2N_R)\times(2N_R)}. \tag{1.2}$$

In this thesis, we use the term *realification* to describe the process of transforming a complex matrix into a real-valued matrix. The resulting matrix is called *realified* matrix. This transformation has two significant consequences:

- **Memory storage growth:** the complex covariance matrix has N_R^2 complex entries. If stored as separate real and imaginary parts, this corresponds to $2N_R^2$ real scalars. *Realification* produces a $(2N_R) \times (2N_R)$ real matrix, i.e., $(2N_R)^2 = 4N_R^2$ real scalars. Therefore, *realification* increases scalar storage by $2\times$ compared to storing $\Re\{\mathbf{C}\}$ and $\Im\{\mathbf{C}\}$ separately.
- **Compute growth:** the covariance dimension scales with the antenna count as $O(N_R)$. Since QRD scales as $O(N^3)$ for an $N \times N$ matrix, operating on the *realified* matrix increases the dimension-driven operation count to $O((2N_R)^3) = O(8N_R^3)$, an $8\times$ increase in computations.

Therefore, for a covariance matrix whose dimension scales with the antenna count (N_R), the combined effect of *realification* overhead, cubic computational complexity and tight real-time budgets, makes QRD a key bottleneck when scaling DBF solutions. To make DBF affordable in next-generation systems, this computational burden must be mitigated while maintaining practical area/resource utilization.

1.2 Problem Statement

For this thesis work, the following constraints are derived from strict system-level requirements for BeammWave™'s digital beamforming accelerator (DBFA) and are outlined below:

- The number of receiving antennas in the system is 8 ($N_R = 8$). Effectively, perform QRD on an 8 $\times$8 complex-valued covariance matrix represented in 16-bit fixed-point precision. In the proposed hardware implementation, this implies QRD of a 16×16 real-valued (RV) matrix.
- Time budget per QRD is given as $T_{\text{QRD\_eff}} \leq 50\ \mu s$

1.2.1 Research Question

This thesis addresses the following research question:

- How can a QR decomposition accelerator be architected to meet the latency, throughput, and resource constraints for an 8×8 fixed-point complex-valued covariance matrices in 5G NR digital beamforming?

1.3 Contributions of the thesis

The main contributions of this thesis are:

- A schedule-driven QRD accelerator architecture for a 16×16 real-valued matrix.
- RTL implementation with memory subsystem and performance counters.
- Evaluation of latency, throughput, and resource utilization on a ZCU216 FPGA development board.

- Discussions on scalability towards larger matrices and Application-Specific Integrated Circuit (ASIC) implementations.
- **Patent Filing:** The "Schedule-Driven" architecture and the memory-visibility barrier mechanism proposed in this thesis have been filed for patent protection.

1.4 Delimitations

This thesis is strictly delimited to the design of a hardware accelerator tailored for the specific linear algebra requirements of 5G digital beamforming. Consequently, the architecture is optimized exclusively for square covariance matrices ($N \times N$); support for general rectangular matrices ($M \neq N$) is excluded. Furthermore, the hardware datapath is delimited to real-valued arithmetic. Processing of complex-valued matrices is supported solely through a pre-processing "realification" transformation. To meet the sub-50 μs latency constraint without incurring the area penalty of floating-point units, the numerical representation is restricted to 16-bit fixed-point `Q1.15` format. Finally, the algorithmic scope is limited to Givens rotation-based QRD [12, Sec. 5.2.5] implemented via CORDIC. Alternative decomposition methods such as Householder Transformation (HH) and Gram-Schmidt process (GS) [12, Sec. 5.2] are not explored in this thesis.

1.5 Thesis Organization

The remainder of this thesis is structured as follows. Chapter 2 presents background and motivation, including fundamentals of the QRD algorithm, complexity analysis, and a survey of state-of-the-art implementations. We end the chapter by identifying the gap in the state-of-the-art. Chapter 3 details the proposed accelerator architecture with an emphasis on scheduling and system constraints. Chapter 4 describes the microarchitecture of the implementation and derives the execution model for the accelerator. Chapter 5 covers verification methodology and FPGA validation. Chapter 6 presents results, discussions, conclusions, and future work.

Chapter 2

Background and Motivations

This chapter provides the technical background and motivation for the proposed QRD accelerator. First, we introduce Givens rotation-based QR decomposition and the CORDIC mapping used in hardware. Next, we quantify the computational complexity and identify where the dominant workload arises in a QRD iteration. Finally, we review state-of-the-art software, hybrid, and hardware approaches, and derive the application-specific gap and system-level requirements that motivate a custom accelerator for the targeted fixed-point use case.

2.1 QRD algorithm

2.1.1 Givens-based QR Decomposition

Givens rotation (GR) is a numerically stable method for QRD that is well-suited to hardware implementations due to its localized two-row updates. The Givens rotation algorithm eliminates sub-diagonal entries of a matrix by applying a sequence of orthogonal plane rotations [12]. Given a real matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$, the goal is to compute an upper triangular matrix $\mathbf{R}$ and an orthogonal matrix $\mathbf{Q}$ such that

$$\mathbf{A} = \mathbf{QR} \tag{2.1}$$

For each element $a_{i,k}$ located below the diagonal, a 2 $\times$ 2 Givens rotation is constructed to zero it out. The rotation matrix is defined as

$$\mathbf{G_k}(i,j) = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}, \tag{2.2}$$

where

$\mathbf{G_k}(i,j)$ specifies the involved rows $(i,j); i \neq j$ and the column $\mathbf{k}$ where a zero will be inserted,

$c = \cos\theta$ and $s = \sin\theta$ and, are chosen such that

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a_{i,k} \\ a_{j,k} \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}, \quad r = \sqrt{a_{i,k}^2 + a_{j,k}^2}. \tag{2.3}$$

The values of c and s can be directly derived as

$$c = \frac{a_{i,k}}{\sqrt{a_{i,k}^2 + a_{j,k}^2}}, \tag{2.4}$$

$$s = \frac{a_{j,k}}{\sqrt{a_{i,k}^2 + a_{j,k}^2}}. \tag{2.5}$$

Once (c, s) are obtained, the rows i and j of the matrix $\mathbf{R}$ are updated as

$$r'_{i,n} = c\, r_{i,n} + s\, r_{j,n} \tag{2.6}$$

$$r'_{j,n} = c\, r_{j,n} - s\, r_{i,n}, \quad \forall n \geq \mathbf{k}. \tag{2.7}$$

Successive application of Givens rotations leads to the formation of $\mathbf{R}$.

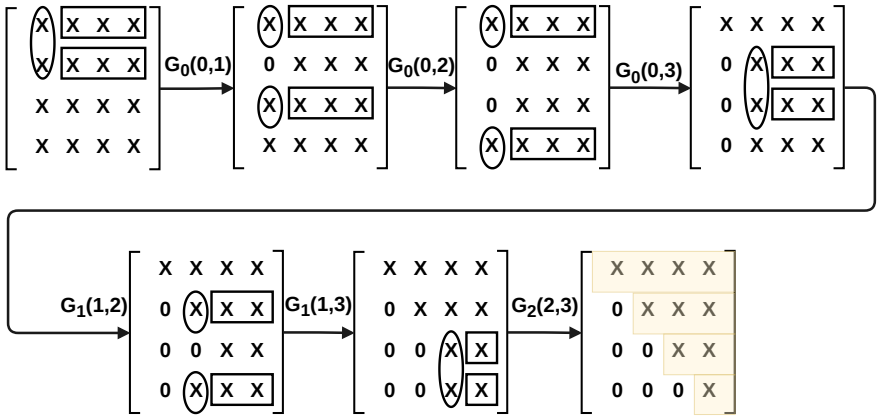


Figure 2.1: Usual Givens rotation schedule for 4×4 matrices

Figure 2.1 shows the process of application of Givens rotations to a 4×4 matrix. The circular areas indicate the elements selected to compute the rotation matrix, $\mathbf{G_k(i, j)}$ using equations (2.3), (2.4) and (2.5). The application of Givens rotation also results in the rotation of the elements within the rectangle. Thus, a Givens rotation affects only a row pair at a time.

To obtain the orthogonal matrix $\mathbf{Q}$, the same sequence of Givens rotations is applied to an identity matrix. This means that the overall transformation can be expressed as

$$\mathbf{Q} = \mathbf{G_1 G_2} \ldots \mathbf{G_m} \tag{2.8}$$

where $\{\mathbf{G_k}\}$ are the individual plane rotations used during the factorization of $\mathbf{A}$. Appendix B provides the pseudocode for Givens rotation based QRD for generalised $N \times N$ matrices.

2.1.2 Givens rotation using CORDIC-based Processing Element

A direct computation of (c, s) involves square-root and division operations, which are expensive in hardware. Fortunately, Coordinate Rotation Digital Computer

(CORDIC) algorithm exists which computes (c, s) using iterative shift-and-add operations. It supports three coordinate systems- circular, linear, and hyperbolic, of which the circular case is directly applicable to QRD using Givens rotations [13].

The basic iterative equations of CORDIC in circular coordinates are given by

$$x_{i+1} = x_i - d_i\, y_i\, 2^{-i}, \tag{2.9}$$

$$y_{i+1} = y_i + d_i\, x_i\, 2^{-i}, \tag{2.10}$$

$$z_{i+1} = z_i - d_i\, \arctan(2^{-i}), \tag{2.11}$$

where

x_i, y_i are the vector components after the i-th iteration,

z_i is the residual angle after the i-th iteration,

$d_i \in \{+1, -1\}$ is the direction of rotation.

Each iteration applies a micro-rotation of $\pm\arctan(2^{-i})$, implemented via a conditional add/subtract and binary shift.

The CORDIC algorithm operates in two modes: *vectoring* and *rotation*. In *vectoring* mode, the input vector (x_0, y_0) is iteratively rotated toward the x-axis. After n iterations, $y_n \approx 0$ and the accumulated angle satisfies $z_n \approx \arctan(y_0/x_0)$. At each iteration, the rotation direction d_i is selected based on the sign of the current y_i. In this thesis, vectoring mode is used in the Givens step to estimate the rotation angle θ and thereby obtain the rotation parameters $c = \cos(\theta)$ and $s = \sin(\theta)$.

In *rotation* mode, the input vector (x_0, y_0) is rotated by a prescribed angle z_0. After n iterations, the output satisfies $(x_n, y_n) \approx \mathbf{R}(\theta)\,(x_0, y_0)$ with $\theta = z_0$. Here, the direction d_i is chosen based on the sign of the residual angle z_i.

The CORDIC algorithm is not a perfect rotation and it introduces a magnitude gain. The scaling factor arises because each CORDIC micro-rotation replaces a true orthonormal rotation matrix with a shift-and-add update (by setting $\tan(\gamma_i) = \pm 2^{-i}$), which introduces a deterministic magnitude change per iteration [14]. After n iterations, the vector magnitude is scaled by a constant gain factor

$$K_n = \prod_{i=0}^{n-1} \sqrt{1 + 2^{-2i}}, \tag{2.12}$$

which can be precomputed. The gain compensation can be applied once at the end of the CORDIC pipeline or absorbed into surrounding stages, depending on the chosen fixed-point scaling strategy.

Finally, a Givens rotation maps naturally to both CORDIC modes: vectoring mode is used to compute the rotation angle (and hence c and s) from the pivot/target pair, and rotation mode is then used to apply the same rotation to the remaining elements of the affected rows.

2.2 Complex-to-Real Transformation (*Realification*)

The QR decomposition algorithm in Section 2.1.1 is introduced for real-valued matrices. However, the input covariance matrix is complex-valued. Therefore,

this thesis adopts an equivalent *real-valued* formulation by *realifying* the complex matrix.

Let

$$\mathbf{A}_c \in \mathbb{C}^{N_c \times N_c}, \qquad \mathbf{A}_c = \mathbf{A}_R + j\mathbf{A}_I,$$

where $\mathbf{A}_R = \Re\{\mathbf{A}_c\}$ and $\mathbf{A}_I = \Im\{\mathbf{A}_c\}$ are real matrices $\in \mathbb{R}^{N_c \times N_c}$.

The *realified* matrix is defined as

$$\mathbf{A}_r \in \mathbb{R}^{N \times N}, \qquad N = 2N_c$$

$$\mathbf{A}_r = \begin{bmatrix} \mathbf{A}_R & -\mathbf{A}_I \\ \mathbf{A}_I & \mathbf{A}_R \end{bmatrix} \tag{2.13}$$

Therefore, QR decomposition can be applied to $\mathbf{A}_r$ using standard real-valued Givens rotations, while preserving the underlying complex linear mapping. Appendix A details the *realification* of an 8×8 CV covariance matrix to its equivalent RV representation.

2.3 Complexity Analysis

The objective of this section is to identify the dominant work items within a GR-based QRD iteration and consequently motivate the acceleration of such work items under a strict time budget.

The computational cost of QR decomposition scales cubically with matrix dimension:

$$\text{Cost} = \mathcal{O}(N^3). \tag{2.14}$$

where N is the dimension of a square matrix.

This scaling arises because every sub-diagonal element must be annihilated by applying successive Givens rotations. Each rotation also updates a trailing row segment of the matrix. For a square matrix of dimension N,

The total number of Givens rotations required is

$$V(N) = \frac{N(N-1)}{2} \tag{2.15}$$

As each rotation also updates trailing elements in the affected rows, and the number of rotations is given by

$$A_R(N) = \frac{(N-1)N(2N-1)}{6} \tag{2.16}$$

The same rotation sequences must be applied to accumulate the orthogonal matrix $\mathbf{Q}$, giving

$$A_Q(N) = \frac{N^2(N-1)}{2} \tag{2.17}$$

Therefore, the total number of element-level updates per QRD iteration is

$$A_{\text{tot}}(N) = V(N) + A_R(N) + A_Q(N) \tag{2.18}$$

If each update is realized through a CORDIC datapath with $CORDIC\_ITER$ micro-rotations, an approximate compute effort can be expressed as

$$C_{\text{QRD}}(N) = A_{\text{tot}}(N) \times CORDIC\_ITER \tag{2.19}$$

Work item	**Formula**	$N = 8$	$N = 16$	$N = 32$
Givens rotations	$V(N)$	28 (7.2%)	120 (3.7%)	496 (1.9%)
Updates for **R**	$A_R(N)$	140 (35.7%)	1240 (37.8%)	10416 (38.9%)
Updates for **Q**	$A_Q(N)$	224 (57.1%)	1920 (58.5%)	15872 (59.2%)
Total updates	$A_{\text{tot}}(N)$	**392 (100%)**	**3280 (100%)**	**26784 (100%)**

Table 2.1: Operation counts for one QRD iteration for $N \in \{8, 16, 32\}$.

Table 2.1 summarizes operation counts for $N = 8$, $N = 16$ and $N = 32$. The results highlight two key observations. First, the computational cost per QRD is dominated by trailing-row updates accounting for $> 92\%$ of the total operations. Second, the workload grows cubically as the matrix dimension doubles.

As the runtime is expected to be dominated by row updates, the above findings motivate the exploration of parallelization of these operations. In the following sections, we review the state-of-the-art implementation and motivate why the targeted use case requires a custom design.

2.4 Literature Survey

The GR-based QRD is one of the most widely used matrix factorization methods. Due to its high computational cost, extensive prior work exists that optimizes the implementation of the underlying algorithm to achieve faster execution and higher throughput. Prior work is grouped into three categories - software solutions, hybrid solutions, and custom hardware accelerator.

2.4.1 Software Solutions

A large body of optimized software exists for QRD and dense linear algebra. On x86 platforms, Intel's Math Kernel Library (MKL) [15] and LAPACK-based

ecosystems (LAPACK/CLAPACK/ScaLAPACK/PLASMA) [16] provide highly tuned kernels for multicore CPUs. On embedded Arm™ platforms, CMSIS-DSP [17] offers signal-processing oriented building blocks. For RISC-V embedded platforms, PULP-DSP [18, 19] targets the Parallel Ultra Low Power (PULP) ecosystem. In QR-PULP [20], the authors demonstrate the use of loop unrolling and memory-layout tuning to optimize QRD on the GAP-9 [21] platform.

These libraries are effective on their intended platforms. However, software execution remains sensitive to control-flow dependencies and memory behavior. For the beamforming use case of this thesis, where QRD is executed repeatedly under a tight time budget, guaranteeing microsecond-scale latency is difficult without dedicated hardware support or multicore processors.

2.4.2 Hybrid Approaches

To overcome CPU limitations, heterogeneous systems offload QRD workloads to accelerators such as Graphics Processing Units (GPU), Field-Programmable Gate Array (FPGA), or Coarse-Grained Reconfigurable Architectures (CGRAs). A common theme in these approaches is to restructure QRD using blocking/tiling and then schedule the resulting tasks across multiple compute engines to increase parallelism and utilization.

Rákossy et al. [22] propose a column-wise Givens scheme that eliminates multiple elements per column to expose parallelism in the GR algorithm. Agullo et al. [23] reformulate QRD into a tile-based task graph and schedule it across multicore CPUs and multiple GPUs. Merchant et al. [24] use algorithm–architecture co-design for HH QRD on the REDEFINE CGRAs [25]. Rodriguez Borbon et al. [26] target tall-and-skinny matrices on FPGAs using a parallel-blocked HH QRD.

These approaches are most effective for large matrices, where the overhead of task scheduling, buffering, and data movement becomes small compared to the compute work. For the small fixed-point matrix case in this thesis, this overhead is harder to justify under tight area, time, and power budgets.

2.4.3 Custom Hardware Accelerators

Custom accelerators can achieve predictable latency and high energy efficiency by using deep pipelining and spatial parallelism. Among QRD algorithms, GR is especially hardware-friendly because it operates on two rows at a time, which exposes parallelism at multiple levels. GR also maps naturally to systolic structures [27–30], making it a common choice for FPGA/ASIC implementations.

Wang [31] proposed a 12 $\times$ 12 IEEE 754 single-precision QRD using a two-dimensional systolic array. Munoz and Javier [32] extended this idea to a 4 $\times$ 4 16-bit fixed-point matrix using CORDIC-based Processing Element (PE)s. Langhammer et al. [33] presented a high-performance floating-point design based on Modified Gram–Schmidt (MGS) algorithm, optimized for medium and large matrices. Xu and Leeser [34] proposed a broader FPGA-based matrix processor that includes QRD, and demonstrated fixed-point configurations targeting 8 $\times$ 8 and

16 $\times$ 16 matrix dimensions. These works primarily focus on optimizing PE architectures to achieve higher throughput.

Another line of work focuses on architectural modularity and cross-layer optimization. Vishnoi et al. [35] propose a family of modular QRD accelerator architectures based on CORDIC-mapped Givens rotations, where the elimination order, array organization, and PE design are co-optimized to improve area and energy efficiency. Their work highlights that accelerator performance depends not only on datapath throughput, but also on mapping the dependency structure to the array while keeping PE utilization high.

On the ASIC side, Chen and Qiu [36] proposed a low-latency group-sorted QRD for a 16 $\times$ 16 fixed-point complex matrix in 65 nm CMOS, achieving up to 513 MHz of clock frequency. Attari et al. [37] presented an SVD accelerator in GF-22nm FD-SOI for 16 $\times$ 16 IEEE 754 single-precision matrices, which can be repurposed for QRD.

Overall, prior work on hardware spans a wide range of matrix sizes, arithmetic formats, and implementation targets. For the small fixed-point beamforming case, the limiting factor is often not only the compute datapath speed, but also how eliminations are scheduled, and the structure of the underlying hardware. This motivates discussing the common hardware structures and the scheduling constraints, explicitly focusing on GR.

2.5 Existing Hardware Structures

Hardware accelerators for QRD are commonly organized as either:

- **Triangular 2D array structures**: These map eliminations to a 2D array of PEs, maximizing throughput and minimizing latency at higher area cost, or
- **Linear 1D array structures**: These map eliminations to a 1D array of PEs, often prioritizing regularity, lower area, and easier scaling. 1D array implementations often focus on optimizing the scheduling of updates to minimize latency.

In both cases, end-to-end performance is dictated by the elimination schedule, dependency depth, and the memory traffic needed to support the targeted throughput of the implementations.

2.6 Identified Gap

Although the literature demonstrates a wide range of QRD accelerators, most designs target larger matrix sizes, floating-point arithmetic, or platform-level compute infrastructures. In contrast, the BeammWave™ use case focuses on performing QRD on an 8 $\times$ 8 complex-valued covariance matrix represented in 16-bit fixed-point precision, executed repeatedly under strict real-time and area constraints, necessitating building of a custom accelerator.

2.6.1 System-level Requirements

Beyond computational complexity, the accelerator design is governed by strict real-time requirements originating from the beamforming use case. Table 2.2 summarizes the high-level parameters of the target system.

sl.no.	Parameter name	Value
1	Sub Carrier Spacings (SCS)	120 kHz
2	System Bandwidth (BW)	800 MHz
3	Sampling Rate	983.04 Msps
4	FFT Size	8192
5	Number of Physical Resource Blocks (PRB)	66
6	Number of sub-carriers	792
7	Number of transmitting antennas (N_T)	1
8	Number of receiving antennas (N_R)	8
9	Input Matrix Dimension	8×8
10	Bitwidth	16 Fixed-point
11	Time window to calculate steering vectors	1 ms

Table 2.2: High level requirements: DBFA.

For this thesis, the following derived timing requirement is used. The accelerator must complete beamforming-relevant processing within a hard wall-time limit of 1 ms. To account for system overhead, a conservative slack of 50% is reserved, leaving an effective compute window of $T_{\mathrm{QRD}} = 500\ \mu s$. Within this budget, at least $I \geq 10$ QRD iterations are required, yielding the per-iteration bound:

$$\left.\begin{aligned} T_{\mathrm{QRD\_eff}} &\leq \frac{T_{\mathrm{QRD}}}{I} \end{aligned}\right\} \text{general case} \tag{2.20}$$

$$\left.\begin{aligned} T_{\mathrm{QRD\_eff}} &\leq 50\,\mu s \end{aligned}\right\} \text{for } I \geq 10 \text{ and } T_{\mathrm{QRD}} = 500\,\mu s \tag{2.21}$$

This hard-timing-bound motivates the development of a custom hardware accelerator for QRD. The accelerator must be both fast and have a small area footprint with realistic memory traffic.

Therefore, there exists a clear need for a dedicated QRD accelerator tailored to the 8×8 fixed-point complex-valued matrix use case, which forms the focus of this thesis.

Chapter 3
Proposal

This chapter proposes the QRD accelerator architecture and motivates the design choices that shape the implementation. The central observation is that, for a Givens-rotation QRD accelerator, the elimination schedule is not merely an algorithmic detail. It determines the dependency depth, the exploitable parallelism, the required memory traffic, and the resulting area/resource utilization. Therefore, scheduling is motivated first, and the architecture is derived as a constrained mapping of the chosen schedule onto a practical datapath and memory system.

3.1 Design Requirements

The target application requires QRD of small 8×8 fixed-point covariance matrices originating from the digital beamforming chain. To avoid complex arithmetic inside the accelerator, the computation is implemented using a *realified* representation, resulting in an effective 16×16 real matrix.

The accelerator must also satisfy the strict real-time constraints per QRD iteration as derived in (2.21).

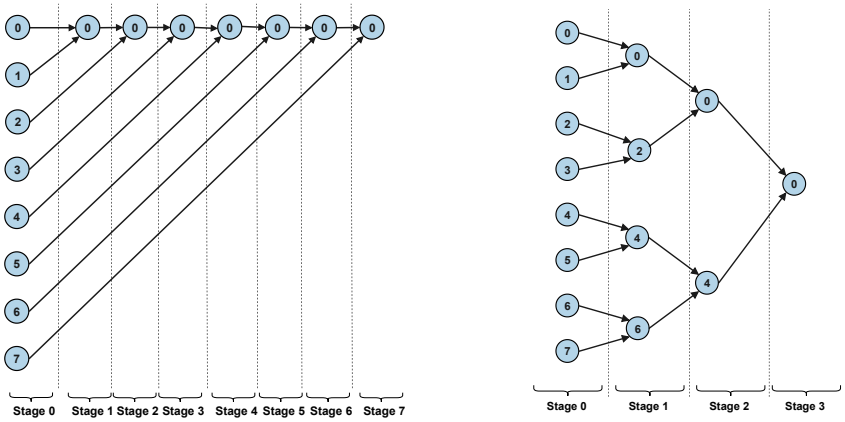
The architecture must target a clock frequency of 491.52 MHz as an initial goal, which is a sub-multiple of the sampling frequency for the system as shown in Table 2.2.

3.2 Algorithm-to-Hardware Mapping

Since each Givens rotation acts on only two rows, multiple levels of parallelism can be exposed if algorithmic dependencies between rotations are respected. Therefore, GR is well-suited for parallel hardware implementation.

In GR, a single elimination step consists of two distinct phases. First, generate rotation parameters from the pivot and target elements. Second, apply the rotation parameters to zero out the target element. The same rotation is also applied to the trailing-row elements.

In this work, both phases are implemented using CORDIC to avoid expensive division and square-root operations. This also results in a regular, deeply pipelined datapath which makes the compute datapath structurally capable of streaming. The achieved throughput is then determined by scheduling dependencies and by the memory behaviour.



(a) Fixed pivot dependency **(b)** Tree based dependency

Figure 3.1: Dependency graph for fixed-pivot and tree-based scheduling for QRD.

As referenced in 2.1.2, CORDIC is used in vectoring mode to derive the rotation control sequence needed to annihilate the selected target element. In rotation mode, the same control sequence is applied across trailing elements to update $\mathbf{Q}$ and $\mathbf{R}$. When CORDIC is implemented as fully-unrolled pipelined hardware, the resulting datapath has a fixed latency, L and can support a sustainable initiation interval, II at its streaming interface. In this thesis, the achieved II is ultimately bounded by the on-chip memory bandwidth and write-back policy.

3.3 Elimination schedule in Givens rotation

A key architectural decision in a Givens-rotation QRD accelerator is the order of elimination of sub-diagonal elements. The selection directly determines

1. the algorithmic dependency depth inside a pivot column.
2. how much parallelism is actually exploitable in hardware, and
3. memory traffic to be sustained by the memory system to support the parallelism.

To make this concrete, we first look at the classical Fixed-pivot (FP) elimination schedule. The main point is that FP scheduling creates a loop-carried dependency due to requiring the updated pivot value for successive elimination steps. This forces serialization within a column and ties the initiation interval to the datapath latency. This is captured in Section 3.3.1. To overcome the loop carried dependencies of FP, we introduce BT elimination schedule. The BT schedule reduces the dependency depth by grouping independent eliminations into stages, thereby allowing higher parallelism.

3.3.1 Why Fixed-Pivot Scheduling Forces $II = L$ within a Pivot Column

The key limitation in FP scheduling is simple - within a pivot column, every elimination depends on the *updated pivot value* produced by the previous elimination. Even if the datapath is fully pipelined, this loop-carried dependency forces the issue interval to match the pipeline latency plus memory overheads.

Consider pivot column j. Let $R^{(k)}$ denote the matrix state after k eliminations in this column. Also, let the current pivot and target entries be defined respectively as

$$r_k \coloneqq \mathbf{R}^{(k)}[j, j] \tag{3.1}$$

$$x_k \coloneqq \mathbf{R}^{(k)}[i_k, j]; \qquad i_k \in \{j+1, \ldots, N-1\} \tag{3.2}$$

The k-th Givens rotation parameters are computed from the *current* pivot-target pair:

$$(c_k, s_k) = \mathbf{G}(r_k, x_k) \tag{3.3}$$

and the pivot-column update produces the next pivot value:

$$\begin{bmatrix} r_{k+1} \\ 0 \end{bmatrix} = \begin{bmatrix} c_k & s_k \\ -s_k & c_k \end{bmatrix} \begin{bmatrix} r_k \\ x_k \end{bmatrix}. \tag{3.4}$$

This creates a dependency chain $r_{k+1} \rightarrow (c_{k+1}, s_{k+1})$ for the next elimination. Therefore, the next elimination must compute new parameters using the updated pivot value r_{k+1}, yielding a true dependence chain $r_{k+1} \rightarrow (c_{k+1}, s_{k+1})$. Figure 3.1a shows this dependency chain as a graph.

Let $t_{j,k}$ denote the cycle index at which the k-th elimination in pivot column j is *issued* into the compute datapath (i.e., when the pivot/target pair for elimination k is accepted for parameter-generation). Let L denote the fixed pipeline latency (in cycles) from this issue event to the time the updated pivot value r_{k+1} becomes available. Since elimination $(k+1)$ requires r_{k+1} to compute the rotation parameters (c_{k+1}, s_{k+1}), it cannot be issued before r_{k+1} is available. This results in a scheduling constraint defined as:

$$t_{j,k+1} \geq t_{j,k} + L. \tag{3.5}$$

From (3.5), it follows that the minimum spacing between two consecutive eliminations within the same pivot column is L cycles at best. Using the standard definition $II = t_{k+1} - t_k$ for a steady stream of dependent operations, the fixed-pivot schedule forces the following constraint within a pivot column:

$$II_{\text{fixed-pivot}} = L \tag{3.6}$$

Equation (3.6) imposes a lower bound on the II for FP scheduling and does not consider memory overheads.

3.3.2 Cycle Count Versus Matrix Dimension for Fixed-Pivot Scheduling

For an $N \times N$ matrix, pivot column j requires elimination of $(N - j - 1)$ subdiagonal entries. Let $t_{j,k}$ denote the cycle at which the k-th elimination in pivot

column j is *issued* into the datapath, where $k \in \{0, \ldots, N-j-2\}$. From (3.6), the next elimination in the same pivot column must satisfy (3.5). Therefore, the cycle cost per pivot column scales as

$$C_j \approx (N-j-1)\,L \tag{3.7}$$

and summing over $j = 0, \ldots, N-2$ gives the total cycle count for one QRD iteration:

$$\begin{aligned} C_{\mathrm{QRD}}(N) &\approx \sum_{j=0}^{N-2} (N-j-1)\,L \\ &= L \sum_{m=1}^{N-1} m = L\frac{N(N-1)}{2}. \end{aligned} \tag{3.8}$$

Equation (3.8) isolates the scheduling-driven serialization that forces $II = L$ in the FP scheduling. In practice, the latency that limits progress is not only the compute pipeline latency, but also the time required for the updated pivot to become usable by the next elimination (e.g., memory read latency, writeback draining, and stage-visibility constraints). These effects can be captured by replacing L with an effective latency $\mathrm{L_{eff}}$, yielding

$$C_{\mathrm{QRD}}(N) \approx L_{\mathrm{eff}}\frac{N(N-1)}{2}. \tag{3.9}$$

This linear scaling in cycle count, along with quadratic scaling of the number of computations, is the main reason to explore schedules that reduce dependency depth in Section 3.3.3.

3.3.3 Reducing dependency depth : Binary-tree based scheduling(BT)

The algorithmic dependency limitations imposed by the FP scheduling on exploiting the parallelism inherent in GR make it necessary to explore a BT-based schedule. A BT schedule avoids the long dependency chain by changing the dependency structure itself. Figure 3.1b shows the dependency graph generated by BT scheduling. This scheduling offers higher levels of parallelism by breaking the elimination into dependent *stages*. Within a stage, there is no dependence and therefore the eliminations can run in parallel.

Consider the same pivot column j, for an $N \times N$ matrix of Section 3.3.2 with an active column segment length $M = N - j$ i.e. M rows. In fixed-pivot scheduling, the eliminations form a single chain of length $(M-1)$ (Figure 3.1a), so the dependency depth is $\mathcal{O}(M)$. In a binary-tree schedule, eliminations are grouped such that:

- in stage s, independent row-pairs are eliminated in parallel (no intra-stage edges),
- each next stage $s+1$, combines the remaining rows, reducing the active set by roughly a factor of two.

As a result, the number of stages needed to reduce M rows to one updated pivot row is:

$$\text{stages}_{\text{tree}}(j) = \lceil \log_2(N-j) \rceil = \lceil \log_2 M \rceil \,. \tag{3.10}$$

This reduces the dependency depth in the pivot column from $\mathcal{O}(M)$ in FP scheduling to $\mathcal{O}(\log_2 M)$, while preserving the total number of eliminations $(M-1)$. The key point is that the tree schedule does not remove work; it reduces *how long the longest dependence chain is*. This exposes higher levels of parallelism as many eliminations become eligible for issue before the updated pivot value becomes the bottleneck.

3.4 Mapping Scheduling to Hardware Architecture

The BT-based scheduling reduces dependency depth (Section 3.3.3) and exposes parallelism, especially in the early stages where up to $\lfloor M/2 \rfloor$ row-pairs can be independent. In theory, this parallelism can be used to *increase throughput* (issue more row-pairs per cycle) and to *reduce latency* (finish a stage using multiple lanes). In practice, however, both are limited by system-level constraints as below:

1. **Triangular utilization.** The workload profile of QRD is inherently triangular and cannot be avoided. As j increases, $M = N - j$ decreases and the available parallelism $\lfloor M/2 \rfloor$ diminishes. In a BT-based schedule, the early stages can contain many disjoint row-pairs, but reducing latency using this parallelism requires a datapath with multiple parallel lanes (up to $\lfloor M/2 \rfloor$ in the best case). This is not scalable as the same lanes will become increasingly underutilized in later stages and later pivot columns, while the total hardware cost grows rapidly with N. This leads to significant degradation in hardware efficiency as the matrix dimension increases.

2. **Memory bandwidth requirements.** A row-pair elimination step consumes two row reads (pivot and target) and produces two row writes (updated pivot and target). Therefore, increasing the number of parallel lanes increases both read and write demand. Designing the memory system for peak early-stage parallelism results in an over-designed system. This is particularly true for QRD workloads with triangular work profiles, where the available parallelism drops, and the extra bandwidth cannot be used efficiently.

3. **Inter-stage synchronization.** Even with a fully parallel datapath, the eliminations in stage $s{+}1$ are only legal once the updated rows from stage s are visible according to the chosen memory model. This is due to true data dependencies between consecutive stages and can result in a Read-after-Write (RAW) hazard. If visibility is enforced through a hard commit barrier, then the start of the next stage depends not only on compute completion, but also on draining writeback and completing memory commits. This means stage-to-stage throughput becomes a function of writeback/commit behavior in addition to the compute latency.

3.4.1 Chosen Architecture: 1D Pipelined Compute With Explicit Memory Commit

As elucidated in section 3.4, the peak parallelism offered by BT scheduling requires a full parallel datapath. This imparts unrealistic demands on area/resources, memory bandwidth, with very little justification on achieved speedup. Therefore, in this thesis, we propose an accelerator design around a 1D array of fully pipelined datapath that streams row-pairs at a fixed II. The on-chip memory system is dimensioned to provide the required steady-state row read/write rates without over-provision for early peak parallelism.

To keep handling of inter-stage dependencies simple and deterministic, we propose a design which uses an explicit write-back policy as a hard barrier, outlined below:

1. All rows updated in stage s are always written back to memory, and
2. Stage s+1 begins only after all write-back of stage s completes, i.e., updated rows become visible in memory.

The above policy eliminates RAW hazards without complex forwarding or scoreboarding logic, at the cost of an inter-stage write-back cycles penalty.

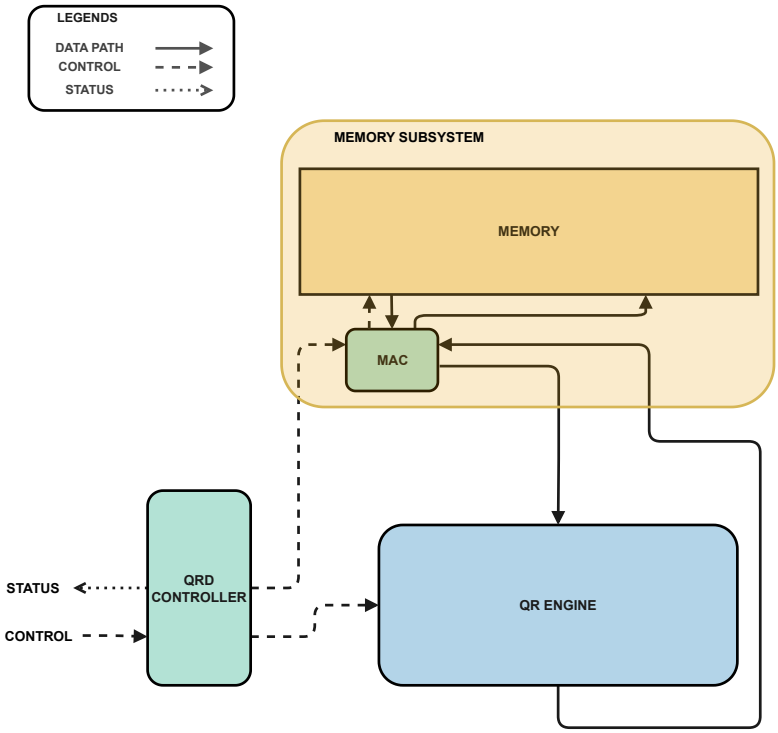


Figure 3.2: Proposed QRD accelerator architecture .

3.5 Proposed Accelerator Architecture

Based on the discussions in the preceding sections 3.1–3.4, we propose an accelerator architecture as shown in Figure 3.2. This architecture allows for higher throughput, lower latency, lower resource/area utilizations, and the compute hardware scales linearly with matrix dimension. The proposed accelerator design is organized around three main blocks:

1. **QR Engine.** A fully pipelined 1D array of CORDIC-based PEs that supports rotation parameter generation and row updates in parallel, providing a fixed compute latency L and a streaming interface with initiation interval II.
2. **QRD Controller.** A schedule-driven controller that issues row-pairs according to the BT schedule. It also enforces the inter-stage RAW hazard handling policy as outlined in 3.4.1.
3. **Memory subsystem.** A memory with independent One Read One Write (1R1W) ports for **Q** and **R** matrices, accessed through a Memory Access Controller (MAC). The MAC handles row reads and writebacks and uses

staging buffers to decouple memory traffic from the compute pipeline while preserving in-order row fetches and write-back requests.

Chapter 4
Implementation

This chapter describes the Register-Transfer Level (RTL) implementation of our proposed QRD accelerator. We keep the focus of our implementation on the following defined constraints:

1. **Functional Correctness:** The hardware must maintain bit-exact equivalence with the golden reference C model.
2. **Execution Latency:** The end-to-end QRD latency is targeted at ≤ 50 μs per iteration.
3. **Area Efficiency:** Optimize PE utilization to minimize area footprint while meeting the latency target.

Section 4.1 gives a system overview and the execution model of our proposed accelerator. Section 4.2 specifies the fixed-point numerical formats used in the proposed implementation. Section 4.3 details the stage scheduling logic and the inter-stage-barrier policy. Section 4.4 details the building blocks of the accelerator, analyzes the memory-bandwidth constraints and derives the pipeline timing. Section 4.6 presents a stage-level timing model of the end-to-end execution. Section 4.7 reports the synthesis results and resulting FPGA resource utilization.

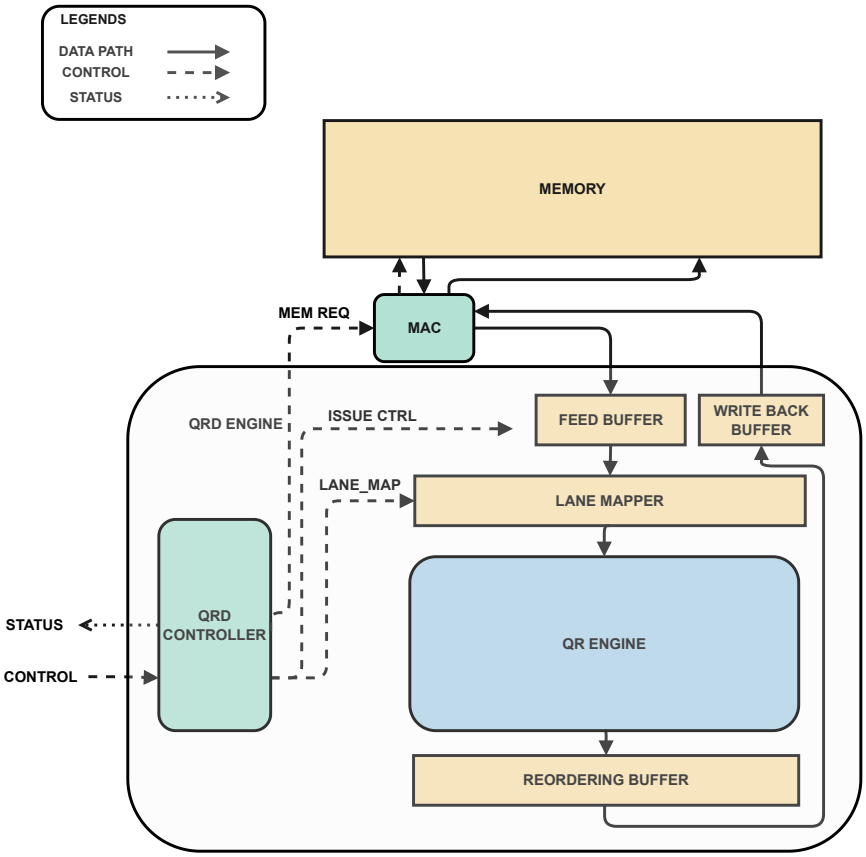


Figure 4.1: Top-level architecture of the QRD accelerator subsystem.

4.1 System Overview and Execution Model

Figure 4.1 shows an overview of the implemented accelerator. The accelerator performs QRD of an $N \times N$ *real-valued* matrix stored in on-chip memory. In this thesis, $N = 16$ and hence the accelerator can operate on a matrix with a maximum dimension of 16×16.

4.1.1 Execution Model

The accelerator is designed to be a *fire-and-forget* subsystem. A host can configure the control registers in the accelerator, load the initial realified matrix $\mathbf{A}$ into memory and issue a `start` signal. The accelerator accepts the job and, upon completion of QRD, writes $\mathbf{Q}$ and $\mathbf{R}$ matrices back to on-chip memory. It then asserts a sticky `done` signal, which can be captured by the host as an indication of

Parameter	Value (this implementation)
Matrix dimension (realified)	$N = 16$
CORDIC micro-iterations	12
Compute latency	$L = 16$
Sustained issue interval	$II = 2$
Memory ports	1R1W; Block 0:**A**/**R**, Block 1:**Q**
Datapath clock	$f_{\text{dp}} = 245.76$ MHz

Table 4.1: Implementation point used throughout Chapter 4.

job completion. This choice of execution model frees up the host for other tasks and improves the overall performance of the system.

4.1.2 Implementation constraints

Table 4.1 summarizes the supported matrix dimension, the chosen CORDIC micro-iterations, and the resulting datapath latency and sustained issue interval. These parameters affect timing, memory bandwidth requirements, and resource utilization for the implemented design.

All timing and bandwidth numbers in this chapter use the implemented datapath clock $f_{\text{dp}} = 245.76$ MHz. Though we set an initial target clock frequency of 491.52 MHz in Chapter 3, limitations imposed by the target platform forced timing closure at 245.76 MHz. Frequency scaling and its impact on accelerator performance are discussed in Chapter 6.

4.2 Numerical Representation and Matrix Layout

4.2.1 Fixed-Point Format and Storage

The software reference model has been developed such that each matrix element uses signed `Q1.15`. For simple addressing and simpler MAC design, this implementation stores each element as a sign-extended 32-bit word in memory, or 4 bytes/element. Internally, the datapath interprets the values as `Q1.15`.

4.2.2 Rounding, Saturation, and Overflow

Because of fixed-point math operations, the intermediate results have to be scaled back into `Q1.15`. This is achieved by:

1. **Rounding:** results are rounded when discarding fractional bits to reduce systematic truncation bias.

2. **Saturation:** overflow never wraps around; values are clamped to maximum/minimum range representable in `Q1.15`.

Also, as described in Section 2.1.2, the CORDIC datapath introduces a gain factor. This implementation compensates for the gain using a single aggregate scaling step applied at the end of the last micro-iteration.

4.2.3 Matrix Representation and Memory Layout

The accelerator operates on real matrices only. Therefore, the input complex-valued matrix is transformed into its real-valued counterpart. This real-valued matrix is stored in a row-major order in memory. Appendix A explains the conversion process in detail.

4.3 Scheduling and Inter-stage-barrier Policy

This section defines the Givens elimination schedule policy implemented by the controller. The accelerator processes the matrix *column-wise* from left to right. For each pivot column j, eliminations are scheduled using the BT schedule introduced in Section 3.3.3. The controller runs the schedule in *stages* and enforces a hard visibility barrier between consecutive stages. Therefore, the datapath behaves like a stream *within* a stage, but progress *across* stages is guarded by the inter-stage-barrier policy. As an example the BT-scheduled elimination steps for a 6×6 matrix are shown in Appendix C.

Stage definition For a pivot column j, a *stage* is one level of the binary-tree schedule. Stage s contains $P_{j,s}$ disjoint row-pairs that are independent *within that stage*. The computation of $P_{j,s}$ (and the stage structure) follows directly from the schedule definition in Section 3.3.3.

Memory visibility In this implementation, an updated row is considered *visible* only after it has been written back and committed to the on-chip memory. The next stage is not allowed to consume updated rows directly from internal buffers or from compute outputs. This enforces a full read–compute–writeback round-trip at every stage boundary.

4.3.1 Intra-stage issue and completion

Within a stage, the controller issues $P_{j,s}$ row-pairs in order as soon as the required rows are present in the feed buffer. Once issued, row-pairs stream through the datapath at the sustained initiation interval II and exit after a fixed latency L as defined in table 4.1.

A stage is marked complete only when *all* row-pairs issued in that stage have exited the compute pipeline, been written back through the writeback path, and been committed to memory via the MAC.

4.3.2 Inter-stage barrier and visibility

To handle inter-stage data dependencies, the controller enforces a hard inter-stage barrier as below:

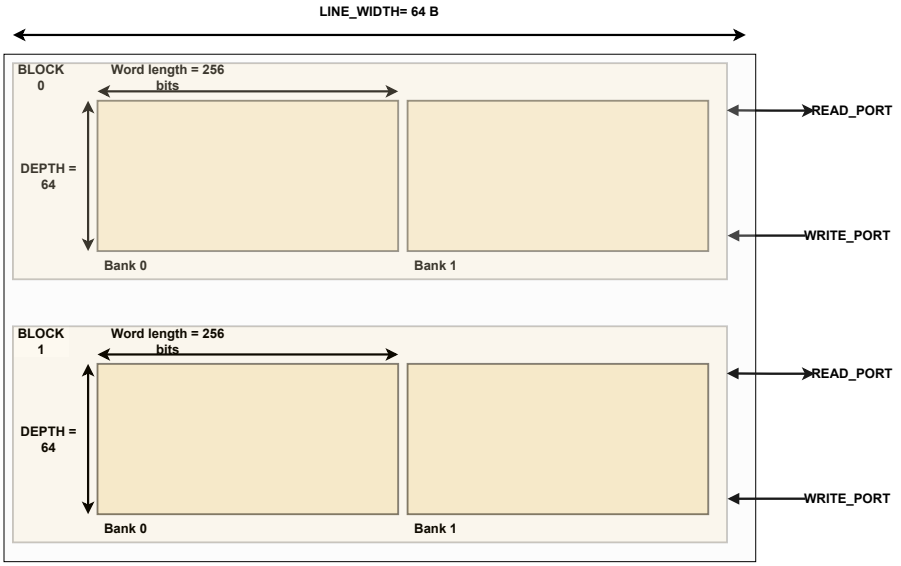


Figure 4.2: On-chip memory organization.

Stage $s{+}1$ starts only after all updates from stage s are committed and visible in memory.

This removes RAW hazards without forwarding or scoreboarding logic. The trade-off, however, is an inter-stage flush penalty, since progress to the next stage is gated by writeback and commit completion. This penalty is quantified by the stage-level timing model in Section 4.6 and is shown in figure 4.4.

4.4 Accelerator Elements

4.4.1 Memory System and Data Movement

This section describes how rows move through the accelerator and why the memory system sets the sustained throughput. In this implementation, a *row* is considered the minimum unit of data transfer. This means that the memory system is organised to provide a full row upon a read request and accept a full row when a write request is made.

On-Chip Memory Organization: Figure 4.2 shows the organization of the on-chip memory. The memory is constructed out of two independent 1R1W blocks, where Block 0 stores the initial input matrix **A** and the updated **R** matrix and Block 1 stores the **Q** matrix. Such an organization avoids **R**/**Q** contention and allows for parallel access.

Memory Line Width and Bandwidth-Limited II: As defined in 4.2.1, each matrix element is stored as 4 bytes in memory. For a realified matrix of

dimension N, the row size then becomes :

$$S_{\text{row}} = N \cdot 4 \text{ bytes.} \tag{4.1}$$

For a memory line width W_{line} (bytes/beat), transferring a full row takes:

$$B_{\text{row}} = \left\lceil \frac{S_{\text{row}}}{W_{\text{line}}} \right\rceil = \left\lceil \frac{N \cdot 4}{W_{\text{line}}} \right\rceil \tag{4.2}$$

beats per row.

With the full round-trip policy, each issued row-pair updates two rows. This applies to both **R** and **Q**. Therefore, *per memory block*, one issued row-pair requires: two row reads (pivot and target) and two row writes (updated pivot and updated target). This corresponds to $2B_{\text{row}}$ read beats and $2B_{\text{row}}$ write beats per row-pair, per block.

A 1R1W block sustains one read beat/cycle and one write beat/cycle. Therefore, the sustained initiation interval must satisfy the following:

$$\frac{2B_{\text{row}}}{II} \leq 1 \;\Rightarrow\; II \geq 2B_{\text{row}}. \tag{4.3}$$

In this implementation, $N = 16$ and we choose $W_{\text{line}} = 64$ bytes, $B_{\text{row}} = 1$. Therefore, the sustained interval is $II = 2$.

At $f_{\text{dp}} = 245.76$ MHz, a 64-byte line corresponds to:

$$BW_{\text{port}} = W_{\text{line}} \cdot f_{\text{dp}} \approx 15.73 \text{ GB/s} \tag{4.4}$$

per read port and per write port, per block. In the current memory layout, each 16-bit value is stored in a 32-bit word. Therefore, only half of the physical line bandwidth carries useful payload data, so the effective payload bandwidth is ≈ 7.86 GB/s. Even with this packing overhead, the bandwidth demand remains reasonable for on-chip memories. Thus, equation (4.3) sets the lower bound on achievable II for the implemented accelerator.

4.4.2 Memory Access Controller

The MAC enforces an in-order, row-based service model. Read requests fetch row lines into the feed buffer, while write requests commit updated row lines from the writeback path back into memory. When all write requests of the current stage have been serviced, the updated rows become visible for the next stage.

4.4.3 Decoupling Buffers

The implementation decouples the streaming compute core from the 1R1W memory through various decoupling buffers. The Feed Buffer (FB) holds pivot and target rows until issued by the controller. The Re-Order Buffer (ROB) reassembles lane-wise output from the compute datapath into row-line order. The Write Back Buffer (WBB) holds full row lines until the MAC commits them.

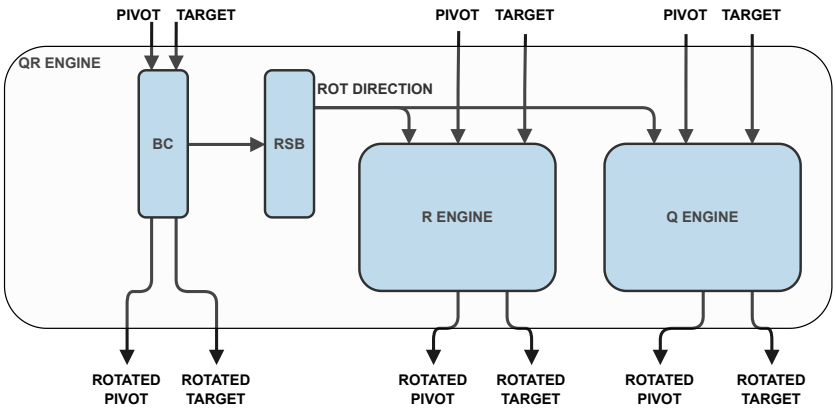


Figure 4.3: QR engine datapath.

4.4.4 Compute Path: QR Engine

The QR Engine (QR ENGINE) applies GR on an issued row-pair and is the main compute unit of the accelerator. Figure 4.3 shows the micro-architecture of the QR ENGINE. The QR ENGINE employs two types of CORDIC-based PEs to perform the required rotation on the rows in parallel. The Boundary Cell (BC) runs CORDIC in vectoring mode and derives the rotation control sequence for the pivot, target pair. The Rotation Sequence Broadcaster (RSB) aligns and broadcasts these sequences to the R ENGINE (R Engine) and Q ENGINE (Q Engine). Both R Engine and Q Engine each deploy $N = 16$ parallel rotation lanes. These rotation lanes run CORDIC in rotation mode to update the trailing elements, thereby updating $\mathbf{R}$ and $\mathbf{Q}$ in parallel. As a result of the chosen structure, the QR ENGINE has a fixed compute latency of L.

4.4.5 Pipeline Timing: Latency and Sustained II

This implementation uses fully unrolled and fully pipelined CORDIC-based PEs in the QR ENGINE. Each lane implements a fixed 12 CORDIC micro-iterations (Table 4.1). Another 4 cycles are added to account for alignment and CORDIC scaling compensation. Therefore, the total QR ENGINE latency is given by:

$$L = 12 + 4 = 16 \text{ cycles.}$$

Structurally, the compute core can accept $II = 1$. However, as derived in equation (4.3) in Section 4.4.1, the achievable II is 2.

4.5 Theoretical Performance Bounds

This section establishes optimistic lower bounds for the proposed architecture. The purpose is to separate limits imposed by the QR ENGINE pipeline from limits imposed by the memory system and the execution contract.

4.5.1 Compute-only lower bound

We first derive a *compute-only* bound. It assumes that row-pairs can be issued into the QR ENGINE without stalls and that stage transitions have no drain or commit overhead. This bound therefore excludes effects that dominate the implemented design, such as MAC writeback/commit service, full read-compute-writeback round-trips, and the hard inter-stage visibility barrier. These effects are discussed in Chapter 6.

For a matrix of dimension $N \times N$, consider a pivot column index $j \in \{0, \ldots, N-1\}$ with

$$m_j = N - j \tag{4.5}$$

active rows. The stage-wise BT schedule reduces m_j rows to one row over

$$S_j = \lceil \log_2(m_j) \rceil \tag{4.6}$$

stages. Let $P_{j,s}$ denote the number of row-pairs processed in stage s. Under the *compute-only* assumptions, stage (j, s) completes after

$$C_{j,s} = \begin{cases} 0, & P_{j,s} = 0 \\ L + (P_{j,s} - 1)\, II, & P_{j,s} \geq 1 \end{cases} \tag{4.7}$$

where L is the fixed QR ENGINE latency in cycles and II is the initiation interval in cycles/row-pair. The full-sweep *compute-only* bound is then given by

$$C_{\text{total}}^{(\text{comp})} = \sum_{j=0}^{N-1} \sum_{s=0}^{S_j - 1} C_{j,s} \tag{4.8}$$

From Table 4.1, $N = 16$ and $L = 16$ cycles and equation (4.8) evaluates to $C_{\text{total}}^{(\text{comp})} = 855$ cycles, when $II = 1$. This is the *compute-only* bound enforced by the streaming QR ENGINE structure and assumes memory can supply the required row-pairs per cycle.

Any real implementation that enforces visibility/commit and performs write-back through the MAC must satisfy

$$C_{\text{measured}} \;\geq\; C_{\text{total}}^{(\text{comp})} \tag{4.9}$$

Therefore, the gap $C_{\text{measured}} - C_{\text{total}}^{(\text{comp})}$ quantifies system-level overhead introduced by the chosen execution contract.

4.5.2 Memory bound throughput

Even if the QR ENGINE is fully pipelined, the sustained throughput is bounded by the on-chip memory port constraints. Equation (4.2) shows that throughput scaling is limited by line width. Increasing compute resources (e.g., more row-pair lanes) does not translate to proportional end-to-end speedup unless the memory system and the visibility/commit policy can supply and retire row-pairs at the

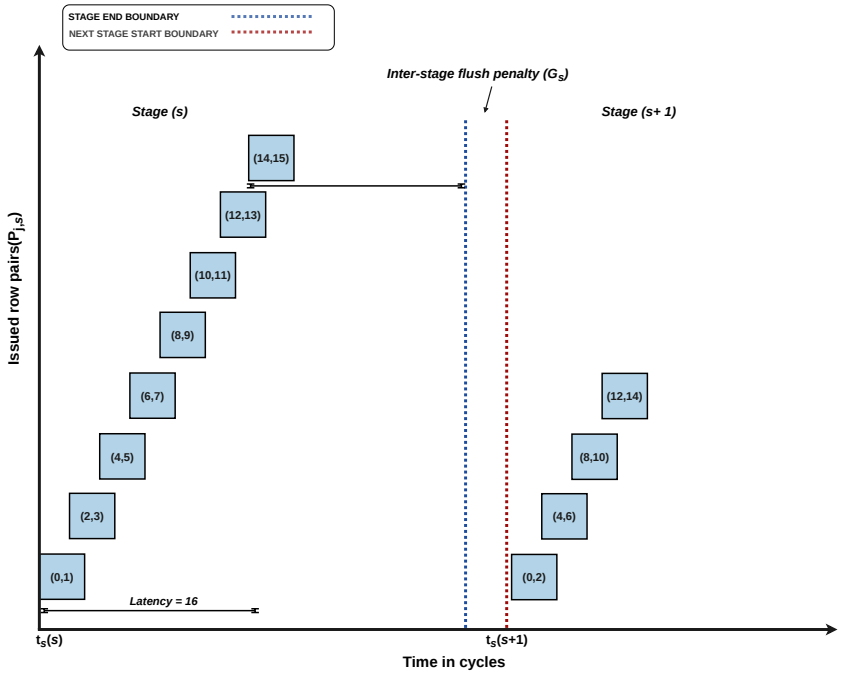


Figure 4.4: Stage timeline under the hard visibility barrier between consecutive stages.

higher rate. As derived in (4.3), the implemented 1R1W memory system limits the II to 2 when $W_{\text{line}} = 64$ bytes.

Therefore, the *compute-only* lower bound relevant for this implementation is given by evaluating equation (4.8) with $II = 2$, which results in $C_{\text{total}}^{(\text{comp})} = 926$ cycles.

4.6 Stage-Level Timing Model

This section states a compact timing model of execution of two stages under the inter-stage-barrier policy outlined in Section 4.3. The full derivation and a worked example are provided in Appendix D.

Let $t_s(s)$ denote the start cycle of stage s and $P_{j,s}$ denote the number of row-pairs issued in the stage. As outlined in 4.4.5, the datapath has a fixed latency, L and equation 4.3 sets the II for this thesis work. Therefore, the last compute output of the stage, s exits at cycle $t_{\text{out,last}}$, which is given by:

$$t_{\text{out,last}}(s) = t_s(s) + (P_{j,s} - 1)\, II + L \tag{4.10}$$

From section 4.3.2, the next stage can start only after the *last* updated rows from stage s are committed to memory. The cycle at which the last row is committed to memory is modelled as $t_{\text{commit,last}}$ and is given by:

$$t_{\text{commit,last}}(s) = t_s(s)+L+\delta+\max\Big((P_{j,s}-1)\,II,\ (2P_{j,s}-1)\,II_w\Big)+(W-1) \quad (4.11)$$

where,

- δ is the fixed offset in cycles from the first QR ENGINE output to the first row-line write-start event,
- II_w is the sustained write acceptance interval (cycles/row-line) at the MAC write interface, and
- W is the commit-to-visibility latency in cycles from write-start/accept at the MAC to the point where the updated row becomes visible in memory.

These parameters are tied to the writeback path and the behaviour of the MAC as described in Section 4.4.2. Now, the earliest start cycle of the next stage is given by:

$$t_s(s+1) \geq t_{\text{commit,last}}(s) + 1. \quad (4.12)$$

If we define the *inter-stage flush penalty*, G_s, as the difference between compute completion of stage s and the next legal start cycle for stage $s+1$:

$$G_s \triangleq t_s(s+1) - t_{\text{out,last}}(s). \quad (4.13)$$

Substituting (4.10)–(4.12) gives:

$$G_s = \delta + W + \max\Big((2P_{j,s}-1)\,II_w - (P_{j,s}-1)\,II,\ 0\Big). \quad (4.14)$$

In this implementation, we minimize the synchronization overhead by setting $W = 1$ and $\delta = 1$. These represent near-ideal synchronization overhead. However, Equation (4.14) reveals that the total flush penalty G_s remains sensitive to the throughput mismatch between the compute initiation interval (II) and the write-back acceptance rate (II_w). Because $P_{j,s}$ decreases with each successive stage due to the triangular workload, G_s creates a dynamic "pipeline bubble" that grows in relative significance as the matrix is reduced. For $N = 16$, the implemented design still meets the latency budget comfortably, but the pipeline bubble contributes a measurable overhead that becomes more relevant as N scales.

4.7 FPGA Implementation Summary

Table 4.2 reports the post-implementation resource utilization on the ZCU216 FPGA development board. The reported numbers include the QR engine datapath, the controller, and the memory system.

Name	Utilization	Available	% Utilization
LUT	56720	425280	13.34
FF	65755	850560	7.73
DSP	132	4272	3.09
BRAM	30	1080	2.78

Table 4.2: Post-Implementation Resource utilization on ZCU216.

The utilization profile is consistent with the architecture choices -

- **DSPs:** Predominantly consumed by the N parallel rotation lanes within the QR ENGINE as outlined in Section 4.4.4.
- **BRAM:** Primarily allocated to the independent memory blocks, as well as the row-level staging buffers necessitated by the hard visibility and commit policies.
- **LUTs/FFs:** Largely attributed to the stage-sequencing control logic, the rotation sequence broadcast (RSB) network. The high LUT and FF count also includes the shift-and-add logic inherent to the unrolled CORDIC iterations.

A key observation is that for $N = 16$, the design is not area-constrained as utilization $< 15\%$. Instead, the design is memory-bound as established by equation (4.3). Simply adding more parallel PEs will not reduce the initiation interval below the $II = 2$ and, therefore will not decrease the measured latency of the accelerator. Detailed throughput analysis and post-implementation timing results are reported in Chapter 6.

Chapter 5
Verification

This chapter describes how correctness is established for the implemented QRD accelerator. The verification is reference-model driven: the implemented RTL is checked for bit-exactness against a fixed-point golden reference C model. The C model uses the same `Q1.15` arithmetic and the same realified representation as the RTL. Therefore, the target of this verification is deterministic equivalence at the memory boundary, not "numerically close" behavior. We conducted the verification in two phases:

- Simulation-based verification phase, which uses a SystemVerilog (SV) test-bench and performs bit-exact checking of the produced **Q** and **R** matrices against the golden reference model.
- FPGA Validation - A hardware-in-the-loop phase that extends bit-exact checking to the target device. Beyond numerical correctness, this phase utilizes embedded hardware cycle counters to capture execution behavior of the accelerator.

5.1 FPGA Validation Setup and Test Flow

The FPGA validation environment is organized into three parts:

- reference generation on the software side,
- the host/test platform on ZCU216 (PS-side infrastructure), and
- the Design Under Test (DUT), i.e., the QRD accelerator as a subsystem in PL.

Figure 5.1 shows the high-level arrangement.

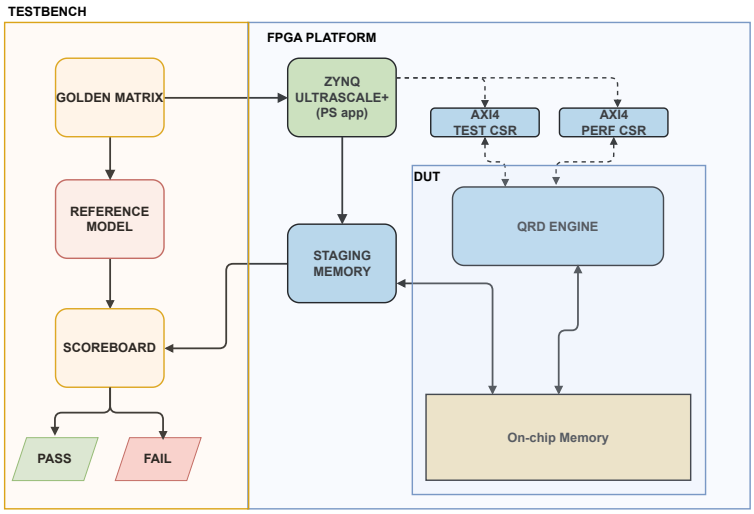


Figure 5.1: FPGA validation setup: reference generation, test platform, and DUT.

5.1.1 Reference Generation (Software Side)

A Python-based test driver generates seeded random input matrices. For each input, it executes the fixed-point C reference model and stores the expected $\mathbf{Q}$ and $\mathbf{R}$ outputs. The expected results are written to disk as a binary file to keep the checking policy deterministic and repeatable.

5.1.2 Test Platform (ZCU216 PS-Side Infrastructure)

The host-side processor system orchestrates accelerator runs and acts as the bridge between software and the on-board memory system. The host writes the input matrix into a staging memory and then triggers accelerator execution. After completion, the host initiates transfer of the produced $\mathbf{Q}$ and $\mathbf{R}$ matrices from the accelerator on-chip memory into the staging memory. The staging memory is then read back, and the retrieved contents are sent to the software side for comparison.

5.1.3 DUT (QRD Accelerator Subsystem)

The DUT is the QRD accelerator implemented as a subsystem in PL. It reads the input matrix from its on-chip memory, executes QR decomposition under the implemented schedule and commit policy, and writes back $\mathbf{Q}$ and $\mathbf{R}$ to the same memory. The accelerator exposes control and performance counters through memory-mapped Configuration and Status Registers (CSR) registers. These registers are used to configure execution parameters, start a run, detect completion, and collect cycle counts across test runs.

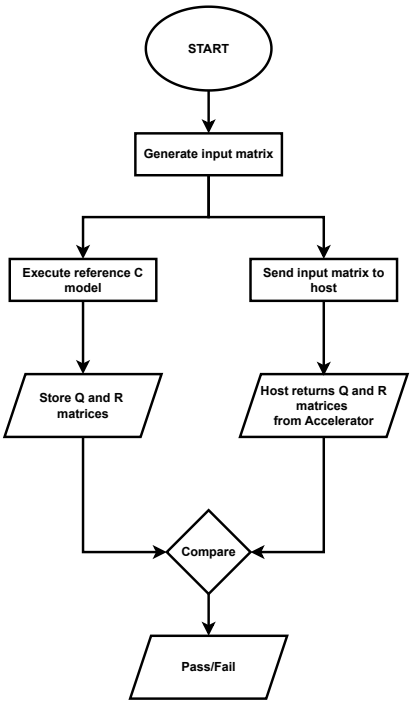


Figure 5.2: Validation Test flow

5.1.4 End-to-End Test Flow

Figure 5.2 shows the test flow for a single validation run. Each run follows the same end-to-end flow as described below:

1. **Generate input matrix:** A Python script generates a seeded random matrix.
2. **Execute reference model:** The fixed-point C model runs with the generated random matrix as input.
3. **Store golden outputs:** The reference **Q** and **R** matrices are stored as binary files.
4. **Send input to host:** The input matrix is sent to the Zynq host, which configures the accelerator and enables execution.
5. **Host returns DUT outputs:** After completion, the host reads back the produced **Q** and **R** matrices from the accelerator memory.
6. **Compare:** The returned matrices are compared element-wise against the stored golden reference.

7. **Pass/Fail:** The run is marked as a pass if all elements match; otherwise fail.

In addition, the reported CSR counters are logged per run to track end-to-end cycles and per-column elimination cycles. These measurements are later used to cross-check against the theoretical performance bounds derived in Section 4.5.

Chapter 6
Results and Conclusions

This chapter evaluates the implemented QRD accelerator against the design requirements and summarizes the results. Results are reported for the thesis configuration ($N = 16$ realified) at $f_{\mathrm{dp}} = 245.76$ MHz. Cycle counts are obtained from the on-chip performance counters described in Chapter 5, and converted to time using f_{dp}. The main outcome is that the implemented system contract (tree scheduling, bandwidth-limited issue, and a hard inter-stage commit barrier) meets the real-time requirement with a substantial margin.

6.1 Target Versus Achieved

Table 6.1 summarizes the primary targets and measured outcomes for the thesis configuration.

Metric	Target	Achieved	Performance Margin
f_{dp} (MHz)	491.52	245.76	Limited by Timing closure
T_{QRD} (μs)	≤ 50	9.83	**Met** (5.1×)
Throughput (kQRD/s)	≥ 20	101.8	**Met** (5.1×)

Table 6.1: Target versus achieved performance results for N=16 (realified). Note that while the frequency target was not met, the latency and throughput targets were exceeded by over 5×.

The measured cycle count is 2415 cycles per QRD. At $f_{\text{dp}} = 245.76MHz$, this results in a latency of:

$$T_{\text{QRD}} = \frac{2415}{245.76 \times 10^6} \approx 9.83\ \mu s. \tag{6.1}$$

This corresponds to $9.83/50 \approx 19.7\%$ of the available latency budget of $\leq 50\ \mu s$.

6.2 Latency Analysis

To put the measured 2415 cycles in context, we compare the implemented design against the theoretical lower bounds derived in Section 4.5.2. The bound assumes sufficient lane parallelism and no writeback/commit-induced inter-stage gaps, meaning a new stage can start immediately when the previous stage finishes its computation.

The cycle breakdown is as follows:

- **Bandwidth-Limited Bound ($II = 2$):** 926 cycles.
- **Measured Implementation:** 2415 cycles.
- **Synchronization Overhead:** 1489 cycles.

The remaining gap of 1489 cycles is dominated by inter-stage commit gating—specifically, the MAC writeback service behavior under the hard inter-stage barrier.

It is worth noting the progress made during the design phase: an early implementation of the accelerator measured 3691 cycles per QRD. After redesigning the MAC, the measured cycle count was reduced to 2415 cycles, achieving $\approx 34\%$ reduction in latency.

6.2.1 Why data forwarding was not implemented

A straightforward way to reduce the 1489-cycle inter-stage gap is to add data forwarding across stages. This would allow stage $s+1$ to consume rows produced by stage s before they are committed to memory. Conceptually, this relaxes the

hard visibility barrier and overlaps writeback with compute, which can lead to reduced end-to-end latency.

In this thesis, forwarding is not implemented. The current design already meets the real-time requirement with margin, and the hard-barrier contract keeps the controller and memory system simple and deterministic while keeping a single source and destination of data: the on-chip memory.

In contrast, forwarding would introduce multiple sources of data: on-chip memory and bypass buffers. In order to maintain the correctness of eliminations, additional control would be required. Specifically, a forwarding-capable design would require tracking row-pairs across stages (especially when the number of rows is odd), bypass logic to route the surviving rows for the next stage of computations, and redesigning the controller to reflect the new rules of stage completion. Practically, this also increases buffering demand, such as requiring deeper feed buffers to hold surviving row-lines across stages. Additionally, the row lines must also be tagged to allow control of issue to the QR ENGINE.

A quantified trade-off between the added control and buffering complexity and the achieved latency reduction is left for future work.

6.3 Throughput

The measured runtime is $T_{\text{QRD}} \approx 9.83\ \mu s$ (Eq. (6.1)). Therefore, the sustained throughput is:

$$\text{Throughput} \approx \frac{1}{T_{\text{QRD}}} \approx \frac{1}{9.83\ \mu s} \approx 101.8\ \text{kQRD/s}. \tag{6.2}$$

The achieved throughput is set by the same system contract as in Chapter 4: a bandwidth-limited issue rate ($II = 2$) and a hard inter-stage commit barrier.

The results from (6.2) and Table 6.1 demonstrate that the accelerator comfortably exceeds the target requirement. It is also important to note that the reported throughput of 101.8 kQRD/s is a serialized throughput, calculated directly as $1/T_{\text{QRD}}$. This represents a "single-job" execution model where a new QRD iteration only begins after the previous one has completely cleared the last inter-stage commit barrier. There remains a significant opportunity to further increase this throughput through two primary architectural enhancements:

1. **Inter-job Pipelining:** By relaxing the hard visibility barrier and allowing the next matrix to enter the first stage while the current matrix is in the final stages, the "dead time" between iterations can be eliminated.

2. **Frequency Optimization:** As the design currently meets all latency targets at 245.76 MHz, closing the timing gap toward the original 491.52 MHz target would effectively double the throughput.

6.4 Throughput sensitivity to Matrix Dimension and Memory Line Width

As derived in Section 4.4.1 (Equation (4.3)), the sustained issue interval is bandwidth-limited by the 1R1W memory system under the full round-trip policy. Because the required beats-per-row (B_{row}) depends on both the matrix dimension (N) and the memory line width (W_{line}), scaling the accelerator to larger matrices requires a proportional scaling of the memory interface to maintain throughput.

Table 6.2 summarizes the scaling for representative configurations. The normalized term highlights the throughput drop relative to the thesis configuration ($N = 16$, $W_{\text{line}} = 64$ B, $II_{\text{mem}} = 2$).

N	W_{line} (B/beat)	Beats per row line (B_{row})	Memory-bound (II_{mem})	Throughput (TP = $1/II_{\text{mem}}$)	Normalized throughput ($\text{TP}/\text{TP}_{\text{base}}$)
16	64	1	2	0.50	1.00
32	64	2	4	0.25	0.50
64	64	4	8	0.125	0.25
64	128	2	4	0.25	0.50
64	256	1	2	0.50	1.00

Table 6.2: Effect of memory line width on beats-per-row and the bandwidth-limited initiation interval...

For the intended maximum dimension $N = 64$, sustaining $II = 2$ under the same policy requires a row-sized line width of $W_{\text{line}} = 256$ bytes. Smaller line widths increase II_{mem}, reducing throughput even if the BT scheduled QR ENGINE provides independent row-pairs within a stage. This enforces the point that the throughput for the implemented architecture is sensitive to memory system design.

6.5 Timing Closure and Clock Frequency

As established in Section 4.7, the accelerator footprint is small (all resources $\leq$ 15%), meaning performance is not constrained by FPGA capacity. However, while the initial target frequency was 491.52 MHz, the implemented design achieved timing closure at 245.76 MHz.

To identify the maximum performance envelope of the current architecture, a timing exploration was conducted by sweeping the clock frequency. The design successfully closed timing at 300 MHz ($T_{clk} = 3.33$ ns), but failed at 350 MHz.

Analysis of the post-implementation timing report and Worst Negative Slack (WNS) identified two primary bottlenecks:

- **Logic-Bound Paths (DSP Slices)**: A critical path was located within the DSP48E2 slices used for the CORDIC scaling compensation logic. While these primitives are high-performance, the current implementation uses them

with minimal internal pipelining. Adding register stages within the DSP slices would break this path, allowing for frequencies closer to 491.52 MHz, but would increase the QR ENGINE latency L. The DSP48E2 slices are capable of having 4 pipeline stages. The implemented design uses 1 of those pipe stages, resulting in a latency of 16 cycles. Utilizing the remaining pipeline stages will mean the QR ENGINE latency would increase to 19. Such a trade-off will eventually be a net gain in the total execution time for the accelerator as the higher frequency will offset the minimal increase in latency.

- **Routing-Bound Paths (Broadcasting Network)**: A second critical path exists between the RSB and the inputs to the parallel rotation lanes. Because the lanes are physically distributed across the FPGA fabric, this path is dominated by long-wire routing delays. Since the architecture follows a rigid dataflow model, every additional register slice used to break these long paths would increase the total execution time T_{QRD} in terms of cycles. Future iterations could mitigate this by utilizing manual floorplanning or PBLOCK constraints to physically group the RSB and lanes.

For this thesis, the 300 MHz result demonstrates that the design has a 22% timing margin over the required 245.76 MHz frequency without requiring these additional latency penalties. This proves that the architecture is robust and capable of exceeding system requirements without requiring aggressive physical-layer optimizations(such as manual floorplanning etc.).

6.6 Conclusions

This thesis demonstrated a QRD accelerator where scheduling and memory visibility are treated as system-level design drivers rather than peripheral concerns. The implemented design successfully integrates:

1. a deeply pipelined CORDIC-based compute core with a deterministic latency of $L = 16$ cycles;
2. a bandwidth-limited initiation interval ($II = 2$) determined by the 1R1W on-chip memory subsystem; and
3. a hard inter-stage commit barrier that guarantees functional correctness by simplifying hazard management.

For the thesis configuration ($N = 16$ realified), the accelerator achieves a latency of 9.83 μs per QRD at 245.76 MHz, exceeding the target requirement by a 5.1$\times$ margin. The corresponding throughput of 101.8 kQRD/s is sufficient for the real-time demands of 5G mmWave digital beamforming systems.

A primary finding of this work is that end-to-end performance is dominated by the synchronization overhead and the memory service model (accounting for 61.7% of the cycle count) rather than raw arithmetic computation. This motivates a focus on data movement efficiency in future high-dimensional matrix scaling. Finally, the novel architecture and scheduling methodology developed in this work have been

filed for patent protection, acknowledging its contribution to the state-of-the-art in digital beamforming hardware.

6.7 Future Work

The measured results for the current implementation show that throughput and latency are primarily limited by system-level data movement: the memory-limited issue rate (II) and the hard inter-stage visibility barrier. Future work should therefore target these limiters first, before scaling the compute core.

- **Inter-stage forwarding** An extension of the current work could be to introduce a controlled forwarding path so that stage s+1 can consume updated rows without waiting for full writeback/commit. This will reduce the inter-stage gap in Eq. (4.14), at the cost of additional hazard management and bookkeeping resources.
- **Sustain low II for larger N.** Redesigning the memory subsystem to sustain multi-beat rows at low II (banking, wider ports, or multi-port SRAM). This will target the bandwidth limiter derived in Section 4.4.1 and potentially increase throughput if $II = 1$ is achieved for the datapath.
- **Timing closure towards** 491.52 **MHz.** Investigate the critical paths that prevented timing closure at the original target clock. Likely actions include manual floorplanning of synthesized logic.
- **Power and energy analysis.** Future work can also target measuring the power on the FPGA prototype and deriving energy per QRD. Since the current architecture is memory-movement dominated, an analysis of energy consumption between compute and memory traffic can lead to data-movement optimizations.
- **ASIC implementation study.** Another area any future work may focus on is to translate the architecture to ASIC context and re-evaluate the design trade-offs: longer logic depth vs register minimizations, SRAM porting/banking/double-pumping, clock frequency targets, fixed-point scaling choices, and the area/energy impact of adding forwarding or extra ports.
- **Support for rectangular matrices ($m \times n$).** The current implementation is optimized for a square matrix. Future work may extend the architecture to support rectangular matrices $m \neq n$.

Appendix A
CVD to RVD

This appendix explains how an 8×8 complex-valued covariance matrix is converted into a real-valued representation. The process results in a *realified* matrix upon which the QRD is performed.

A.1 Worked example: 8×8 complex matrix

Consider an 8×8 complex-valued matrix:

$$\mathbf{A_c} \in \mathbb{C}^{8\times8}, \qquad \mathbf{A_c} = \mathbf{A_R} + j\mathbf{A_I},$$

where

$$\mathbf{A_R} \in \mathbb{R}^{8\times8}, \quad \mathbf{A_I} \in \mathbb{R}^{8\times8}.$$

The realified matrix has dimension:

$$N = 2N_c = 16 \quad \Rightarrow \quad \mathbf{A_r} \in \mathbb{R}^{16\times16}.$$

Using (2.13), the 16×16 matrix is constructed as a 2×2 block matrix:

$$\mathbf{A_r} = \begin{bmatrix} \Re\{\mathbf{A_c}\} & -\Im\{\mathbf{A_c}\} \\ \Im\{\mathbf{A_c}\} & \Re\{\mathbf{A_c}\} \end{bmatrix} = \begin{bmatrix} \mathbf{A_R} & -\mathbf{A_I} \\ \mathbf{A_I} & \mathbf{A_R} \end{bmatrix}. \tag{A.1}$$

A.2 Hardware Interpretation

With the mapping as shown in (A.1), the QR engine can run on $\mathbf{A_r}$ using real arithmetic only. In this thesis, $\mathbf{A_r}$ is the matrix stored and processed by the accelerator, and the resulting $\mathbf{Q}$ and $\mathbf{R}$ are produced in the same realified layout.

Appendix B

QRD using Standard Givens rotation

This appendix presents the pseudocode for standard Givens rotation based QRD. The standard form requires square-root and division operations for each rotation operations.

Algorithm 1 QR Decomposition via Givens rotations

Require: $\mathbf{A} \in \mathbb{R}^{N \times N}$
Ensure: $\mathbf{Q}$ orthogonal, $\mathbf{R}$ upper triangular

$\mathbf{R} \leftarrow \mathbf{A}, \quad \mathbf{Q} \leftarrow \mathbf{I_N}$
for $k = 1$ to $N - 1$ **do** bottom-up per column
  for $i = N$ down to $k + 1$ **do**
    $x \leftarrow \mathbf{R}[i-1, k]$, $y \leftarrow \mathbf{R}[i, k]$, $r \leftarrow \sqrt{x^2 + y^2}$
    if $r \neq 0$ **then**
      $c \leftarrow x/r, \quad s \leftarrow y/r$ zero $\mathbf{R}[i,k]$ with $\mathbf{G} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$
      for $n = k$ to N **do** row-rotate $\mathbf{R}$: $[i-1, i] \leftarrow \mathbf{G}\,[i-1, i]$
        $t \leftarrow c\,\mathbf{R}[i-1, n] + s\,\mathbf{R}[i, n]$
        $\mathbf{R}[i, n] \leftarrow c\,\mathbf{R}[i, n] - s\,\mathbf{R}[i-1, n]$
        $\mathbf{R}[i-1, n] \leftarrow t$
      end for
      for $n = 1$ to N **do** column-rotate $\mathbf{Q}$: $[:, \{i-1, i\}] \leftarrow [:, \{i-1, i\}]\,\mathbf{G}^\top$
        $t \leftarrow c\,\mathbf{Q}[n, i-1] - s\,\mathbf{Q}[n, i]$ $\mathbf{G}^\top = \begin{bmatrix} c & -s \\ s & c \end{bmatrix}$
        $\mathbf{Q}[n, i] \leftarrow s\,\mathbf{Q}[n, i-1] + c\,\mathbf{Q}[n, i]$
        $\mathbf{Q}[n, i-1] \leftarrow t$
      end for
    end if
  end for
end for
return $(\mathbf{Q}, \mathbf{R})$

Appendix C

Binary tree scheduling of eliminations in Givens rotation

This appendix illustrates the execution of the Binary-Tree (BT) scheduling algorithm proposed in this thesis on a representative 6×6 square matrix. This visualization demonstrates how the dependency depth is reduced compared to a fixed-pivot schedule, as discussed in Section 3.3.

C.1 Notation and Setup

In the accompanying Figure C.1, the matrix is processed column by column to annihilate sub-diagonal elements.

- **X**: Represents a non-zero element.
- **0**: Represents an element that has been annihilated (zeroed out).
- $G_{j,s}$ (**pivot**, **target**): Denotes a set of Givens rotation applied for column j in stage s, utilizing the specified *pivot* row to eliminate the element in the *target* row.

C.2 Stage-wise Execution

The diagram tracks the transformation of the matrix **A** into the upper triangular matrix **R**. The parallelism is explicit: operations listed within the same stage can be executed simultaneously, provided sufficient hardware support is available.

C.2.1 Pivot Column $j = 0$

For the first column, the algorithm pairs adjacent rows to maximize parallelism:

- **Stage 1:** Three independent row-pairs are processed in parallel:

$$G_{1,0} : \{(0,1), (2,3), (4,5)\}$$

  This eliminates elements at indices $1, 3, 5$ simultaneously.

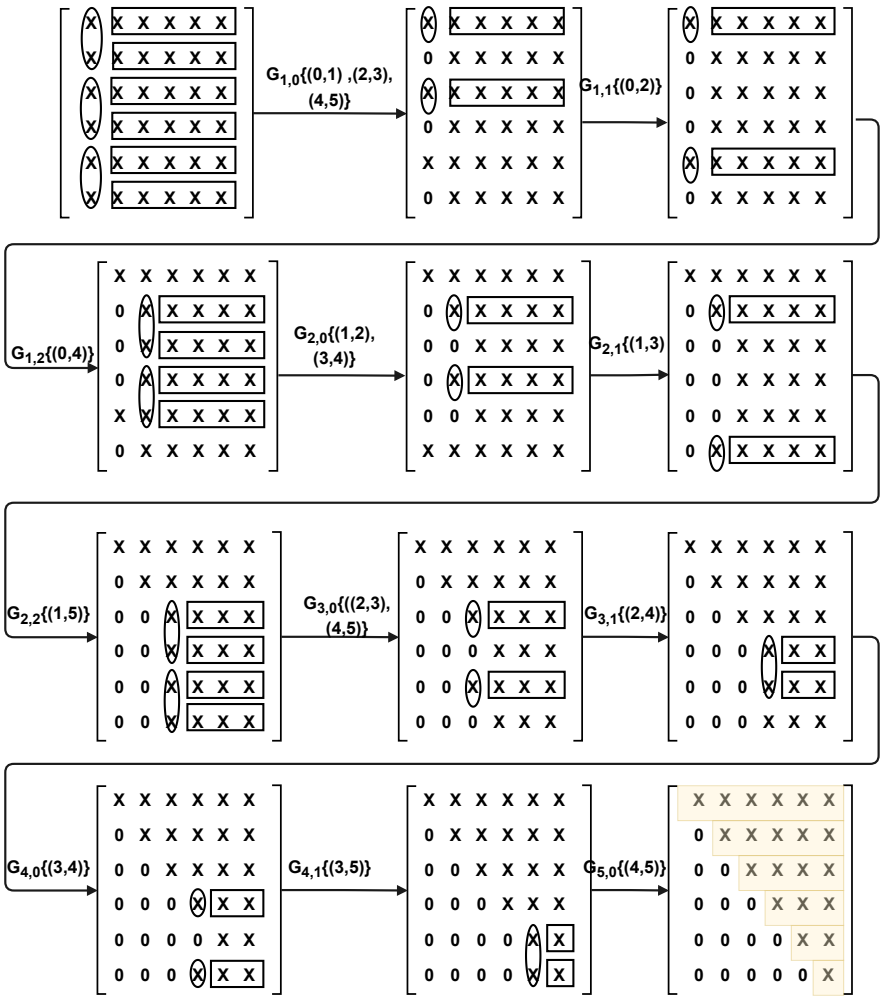


Figure C.1: A binary-tree scheduled Givens rotation for formation of $\mathbf{R}$

- **Stage 2:** The surviving rows from Stage 1 are paired. One row-pair is processed, as there is no target row for the surviving row index 4.

$$G_{1,1} : \{(0, 2)\}$$

- **Stage 3:** The final reduction for this column occurs, pairing the pivot row (0) with the remaining candidate row (4) to complete the column.

$$G_{1,2} : \{(0, 4)\}$$

This "tree" reduction reduces the dependency depth for the column from $\mathcal{O}(N)$ to $\mathcal{O}(\log_2 N)$.

C.3 Subsequent Columns

As the algorithm moves to Pivot Column $j = 1$ and beyond, the number of active rows decreases ($M = N - j$).

- The schedule continues to identify disjoint row-pairs among the remaining active rows.
- As M decreases, the available parallelism (number of pairs per stage) naturally diminishes, reflecting the "triangular" workload profile described in Section 3.4.

C.4 Conclusion

The final state shown at the bottom right of Figure C.1 is the upper triangular matrix $\mathbf{R}$, where all sub-diagonal entries are zero. This schedule confirms that multiple eliminations per stage are possible, validating our architectural choices for the datapath.

Appendix D

Derivation of Stage Timing

This appendix derives the stage timing model used in Section 4.6. The key point is the same as in Section 4.3: the datapath streams within a stage, but the next stage is gated by *memory visibility*. In this implementation, a row is visible only after it is committed to on-chip memory via the MAC.

D.1 Definitions and assumptions

We consider one stage s of a fixed pivot column j.

- t_s: start cycle of stage s (first legal issue time for this stage).
- P_s: number of disjoint row-pairs in stage s.
- II: sustained initiation interval for issuing row-pairs (cycles/row-pair).
- L: fixed compute latency from issue to the corresponding compute output (cycles).
- δ: fixed offset from the first compute output to the first write-start (ROB reassembly + WBB staging + MAC request setup).
- II_w: write acceptance interval of the commit path (cycles/row). This models how often the system can *start* a new row write transaction.
- W: commit latency from write-start to memory-committed/visible (cycles).

Within a stage, row-pairs are disjoint. Therefore, stage s updates $2P_s$ distinct rows for $\mathbf{R}$, and $2P_s$ distinct rows for $\mathbf{Q}$. Since the barrier requires both memories to be committed, we model commit time using the slowest effective write/commit path.

D.2 Compute exit time

The controller issues the k-th row-pair of stage s at:

$$t_{\text{iss}}(s,k) = t_s + k \cdot II, \qquad k \in \{0, 1, \ldots, P_s - 1\}. \tag{D.1}$$

With fixed compute latency L, the corresponding compute output exits at:

$$t_{\text{out}}(s,k) = t_{\text{iss}}(s,k) + L = t_s + k \cdot II + L. \tag{D.2}$$

Therefore, the last compute output of stage s exits at:

$$t_{\text{out,last}}(s) = t_{\text{out}}(s, P_s - 1) = t_s + (P_s - 1)\, II + L. \tag{D.3}$$

D.3 From compute outputs to committed rows

The hard barrier (Section 4.3) requires stage $s{+}1$ to wait until all updated rows from stage s are committed to memory (visible). We derive the time at which the *last* updated row becomes committed.

First write-start. The first compute output of the stage exits at $t_s + L$. The earliest write-start is modeled as:

$$t_{\text{w,first}}(s) = t_s + L + \delta. \tag{D.4}$$

Write-start spacing. Stage s produces $2P_s$ updated rows that must be written (per memory block). If the write path can start one row every II_w cycles, then the start time of the last row write is:

$$t_{\text{w,last}}(s) = t_{\text{w,first}}(s) + (2P_s - 1)\, II_w. \tag{D.5}$$

Row availability constraint. The write path cannot write data that has not been produced yet. The last row-pair output exits at $t_{\text{out,last}}(s)$, i.e., $(P_s - 1)\, II$ cycles after the first output. Therefore, even if the write path is fast, the last write-start cannot occur earlier than:

$$t_{\text{w,last}}(s) \geq t_{\text{w,first}}(s) + (P_s - 1)\, II. \tag{D.6}$$

Combining the two constraints (write acceptance rate and data availability), we model the last write-start as:

$$t_{\text{w,last}}(s) = t_{\text{w,first}}(s) + \max\Big((P_s - 1)\, II,\ (2P_s - 1)\, II_w\Big). \tag{D.7}$$

Commit (visibility) time. If W is the latency from write-start to memory-committed/visible, the last commit time is:

$$t_{\text{commit,last}}(s) = t_{\text{w,last}}(s) + (W - 1). \tag{D.8}$$

Substituting (D.4) and (D.7) into (D.8) gives:

$$t_{\text{commit,last}}(s) = t_s + L + \delta + \max\Big((P_s - 1)\, II,\ (2P_s - 1)\, II_w\Big) + (W - 1). \tag{D.9}$$

D.4 Hard barrier and inter-stage gap

The hard barrier enforces:

$$t_{s+1} = t_{\text{commit,last}}(s) + 1. \tag{D.10}$$

We define the inter-stage gap as:

$$G_s \triangleq t_{s+1} - t_{\text{out,last}}(s). \tag{D.11}$$

Using (D.3), (D.9), and (D.10):

$$\begin{aligned} G_s &= \Big(t_{\text{commit,last}}(s) + 1\Big) - \Big(t_s + (P_s - 1)\, II + L\Big) \\ &= \delta + W + \max\Big((P_s - 1)\, II,\ (2P_s - 1)\, II_w\Big) - (P_s - 1)\, II \\ &= \delta + W + \max\Big((2P_s - 1)\, II_w - (P_s - 1)\, II,\ 0\Big). \end{aligned} \tag{D.12}$$

This is the same gap term used in Section 4.6. It makes the architectural point explicit: even if compute finishes at $t_{\text{out,last}}(s)$, the next stage is gated by commit visibility.

D.5 Worked example: 16×16 real matrix, pivot column $j = 0$

For the realified implementation, $N = 16$. For pivot column $j = 0$, the active column segment length is:

$$M = N - j = 16.$$

Under the binary-tree schedule, the number of stages is:

$$\lceil \log_2 M \rceil = \lceil \log_2 16 \rceil = 4,$$

and the stage row-pair counts are:

$$(P_0, P_1, P_2, P_3) = (8, 4, 2, 1).$$

(Each stage halves the number of active rows; total eliminations remain $M - 1 = 15$.)

We use the validated implementation point:

$$II = 2, \qquad L = 16.$$

For the write/commit side, we keep the parameters explicit as (δ, II_w, W), and then show the common fast-commit case at the end.

Stage-wise expressions. From (D.3) and (D.9), the per-stage offsets are:

- Compute completion (relative to t_s):

$$t_{\text{out,last}}(s) - t_s = (P_s - 1)\, II + L.$$

- Last commit time (relative to t_s):

$$t_{\text{commit,last}}(s) - t_s = L + \delta + \max\Big((P_s - 1)\, II,\ (2P_s - 1)\, II_w\Big) + (W - 1).$$

- Next stage start:

$$t_{s+1} - t_s = \big(t_{\text{commit,last}}(s) - t_s\big) + 1.$$

Numeric compute exits. With $II = 2$ and $L = 16$:

$$t_{\text{out,last}}(0) - t_0 = (8 - 1) \cdot 2 + 16 = 30,$$

$$t_{\text{out,last}}(1) - t_1 = (4 - 1) \cdot 2 + 16 = 22,$$

$$t_{\text{out,last}}(2) - t_2 = (2 - 1) \cdot 2 + 16 = 18,$$

$$t_{\text{out,last}}(3) - t_3 = (1 - 1) \cdot 2 + 16 = 16.$$

In the implemented design:

$$II_w = 1, \qquad W = 1.$$

Then the gap term from (D.12) becomes:

$$G_s = \delta + 1 + \max\big((2P_s - 1) - (P_s - 1) \cdot 2,\ 0\big) = \delta + 2,$$

so each stage-to-stage gap is $\delta + 2$ cycles.

Therefore, the stage durations (from t_s to t_{s+1}) for pivot column $j = 0$ are:

$$t_1 - t_0 = (30) + (\delta + 2) = 32 + \delta,$$

$$t_2 - t_1 = (22) + (\delta + 2) = 24 + \delta,$$

$$t_3 - t_2 = (18) + (\delta + 2) = 20 + \delta,$$

$$t_4 - t_3 = (16) + (\delta + 2) = 18 + \delta.$$

Summing these gives the total cycles to complete pivot column $j = 0$ (four stages, including hard barriers between stages):

$$t_4 - t_0 = (32 + 24 + 20 + 18) + 4\delta = 94 + 4\delta.$$

This example shows the intended behavior: compute runs as a stream inside each stage, but the column-level progress is explicitly shaped by the per-stage commit/visibility overhead through δ, II_w, and W.

Bibliography

[1] Ericsson mobility report -june 2024. URL: https://www.ericsson.com/49ed78/assets/local/reports-papers/mobility-report/documents/2024/ericsson-mobility-report-june-2024.pdf.

[2] B.D. Van Veen and K.M. Buckley. Beamforming: a versatile approach to spatial filtering. *IEEE ASSP Magazine*, 5(2):4–24, 1988. doi:10.1109/53.665.

[3] Ivan A. Rumyancev and Alexander S. Korotkov. Survey on beamforming techniques and integrated circuits for 5g systems. In *2019 IEEE International Conference on Electrical Engineering and Photonics (EExPolytech)*, pages 76–80, 2019. doi:10.1109/EExPolytech.2019.8906842.

[4] Manish Sharma and Anand Agrawal. Comparative analysis of analog, digital, and hybrid beamforming techniques for enhanced mimo wireless communication systems: Radiation pattern and normalized power. In *2024 International Conference on Computational Intelligence and Network Systems (CINS)*, pages 1–7, 2024. doi:10.1109/CINS63881.2024.10864437.

[5] Aakash Arora, Christos G. Tsinos, Bhavani Shankar Mysore R, Symeon Chatzinotas, and Björn Ottersten. Analog beamforming with antenna selection for large-scale antenna arrays. In *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4795–4799, 2021. doi:10.1109/ICASSP39728.2021.9414673.

[6] Aarushi Dhami, Naitik N Parekh, and Yash Vasavada. Digital beamforming for antenna arrays. In *2019 IEEE Indian Conference on Antennas and Propogation (InCAP)*, pages 1–5, 2019. doi:10.1109/InCAP47789.2019.9134687.

[7] Jun Zhang, Xianghao Yu, and Khaled B. Letaief. Hybrid beamforming for 5g and beyond millimeter-wave systems: A holistic view. *IEEE Open Journal of the Communications Society*, 1:77–91, 2020. doi:10.1109/OJCOMS.2019.2959595.

[8] Shuangfeng Han, Chih-lin I, Zhikun Xu, and Corbett Rowell. Large-scale antenna systems with hybrid analog and digital beamforming for millimeter wave 5g. *IEEE Communications Magazine*, 53(1):186–194, 2015. doi:10.1109/MCOM.2015.7010533.

[9] Jeffrey G. Andrews, Stefano Buzzi, Wan Choi, Stephen V. Hanly, Angel Lozano, Anthony C. K. Soong, and Jianzhong Charlie Zhang. What will 5g be? *IEEE Journal on Selected Areas in Communications*, 32(6):1065–1082, 2014. doi:10.1109/JSAC.2014.2328098.

[10] Xinyu Gao, Linglong Dai, Shuangfeng Han, Chih-Lin I, and Robert W. Heath. Energy-efficient hybrid analog and digital precoding for mmwave mimo systems with large antenna arrays. *IEEE J.Sel. A. Commun.*, 34(4):998–1009, April 2016. doi:10.1109/JSAC.2016.2549418.

[11] Binqi Yang, Zhiqiang Yu, Ji Lan, Ruoqiao Zhang, Jianyi Zhou, and Wei Hong. Digital beamforming-based massive mimo transceiver for 5g millimeter-wave communications. *IEEE Transactions on Microwave Theory and Techniques*, 66(7):3403–3418, 2018. doi:10.1109/TMTT.2018.2829702.

[12] Gene H. Golub and Charles F. Van Loan. *Matrix Computations - 4th Edition*. Johns Hopkins University Press, Philadelphia, PA, 4 edition, 2013. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781421407944, arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781421407944, doi:10.1137/1.9781421407944.

[13] J.W. Luo and C.C. Jong. Scalable linear array architectures for matrix inversion using bi-z cordic. *Microelectronics Journal*, 43(2):141–153, 2012. URL: https://www.sciencedirect.com/science/article/pii/S0026269211002199, doi:10.1016/j.mejo.2011.10.009.

[14] Jack Volder. The cordic computing technique. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, IRE-AIEE-ACM '59 (Western), page 257–261, New York, NY, USA, 1959. Association for Computing Machinery. doi:10.1145/1457838.1457886.

[15] Intel® oneapi math kernel library (onemkl). URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html.

[16] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. *LAPACK users' guide*. SIAM, 1999.

[17] Cmsis dsp software library. URL: https://arm-software.github.io/CMSIS-DSP/main/index.html#license.

[18] Pulp. URL: https://pulp-platform.org/.

[19] Pulp dsp: Digital signal processing on parallel ultra low power platform. URL: https://github.com/pulp-platform/pulp-dsp.

[20] Amirhossein Kiamarzi, Davide Rossi, and Giuseppe Tagliavini. Qr-pulp: Streamlining qr decomposition for risc-v parallel ultra-low-power platforms. In *Proceedings of the 21st ACM International Conference on Computing Frontiers*, CF '24, page 147–154, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3649153.3649210.

[21] Gap. URL: https://en.wikichip.org/wiki/greenwaves/gap8.

[22] Zoltán Endre Rákossy, Farhad Merchant, Axel Acosta-Aponte, S.K. Nandy, and Anupam Chattopadhyay. Efficient and scalable cgra-based implementation of column-wise givens rotation. In *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, pages 188–189, 2014. doi:10.1109/ASAP.2014.6868659.

[23] Emmanuel Agullo, Cedric Augonnet, Jack Dongarra, Mathieu Faverge, Hatem Ltaief, Samuel Thibault, and Stanimire Tomov. Qr factorization on a multicore node enhanced with multiple gpu accelerators. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 932–943, 2011. doi:10.1109/IPDPS.2011.90.

[24] Farhad Merchant, Tarun Vatwani, Anupam Chattopadhyay, Soumyendu Raha, S.K. Nandy, and Ranjani Narayan. Achieving efficient qr factorization by algorithm-architecture co-design of householder transformation. In *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, pages 98–103, 2016. doi:10.1109/VLSID.2016.109.

[25] Mythri Alle, Keshavan Varadarajan, Alexander Fell, Ramesh Reddy C., Nimmy Joseph, Saptarsi Das, Prasenjit Biswas, Jugantor Chetia, Adarsh Rao, S. K. Nandy, and Ranjani Narayan. Redefine: Runtime reconfigurable polymorphic asic. *ACM Trans. Embed. Comput. Syst.*, 9(2), October 2009. doi:10.1145/1596543.1596545.

[26] Jose M. Rodriguez Borbon, Junjie Huang, Bryan M. Wong, and Walid Najjar. Acceleration of parallel-blocked qr decomposition of tall-and-skinny matrices on fpgas. *ACM Trans. Archit. Code Optim.*, 18(3), May 2021. doi:10.1145/3447775.

[27] Kung. Why systolic architectures? *Computer*, 15(1):37–46, 1982. doi:10.1109/MC.1982.1653825.

[28] W. Morven Gentleman and H. T. Kung. Matrix triangularization by systolic arrays. In *Optics & Photonics*, 1982. URL: https://api.semanticscholar.org/CorpusID:63527864.

[29] C.M. Rader. Vlsi systolic arrays for adaptive nulling [radar]. *IEEE Signal Processing Magazine*, 13(4):29–49, 1996. doi:10.1109/79.526897.

[30] R.L. Walke, R.W.M. Smith, and G. Lightbody. Architectures for adaptive weight calculation on asic and fpga. In *Conference Record of the Thirty-Third Asilomar Conference on Signals, Systems, and Computers (Cat. No.CH37020)*, volume 2, pages 1375–1380 vol.2, 1999. doi:10.1109/ACSSC.1999.831931.

[31] Xiaojun Wang and Miriam Leeser. A truly two-dimensional systolic array fpga implementation of qr decomposition. *ACM Trans. Embed. Comput. Syst.*, 9(1), October 2009. doi:10.1145/1596532.1596535.

[32] Sergio D. Muñoz and Javier Hormigo. High-throughput fpga implementation of qr decomposition. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 62(9):861–865, 2015. doi:10.1109/TCSII.2015.2435753.

[33] Martin Langhammer and Bogdan 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. doi:10.1145/3174243.3174273.

[34] Jieming Xu and Miriam Leeser. Accelerating matrix processing for mimo systems. In *Proceedings of the 11th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, HEART '21, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3468044.3468050.

[35] Upasna Vishnoi, Michael Meixner, and Tobias G. Noll. A family of modular qrd-accelerator architectures and circuits cross-layer optimized for high area- and energy-efficiency. *Journal of Signal Processing Systems*, 83(3):329–356, 2016. doi:10.1007/s11265-015-0976-6.

[36] Lirui Chen, Yu Wang, Zuocheng Xing, Shikai Qiu, Qinglin Wang, and Yongzhong Li. Low latency group-sorted qr decomposition algorithm for larger-scale mimo systems. *IET Communications*, 15(12):1548–1560, 2021. URL: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/cmu2.12168, arXiv:https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/cmu2.12168, doi:10.1049/cmu2.12168.

[37] Mohammad Attari, Jesus Rodriguez Sanchez, and Liang Liu. A floating-point 16×16 svd accelerator for beyond-5g large intelligent surfaces. In *2023 IEEE 66th International Midwest Symposium on Circuits and Systems, MWSCAS 2023*, pages 967–971, United States, 2023. IEEE - Institute of Electrical and Electronics Engineers Inc. 2023 IEEE 66th International Midwest Symposium on Circuits and Systems, MWSCAS 2023 ; Conference date: 06-08-2023 Through 09-08-2023. doi:10.1109/MWSCAS57524.2023.10406077.