Politecnico di Torino

Politecnico di Torino

KTH Royal Institute of Technology

Heterogeneous CGRA Cells for FFT acceleration in FMCW radar systems

Master's Degree in Electronic Engineering

A.a. 2024/2025

Supervisor:
Saba Yousefzadeh

Candidate:
Giuseppe Webber

Examiners:
Andrea Calimera
Ahmed Hemani

October 2025

Abstract

Efficient and reliable Fast Fourier Transform (FFT) computation is essential for radar signal processing, especially in real-time millimeter-wave (mmWave) Frequency-Modulated Continuous-Wave (FMCW) systems where requirements on throughput, latency, and power consumption are critical. Conventional platforms such as Digital Signal Processors (DSPs), Application-Specific Integrated Circuits (ASICs), and Field-Programmable Gate Arrays (FPGAs) inevitably involve trade-offs, as each technology faces inherent limitations in flexibility, energy efficiency, silicon area, and development complexity.

This thesis addresses these challenges by designing a reconfigurable FFT accelerator on the Dynamically Reconfigurable Resource Array (DRRA), a coarse-grained reconfigurable architecture developed by the Silicon Large Grain Objects (SiLago) team at KTH. The proposed design adapts to varying transform sizes and configurations, achieving high throughput and low latency with reduced power consumption. The accelerator is mapped onto the DRRA fabric by instantiating dedicated processing units, designing efficient memory access schemes, and coordinating the operations through a resource-centric instruction model. The assembly and simulation of the architecture are performed by Vesyla, a domain-specific toolchain for DRRA.

Simulation results confirm the functional correctness of the proposed design by direct comparison with the reference model implemented in MATLAB. A further evaluation with a Python-based model shows that the proposed design achieves accuracy levels close to floating-point precision while leveraging fixed-point efficiency, resulting in a relative error of 0.0857% against the reference data and outperforming software implementations using the same numeric format.

Synthesis reports indicate that the accelerator, implemented in 22 nm CMOS technology, operates at frequencies up to 1 GHz and achieves a compact silicon footprint of $0.24\,\mathrm{mm}^2$. Compared with other memory-based reconfigurable FFT accelerators, this represents a significant reduction in area while maintaining competitive performance. The design also demonstrates strong power efficiency, consuming 98.14 mW and achieving 25.3 nJ per 256-point FFT operation, offering a favorable trade-off between area, throughput, and power consumption.

Overall, this work presents a promising solution for next-generation radar systems that combines energy efficiency and flexibility in a compact form factor, suitable for embedded applications. The results demonstrate that the DRRA architecture can deliver high-performance, reconfigurable FFT acceleration, confirming its suitability for demanding signal processing tasks and offering a scalable and reusable framework for future radar and communication systems.

Acknowledgements

I would like to express my sincere gratitude to Professor Ahmed Hemani, head of the SiLago research group at KTH Royal Institute of Technology, for giving me the opportunity to work within his team and for the invaluable experience of contributing to such an inspiring research project.

I would also like to thank Saba Yousefzadeh and the entire SiLago team for their continuous guidance, support, and collaboration throughout this project. Their expertise, availability, and constructive feedback have been essential to the successful development of this work.

A special thanks goes to my friends, who, despite the distance, have always remained close and made me feel as if I had never left.

To my girlfriend, thank you for your constant support and for all the wonderful moments we shared throughout this past year. You have been an essential part of this journey, and your presence has made it so much brighter.

Above all, I am deeply grateful to my parents for their infinite support and belief in me, without which I would not be where I am today. Making you proud has been my greatest motivation.

Table of Contents

List of Figures

Glossary

ACU Accumulation Unit. 24, 25

ADAS Advanced Driver Assistant System. 1

ADC Analog-to-Digital Converter. 15, 16, 52

AGU Address Generation Unit. 35, 40–43, 45, 46, 54, 55

AoA Angle of Arrival. 17, 24

ASIC Application-Specific Integrated Circuit. i, 3, 5, 27, 28, 67

BFU Butterfly Unit. 24

BU Butterfly Unit. 19, 40–44, 52, 54, 55, 67

CADFG Control Address Data-Flow Graph. 36

CFAR Constant False Alarm Rate. 24, 26

CGRA Coarse-Grained Reconfigurable Architecture. 5, 27, 28

CIS Composable Instruction Set. 5, 27, 30, 31, 67

CORDIC Coordinate Rotation Digital Computer. 21, 22, 24, 25

DBF Digital Beam-Forming. 24

DFT Discrete Fourier Transform. 8–10, 12, 20

DIF Decimation in Frequency. 9–11, 20, 39, 41

DIT Decimation in Time. 9, 10, 20

DoA Direction of Arrival. 18

DPU Data Path Unit. 22, 28, 29, 35

DRRA Dynamically Reconfigurable Resource Array. i, 5, 6, 21, 27, 28, 30–37, 39, 52, 54, 57–59, 64, 65, 67, 68

DSP Digital Signal Processor. i, 3, 5, 15, 67

EBE Elementary Bit Exchange. 23

FD-SOI Fully-Depleted Silicon-on-Insulator. 61

FFT Fast Fourier Transform. i, 3–6, 8–26, 29, 39–46, 48, 50–52, 54, 55, 57–59, 61–65, 67, 68

FIFO First-In First-Out. 34

FMCW Frequency-Modulated Continuous-Wave. i, 1–6, 13–18, 24, 39, 52, 67, 68

FPGA Field-Programmable Gate Array. i, 3, 5, 27, 28, 67

FSM Finite State Machine. 31, 41, 42, 50

GF GlobalFoundries. 61

GPU Graphics Processing Unit. 27

HAD Highly Automated Driving. 1

HFP Half-precision floating-point. 4

HLS High-Level Synthesis. 36

HPF Half-Precision Floating-point. 25

HWA Hardware Accelerator. 59

IC Integrated Circuit. 27

IDG Instruction Dependent Graph. 36

IF Intermediate Frequency. 14, 15, 17, 24

IO Input/Output. 28, 33–35, 52, 54

IP Intellectual Property. 28

ISA Instruction Set Architecture. 30, 32, 37

LiDAR Light Detection and Ranging. 1

LPF Low-Pass Filter. 15

LSB Least Significant Bit. 10, 22, 43, 58

MIMO Multiple-Input Multiple-Output. 25

mmWave millimeter-wave. i, 1, 2, 6, 39, 67

MPU Magnitude/Phase Calculation Unit. 24

MSB Most Significant Bit. 10, 48

NLOS Non-Line-of-Sight. 2

NoC Network-on-Chip. 28

PASM Pseudo-Assembly. 30–32, 36, 37, 39, 54, 55, 57

PE Processing Element. 19, 25, 28

RF Register File. 28, 29, 34, 35, 39, 41, 42, 52, 54, 55, 67

RISC-V Reduced Instruction Set Computer. 25

ROM Read Only Memory. 20–22, 25, 45, 46, 48, 62

RTL Register Transfer Level. 37, 57

sFFT Sparse Fast Fourier Transform. 25

SiLago Silicon Large Grain Objects. i

SQNR Signal-to-Quantization-Noise Ratio. 4

SRAM Static Random Access Memory. 3, 24, 28, 29, 33–35, 41, 52, 54, 55, 62

SST Structural Simulation Toolkit. 37

SWB Switch Box. 32, 34, 52

TFG Twiddle Factor Generator. 20, 22, 42, 44, 45, 52, 54, 55, 67

UAV Unmanned Aerial Vehicle. 1

USV Unmanned Surface Vehicle. 2

WAR Write-After-Read. 37

WAW Write-After-Write. 37

WMU Window Multiplication Unit. 24

Chapter 1

Introduction

1.1 Background

Radar technologies play a key role in modern sensing systems, providing crucial information about the environment and moving targets. Among the various radar technologies, particular attention has been paid in recent years to Frequency-Modulated Continuous-Wave (FMCW) radars, especially within the millimeter-wave (mmWave) band, which have emerged due to their versatility.

FMCW radars, compared to other radar technologies, have the great advantage of being able to simultaneously determine both the range and the radial velocity of the target, while ensuring high accuracy and range resolution. This capability, combined with advantages such as its small form factor and low cost compared to technologies such as Light Detection and Ranging (LiDAR), and its good robustness to various environmental conditions, has positioned FMCW radars as the sensor of choice in many demanding applications.

One of the most significant areas where FMCW radar is widely applied is in the automotive domain. In this field, the reliable detection of obstacles and road users and, therefore, the robustness to environmental factors, such as rain and fog, is of fundamental importance. FMCW radar finds wide application in Advanced Driver Assistant System (ADAS) and is considered a key sensing technology for Highly Automated Driving (HAD) [1]. Specific functions in ADAS include adaptive cruise control and lane change assistance [1][2]. Modern high-performance automotive radar has evolved into imaging sensors capable of mapping the surrounding environment in multiple dimensions, without incurring privacy issues, which must be considered with traditional camera-based systems.

Radar FMCW-based systems find extensive applications in robotics and autonomous vehicles. Single-chip mmWave radars, valued for their high integration and lower price, are used in mobile robots such as wheeled robots, Unmanned

Aerial Vehicles (UAVs), and Unmanned Surface Vehicles (USVs) [2]. They play a crucial role in the perception of mobile robots under harsh conditions, in tasks such as environment mapping, target localization, and object detection [2].

FMCW mmWave radars are also finding their way into novel applications in the field of health and monitoring, in particular for automatic non-contact monitoring of respiratory and heart rates [3][4]. Exploiting its high range resolution, this technology is capable of detecting minute chest displacements caused by breathing and heartbeat, and thus, the waveforms of vital signals can be extracted and analyzed. This capability is essential for monitoring vital signals in real-time, with applications in medical diagnostics, elderly care, sleep monitoring, psychological wellness, or the detection of infectious diseases [3]. The non-contact detection ability solves the drawbacks associated with contact-based methods, and radar is chosen over camera-based systems for its robustness in variable lighting conditions and for privacy issues [3].

Furthermore, FMCW radars are traditionally employed in target detection and tracking for indoor and outdoor surveillance [5]. In complex environments, this extends to classifying moving targets as human vs. non-human, helping to prevent false alarms caused by pets, fans, or trees, and detecting and localizing the source of motion [5].

A particularly challenging application is the detection of actionless humans hidden in Non-Line-of-Sight (NLOS) regions [4]. This capability is highly relevant for scenarios like fire rescue/earthquake relief to find trapped people and in military actions to confirm the presence of individuals hiding behind corners. Since actionless targets are difficult to distinguish from static objects, detecting vital signs, particularly breathing, is key to confirming human presence under NLOS conditions [4].

1.2 Requirements

The wide range of safety-critical applications served by mmWave FMCW radars places demanding constraints on the radar's digital signal processing capabilities. In scenarios such as autonomous driving, real-time and continuous environmental perception is essential to make immediate and accurate decisions in order to avoid potential collisions or other hazards [6]. This necessitates that all signal processing tasks, from data acquisition to target detection, be completed within extremely tight temporal bounds. Specifically, the complete signal processing chain must operate within the refresh rate of the input data to avoid data loss and ensure continuous operation. For example, in a mmWave FMCW radar operating at a typical frequency of 76-77 GHz, the time frame between consecutive data inputs is on the order of tens of milliseconds [6].

Traditionally, the high computational demands have been met using multi-core general-purpose processors, Digital Signal Processor (DSP) cores, or Field-Programmable Gate Arrays (FPGAs). While these platforms offer flexibility and ease of development, they frequently encounter limitations in terms of chip area and power consumption, factors that are especially critical for embedded real-time systems with constrained power and hardware resources.

Meeting the most stringent timing and power constraints requires the use of highly optimized, dedicated hardware accelerators. Application-Specific Integrated Circuits (ASICs), designed explicitly for radar processing tasks, can offer orders-of-magnitude improvements in both speed and energy efficiency compared to software-based or programmable alternatives, often resulting in reduced chip area. Such custom architectures are capable of sustaining the high throughput and low latency required for continuous, reliable operation in safety-critical radar systems.

Within the signal processing pipeline of FMCW radar systems, the Fast Fourier Transform (FFT) represents the most computationally intensive operation and must handle large volumes of data at high speeds. It plays a central role in converting time-domain radar signals into the frequency domain, enabling the extraction of target range and velocity. The theory behind this operation will be explained in more detail in Section 2.1. Due to the high computational load and real-time processing demands, the implementation of optimized FFT hardware accelerators is essential. A well-designed FFT accelerator can significantly reduce latency, increase throughput, and minimize power consumption, all of which are critical for embedded and automotive radar systems [7]. Several design strategies and architectural considerations can contribute to the efficiency of FFT accelerators.

FFT accelerators architectures can be broadly categorized into pipelined and memory-based designs. Pipelined architectures are well-suited for applications requiring high throughput and low latency, as they allocate dedicated hardware to each stage of the FFT, allowing data to flow without interruption. However, this comes at the expense of increased chip area and power consumption, since each computational stage requires its own resources.

Conversely, in memory-based architectures, the FFT is performed "in-place" or iteratively, reusing processing elements and memory blocks within the design to execute the multiple stages of the FFT sequentially. This approach significantly reduces chip area and is particularly advantageous for longer FFT transforms [8]. However, it typically incurs higher latency and involves more complex control logic due to the need for repeated memory accesses and data shuffling.

To balance the trade-offs between area efficiency and computational speed, hybrid architectures have emerged. Static Random Access Memorys (SRAMs) buffers are incorporated within a pipelined structure, minimizing external memory bandwidth and optimizing overall resource utilization. This makes hybrid architectures especially attractive for real-time radar applications with moderate area

constraints [6].

Given the wide variety of FMCW radar applications, other factors must also be considered when designing the FFT accelerators, such as the transform length and the desired numerical precision, to enable adaptation to various system constraints. These factors directly influence both accuracy and the Signal-to-Quantization-Noise Ratio (SQNR), which has to be carefully evaluated to avoid potential loss of information. FFT accelerators typically support variable transform lengths ranging from 64 to 4096 points. Smaller points transforms may be sufficient for applications demanding low latency or lower resolution, while longer FFTs are essential when high range or velocity resolution is required [7]. To maintain efficiency across different FFT lengths, modern architectures employ mixed-radix algorithms. These algorithms use combinations of radix-2, radix-4, or higher-order butterfly units to decompose the FFT into stages that match with the transform size. By dynamically selecting radix types and stage structures, the processor can minimize the number of complex multiplications and memory operations, thereby optimizing performance and resource utilization for any given FFT length [7].

Numerical precision plays a critical role in FFT accelerator design. The choice between fixed-point and floating-point arithmetic directly impacts computational accuracy, dynamic range, hardware complexity, and power consumption. Fixed-point arithmetic is generally preferred in constrained embedded systems due to its simplicity and low resource usage. However, fixed-point computations are more susceptible to quantization noise and dynamic range limitations, particularly in multistage 2D-FFTs, where noise accumulation and signal scaling can significantly degrade output accuracy. To overcome these limitations, many radar-oriented FFT accelerators adopt floating-point arithmetic, particularly in stages where precision is critical. Half-precision floating-point (HFP) formats, which use fewer bits than standard IEEE-754 single-precision representations, offer a compromise between accuracy and resource efficiency. HFP supports a broader dynamic range than fixed-point while requiring fewer logic gates and memory resources compared to full-precision floating-point [7].

Ultimately, the decision between fixed- and floating-point implementations is context-dependent. In low-power radar modules with moderate performance, fixed-point may be sufficient. In contrast, floating-point FFT accelerators are preferred in high-resolution, high-dynamic-range radar systems, where precision is critical and energy trade-offs are justified by performance gains.

As FMCW radar systems continue to evolve toward higher resolutions and more complex signal processing chains, the design of highly efficient dedicated hardware FFT accelerators will be indispensable for developing low-power, high-speed, and compact radar systems suitable for demanding applications in automotive, drone, and wearable device markets.

1.3 Motivation

Given the demands of FMCW radar processing, particularly the need for real-time, low-latency execution of FFTs combined with flexibility and adaptability to diverse application scenarios, there is a strong incentive to design specialized accelerators. Traditional CPU or GPU-based processing may be inadequate for embedded radar systems due to constraints on power consumption and physical size. While ASICs offer high efficiency, they lack flexibility. Conversely, FPGAs are reconfigurable but often inefficient in terms of area and energy consumption for FFT-dominated workloads.

This motivates the adoption of Coarse-Grained Reconfigurable Architectures (CGRAs), which strike a balance between performance and flexibility. One notable CGRA platform is the Dynamically Reconfigurable Resource Array (DRRA), developed by the Silago group at KTH. The DRRA is designed as a computational fabric, specifically tailored for signal processing and compute-intensive applications [9, 10]. Its design vision is to deliver significantly improved performance and efficiency over FPGAs and DSPs, while offering crucial reconfigurability and shorter development cycles compared to ASICs, albeit at the cost of slightly lower peak performance and area density.

The implementation of an FFT accelerator for FMCW radar systems can greatly benefit from the DRRA architecture, as it aligns well with the requirements discussed in the previous section. These include exploiting parallelism, efficiently handling streaming data, and offering advanced configurability and adaptability. The spatial computing nature of the DRRA enables multi-level parallelism, which, in the context of an FFT accelerator, translates into the use of multiple butterfly units and concurrent FFT computations on different chirps, resulting in significantly faster overall data processing. Furthermore, DRRA is particularly well-suited for streaming data applications, which are prevalent in FMCW radar signal processing and often characterized by static loops. This efficiency stems from DRRA's unique Composable Instruction Set (CIS), which is based on resource-centric instructions. In this paradigm, each instruction configures and controls a single resource, leading to a substantial reduction in control overhead [11]. Finally, the DRRA's ability to offer a high degree of configurability and adaptability is a critical feature for evolving FMCW radar systems. A DRRA-based FFT accelerator can be reconfigured to meet the needs of diverse operating environments, varying object sizes, and differing performance requirements, without the need for new hardware fabrication. This is achieved by dynamically configuring the architecture to process variable FFT lengths and using different radix combinations.

1.4 Objectives

This master's thesis aims to address the critical demand for highly efficient and adaptable computational solutions in advanced mmWave FMCW radar systems by developing and evaluating novel hardware accelerators for FFT computations on the DRRA architecture.

The primary objective of this work is to exploit multi-level parallelism in FFT implementations using the reconfigurable DRRA fabric. An FFT resource conforming to the DRRA template will be designed and investigated within the context of 2D-FFTs for FMCW radar applications, exploring varying transform dimensions and degrees of parallelism enabled by the architecture's inherent flexibility.

The effectiveness of the implemented FFT accelerators will be evaluated using real-world mmWave radar use-case data, with specific requirements drawn from an ongoing industrial project. This practical validation will highlight the viability and advantages of employing reconfigurable architectures to meet the computational demands of modern radar signal processing workloads.

Ultimately, this thesis seeks to establish the foundation for a flexible DRRA-based development kit for rapid exploration of the FFT implementation space, providing a pathway towards more energy-efficient, high-performance, and adaptable hardware solutions for next-generation FMCW radar systems.

Chapter 2

State of the Art

2.1 The fast Fourier transform

Fourier analysis is a fundamental tool in signal processing, enabling the transformation of a signal from the time domain to its representation in the frequency domain. Thus, any complex signal can be expressed as the sum of sine waves, each characterized by its amplitude, frequency, and phase. This decomposition facilitates spectral analysis, which is the basis for a wide range of modern applications, from communications to audio, image, and radar processing.

In the continuous domain, the Fourier series allows the representation of a periodic function $x(t)$ with period T as a sum of harmonically related complex exponentials:

$$x(t) = \sum_{n=-\infty}^{\infty} F(n)\, e^{j2\pi nt/T} \tag{2.1}$$

where the coefficients $F(n)$ quantify the contribution of the n-th harmonic component and are given by:

$$F(n) = \frac{1}{T} \int_0^T x(t)\, e^{-j2\pi nt/T}\, dt \tag{2.2}$$

For non-periodic signals, this concept is generalized using the Fourier transform, which describes a signal's frequency content over a continuous range:

$$X(f) = \int_{-\infty}^{\infty} x(t)\, e^{-j2\pi ft}\, dt \tag{2.3}$$

$$x(t) = \int_{-\infty}^{\infty} X(f)\, e^{j2\pi ft}\, df \tag{2.4}$$

The reciprocal and bidirectional relationship between Equations 2.3 and 2.4 defines the Fourier transform. It provides a complete representation of a signal in both the time and frequency domains.

In real-world systems, continuous-time signals are neither infinite in duration nor continuously observable. Instead, they are sampled and digitized, resulting in finite-length, discrete-time signals. This transition requires discrete-time analysis and leads to the formulation of the Discrete Fourier Transform (DFT).

To analyze signals digitally, a continuous-time function $x(t)$ is sampled at regular intervals, yielding a discrete sequence $x[n] = x(nT_s)$, where T_s is the sampling period. If the signal is band-limited with a maximum frequency f_{max}, the Nyquist-Shannon sampling theorem states that the original signal can be perfectly reconstructed from its samples, without loss of information, provided that the sampling frequency $F_s = 1/T_s$ satisfies $F_s > 2f_{max}$.

The DFT provides a numerical method to evaluate the frequency content of sampled, finite-length signals. Given a sequence of N samples $x[n]$, for $n = 0, 1, \ldots, N-1$, the DFT is defined as:

$$X[k] = \sum_{n=0}^{N-1} x[n]\, e^{-j2\pi kn/N}, \quad k = 0, 1, \ldots, N-1 \tag{2.5}$$

This transformation maps the time-domain sequence $x[n]$ to a frequency-domain sequence $X[k]$, where each coefficient $X[k]$ corresponds to a discrete frequency bin. The DFT assumes that the input signal is periodic with period N, and consequently the frequency spectrum is also periodic with the same length.

Although the DFT is conceptually straightforward, its computational cost is significant. Computing each of the N output values requires a sum of N terms, resulting in a total computational complexity of $\mathcal{O}(N^2)$ [12]. This quadratic complexity becomes a major bottleneck for large N, particularly in real-time signal processing applications such as radar. To overcome this limitation, the FFT was developed.

The FFT is a class of algorithms that compute the DFT efficiently by eliminating redundant calculations. The most well-known and widely used algorithm is the Cooley-Tukey FFT [13], introduced in 1965, which is especially efficient when the sequence length N is a power of two, i.e., $N = 2^\gamma$.

The Cooley-Tukey algorithm employs a divide-and-conquer strategy to recursively decompose a large DFT into smaller DFTs, leveraging the periodicity and symmetry properties of the complex exponential functions. This approach significantly reduces the number of required operations to $\mathcal{O}(N \log N)$, resulting in substantial performance gains.

The FFT operates through a sequence of stages composed of basic addition and subtraction operations, commonly referred to as butterfly computations. For an N-point input sequence $x[n]$, each stage combines pairs of inputs with twiddle factors, which are complex exponentials defined as:

$$W_N^k = e^{-j\frac{2\pi k}{N}} \tag{2.6}$$

Twiddle factors are complex numbers that determine the specific angle of a phase shift. They can be represented as a rotation matrix:

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} = \begin{bmatrix} C & -S \\ S & C \end{bmatrix} \tag{2.7}$$

The complex rotation operation accounts for the phase shifts introduced during the transformation and is central to the efficient computation of the FFT. The operation can be described by the following formula:

$$\begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} C & -S \\ S & C \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{2.8}$$

where $X + jY$ is the result of the rotation and $C, S \in \mathbb{Z}$ are the real and imaginary part of the rotation coefficient $C + jS$.

2.1.1 Decimation of the FFT

Decomposition is the fundamental strategy that reduces computational complexity and enables faster execution of the FFT. In the Cooley-Tukey algorithm, this is accomplished through two main variants: Decimation in Time (DIT) and Decimation in Frequency (DIF).

In the DIT FFT, the time-domain input sequence $x(n)$ is divided into even- and odd-indexed samples, resulting in two DFTs of size $N/2$. These smaller DFTs are then combined by first multiplying the coefficients of the odd-indexed part by the corresponding twiddle factors, as defined in Equation 2.6, and subsequently applying the butterfly operation:

$$\begin{aligned} y_0 &= x_0 + x_1 \cdot W_N^k \\ y_1 &= x_0 - x_1 \cdot W_N^k \end{aligned} \tag{2.9}$$

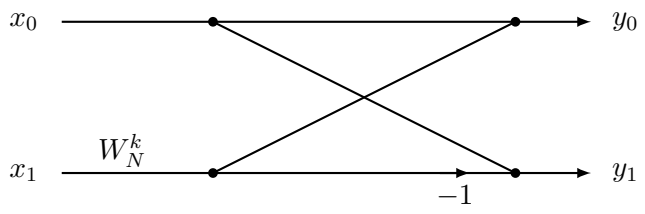


Figure 2.1: Dataflow of the DIT radix-2

A key feature of the DIT implementation is the bit-reversal permutation applied to the input sequence as a pre-processing step. Specifically, each index i in the

sequence $0, \ldots, N-1$ is assigned to a new position whose binary representation is the reverse of the original index (i.e., reading bits from Least Significant Bit (LSB) to Most Significant Bit (MSB)). This reordering simplifies the recursive decomposition by enabling efficient separation of even and odd elements once, before the FFT stages are executed.

In the DIF FFT, the input sequence is split between its first and second halves, and the size-$N/2$ DFTs are recursively applied to these two contiguous segments. Unlike the DIT variant, the input sequence in DIF is already in correct order, requiring no initial reordering. However, as the recursion progresses, the intermediate data are interleaved, resulting in the final FFT coefficients being in bit-reversed order. Consequently, a bit-reversal permutation is required at the output rather than the input.

The butterfly operation in the DIF case is given by:

$$\begin{aligned} y_0 &= x_0 + x_1 \\ y_1 &= (x_0 - x_1) \cdot W_N^k \end{aligned} \tag{2.10}$$

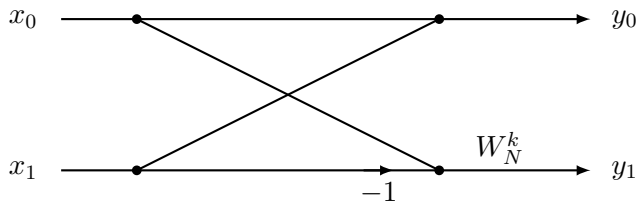


Figure 2.2: Dataflow of the DIF radix-2

Figure 2.3 illustrates the dataflow graph of a 16-point radix-2 DIF FFT, showing the recursive structure of butterfly operations across its $\log_2 16 = 4$ stages.

2.1.2 The radix of the FFT

Up to this point, the discussion has focused on a recursive decomposition into two sub-FFTs, corresponding to the radix-2 formulation, which is the canonical and most straightforward version of the Cooley-Tukey algorithm. However, alternative radix variants offer trade-offs in terms of arithmetic complexity, memory access patterns, and suitability for hardware implementation. To optimize both computation and hardware efficiency, several FFT algorithms have been developed, including radix-2, radix-4, mixed-radix, radix-2^i, and split-radix approaches.

These alternative methods often provide advantages such as reduced arithmetic operations or improved memory locality. However, these benefits come at the cost of increased control complexity, more intricate memory access schemes, and more elaborate data permutations.

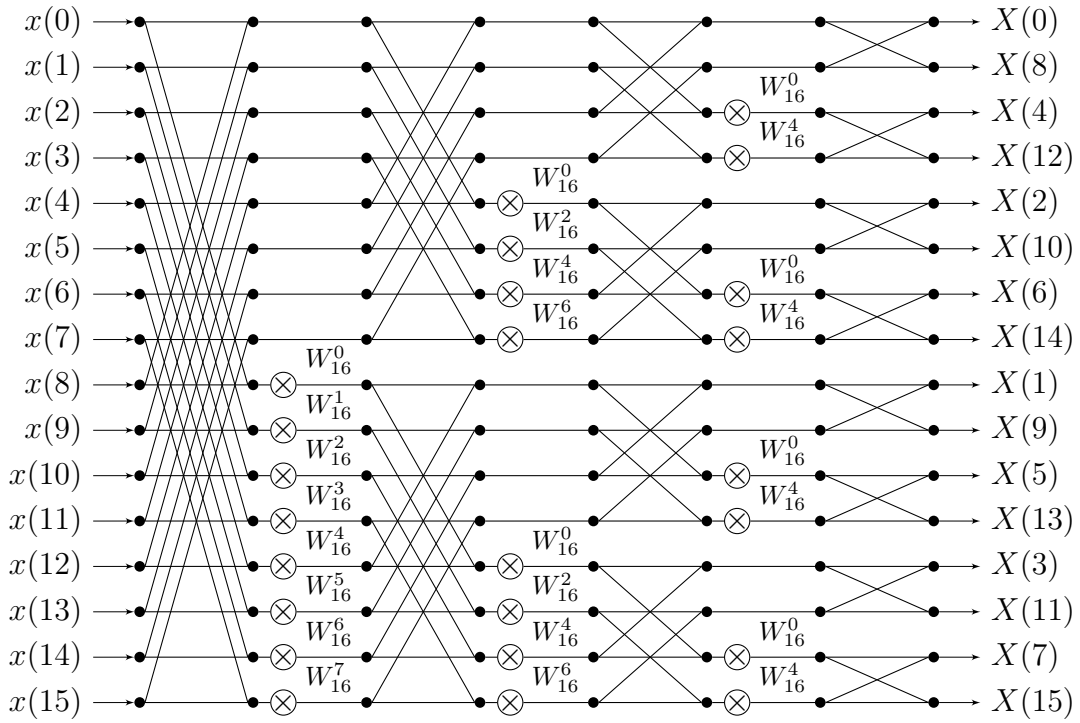


Figure 2.3: Dataflow of a 16 points DIF radix-2 FFT

Radix-2 FFT is the most basic variant of the algorithm, with its primary advantage being its simplicity of implementation and broad applicability. However, this comes at the cost of a relatively high number of stages and complex multiplications, which can limit performance in time- and resource-constrained applications. Higher-radix FFT algorithms, such as radix-4 and radix-8, extend the concept of radix-2 by processing more input elements per stage. For example, radix-4 DIF butterfly operations can be described by the equation:

$$Y = D(k)\, F_4\, x \tag{2.11}$$

where F_4 is the set of radix-4 operations and $D(k)$ the diagonal matrix for twiddle factor multiplication. So, the corresponding matrix multiplication is as follows:

$$\begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & W_N^k & 0 & 0 \\ 0 & 0 & W_N^{2k} & 0 \\ 0 & 0 & 0 & W_N^{3k} \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{2.12}$$

that can also be expressed linearly as:

$$\begin{aligned} Y_0 &= x_0 + x_1 + x_2 + x_3, \\ Y_1 &= (x_0 - jx_1 - x_2 + jx_3) \cdot W_N^k, \\ Y_2 &= (x_0 - x_1 + x_2 - x_3) \cdot W_N^{2k}, \\ Y_3 &= (x_0 + jx_1 - x_2 - jx_3) \cdot W_N^{3k} \end{aligned} \tag{2.13}$$

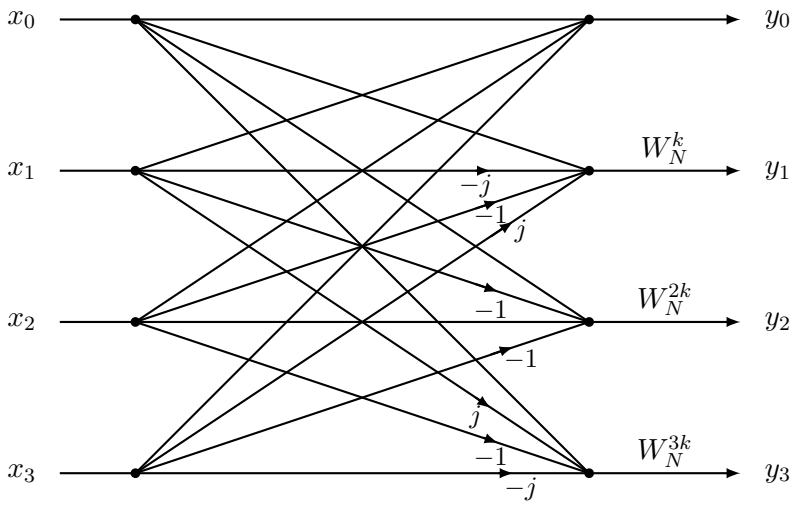


Figure 2.4: Dataflow of the DIF radix-4

If the input size N is a power of the radix used, these algorithms reduce the total number of stages and arithmetic operations compared to radix-2, resulting in improved throughput and reduced latency. On the downside, higher-radix algorithms require more complex butterfly structures and control logic, and they restrict flexibility in terms of supported FFT sizes.

To overcome these limitations, mixed-radix FFT algorithms have been formulated. It decomposes the FFT length N into a product of smaller, possibly unequal factors, such as 2, 3, 4, 5, or 8, and applies different radix strategies at each stage. Mixed-radix FFTs are widely used in general-purpose processors, as they support arbitrary FFT lengths and enable optimizations based on input characteristics. However, they introduce additional complexity in scheduling and twiddle factor management, particularly when implemented in hardware.

The split-radix FFT was developed to combine the radix-2 and radix-4 strategies, reducing the number of arithmetic operations. This hybrid approach decomposes a size-N DFT into one DFT of size $N/2$ and two DFTs of size $N/4$. It is especially efficient for real-valued input data and achieves the lowest known operation count among conventional FFT algorithms [14]. The trade-off lies in the complexity

of its implementation, which demands intricate control flow and twiddle factor scheduling. Consequently, split-radix FFTs are more common in high-performance software applications than in fixed-function hardware.

In [15], the radix-2^2 algorithm, also known as radix-2^i, is introduced as an interesting compromise. It retains the simpler butterfly structure of the radix-2 algorithm while achieving the same multiplicative complexity as radix-4. This makes it particularly appealing for hardware implementations, where regularity and structural simplicity are often prioritized over minimal arithmetic operation counts.

2.2 mmWave FMCW radars

An FMCW radar operates by transmitting a sequence of linearly modulated frequency signals, called chirps. Each chirp is characterized by its start frequency f_c, bandwidth B and duration T_c. The instantaneous frequency of a sawtooth waveform, commonly used in fast-ramp FMCW systems, can be expressed as:

$$f(t) = f_c + St \tag{2.14}$$

where $S = B/T_c$ is the slope of the chirp and defines the rate at which its frequency increases over time, as shown in Figure 2.5

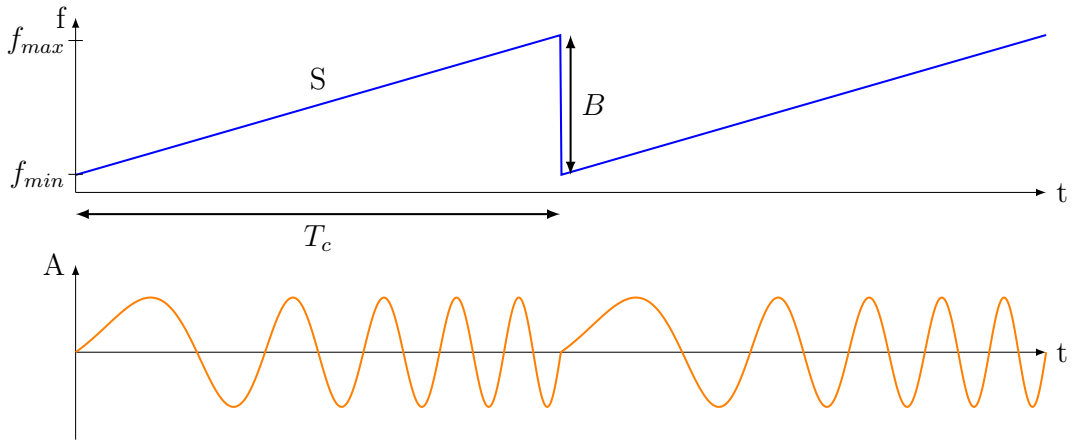


Figure 2.5: Frequency-modulated signal transmitted by the radar

This signal is generated by a synthesizer and transmitted by the TX antenna. The chirp reflects off an object and is subsequently received by the RX antenna after a round-trip delay $\tau = 2d/c$, where d is the distance to the object and c is the speed of light. In an FMCW radar system, a mixer combines the transmitted

and received signals, producing an output signal x_{out}. Given the input signals:

$$\begin{aligned} x_1 &= \sin(w_1 t + \phi_1) \\ x_2 &= \sin(w_2 t + \phi_2) \end{aligned} \tag{2.15}$$

the mixer operation yields:

$$x_{out} = \sin[(w_1 - w_2)t + (\phi_1 - \phi_2)] \tag{2.16}$$

Thus, the output sinusoid has a frequency equal to the difference between the input frequencies and a phase equal to the difference in their phases. In the FMCW context, the mixer output is referred to as the Intermediate Frequency (IF) signal, which contains information about the range, velocity, and angle of the target [16].

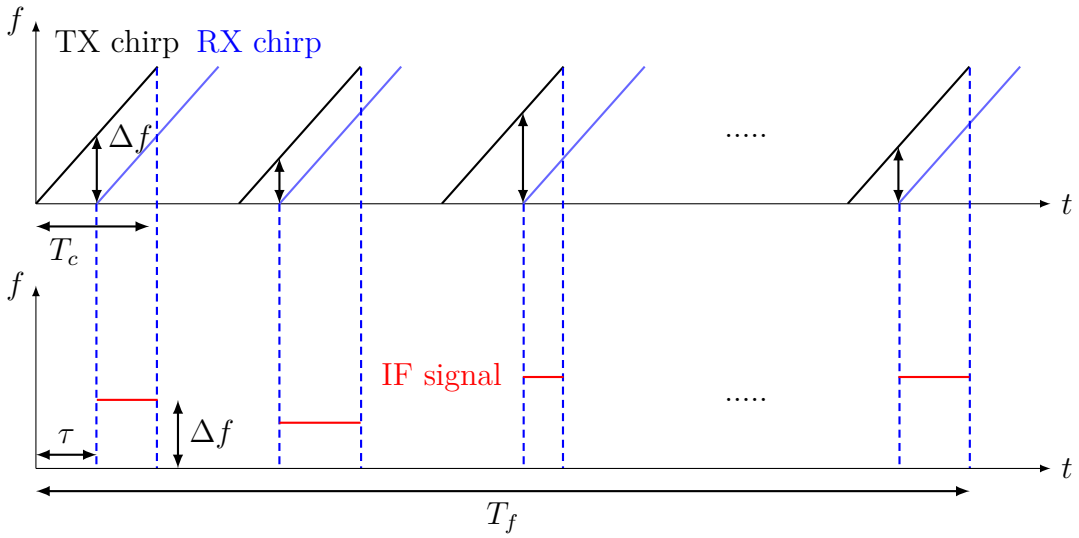


Figure 2.6: IF signal resulting from the difference between the TX and RX signals

2.2.1 The range-FFT

For a single object at distance d, the IF signal results in a constant-frequency tone, with frequency directly proportional to the object's range:

$$f_{IF} = \frac{S2d}{c} \tag{2.17}$$

where S is the chirp slope and c is the speed of light.

The FFT converts the time-domain IF signal into the frequency domain, producing a peak at the frequency tone corresponding to the target range. For this reason, this first FFT is known as the range-FFT.

When multiple objects are present in the radar field of view, the RX antenna receives multiple reflected signals for the same chirp, resulting in multiple tones in the IF signal. The resulting frequency spectrum will exhibit multiple peaks. In this case, an important parameter is the range resolution d_{res}, which refers to the radar's ability to distinguish between two closely spaced targets [16].

If two targets are separated by a distance Δd, the frequency difference in their respective IF signals is:

$$\Delta f = \frac{S2\Delta d}{c} \tag{2.18}$$

Given the frequency resolution of the FFT $\Delta f = 1/T$, we obtain:

$$\Delta f > \frac{1}{T_c} \Rightarrow \frac{S2\Delta d}{c} > \frac{1}{T_c} \Rightarrow \Delta d > \frac{c}{2ST_C} \Rightarrow \frac{c}{2B} \tag{2.19}$$

Thus, the range resolution d_{res} of an FMCW radar depends solely on the chirp bandwidth B: increasing the bandwidth swept by the chirp improves resolution.

Another important design consideration is the maximum detectable range d_{max}. The maximum frequency in the IF signal f_{IFmax} is directly linked to d_{max}, as evident from Equation 2.17. To enable digital processing, on a DSP or hardware accelerator, the IF signal must be digitized, typically using a Low-Pass Filter (LPF) followed by an Analog-to-Digital Converter (ADC). The ADC samples the signal at a fixed rate F_s, which limits the IF bandwidth and consequently the maximum detectable range:

$$d_{max} = \frac{F_s c}{2S} \tag{2.20}$$

2.2.2 The Doppler-FFT

So far, only static targets have been considered. However, in practical applications, detecting moving objects is essential, whether they are fast-moving vehicles or slow periodic motions such as breathing. In some cases, such as medical monitoring, the frequency shift in the IF signal caused by small displacements may be too small to detect in the spectrum. For this reason, the phase of the IF signal must be considered.

As shown in Equation 2.16, the phase of the IF signal corresponds to the phase difference between the TX and RX signals. If the round-trip delay between two consecutive chirps changes by a small amount $\Delta\tau$, the resulting phase change $\Delta\phi$ is:

$$\Delta\phi = 2\pi f_c \Delta\tau = \frac{4\pi\Delta d}{\lambda} \tag{2.21}$$

For millimeter-wave radars, a displacement of a few millimeters corresponds to a significant phase shift, on the order of $\pi = 180°$. Therefore, by observing the phase

difference ω between the peaks in the frequency spectrum of successive chirps, the target velocity v can be estimated as:

$$\omega = \frac{4\pi\Delta d}{\lambda} = \frac{4\pi v T_c}{\lambda} \Rightarrow v = \frac{\lambda\omega}{4\pi T_c} \tag{2.22}$$

In a frame composed of N equally spaced chirps, analyzing the time evolution of the phase across chirps enables the estimation of small displacements and periodic signals, such as heartbeat or respiration.

The maximum measurable velocity v_{max} is constrained by the chirp repetition interval T_c:

$$v_{max} = \frac{\lambda}{4T_c} \tag{2.23}$$

So, to measure higher velocities, more closely spaced chirps (smaller T_c) are needed.

When multiple targets in the same range but with different velocities are in the radar field, they cannot be separated using only the range-FFT. To resolve them, a second FFT is applied to the sequence of phasors from the range-FFT peaks. This operation, called Doppler-FFT, extracts the phase variation ω between chirps, revealing the target motion.

As with the range-FFT, it is essential to evaluate the velocity resolution v_{res}, i.e., the minimum velocity difference Δv that can be distinguished. From the FFT resolution in the frequency domain:

$$\Delta\omega = \frac{2\pi}{N} \text{ rad/sample} \tag{2.24}$$

and using Equation 2.22, we find:

$$\Delta\omega = \frac{4\pi\Delta v T_c}{\lambda} \Rightarrow \Delta v > \frac{\lambda}{2NT_c} \tag{2.25}$$

Hence, the velocity resolution is:

$$v_{res} = \frac{\lambda}{2T_f} \tag{2.26}$$

where $T_f = NT_c$ is the frame time. The longer the frame duration, the finer the velocity resolution.

In FMCW radar systems, the collected data is typically structured in a radar matrix, where each row corresponds to a chirp, represented as a data vector after ADC sampling. The range-FFT is applied across each row, while the Doppler-FFT is performed column-wise on the range-FFT output. This two-stage process is known as 2D-FFT processing, and yields the range-Doppler map, which localizes objects in a 2D space of range and velocity [16]. In Figure 2.7, a simulation of the range-Doppler spectrum is shown.

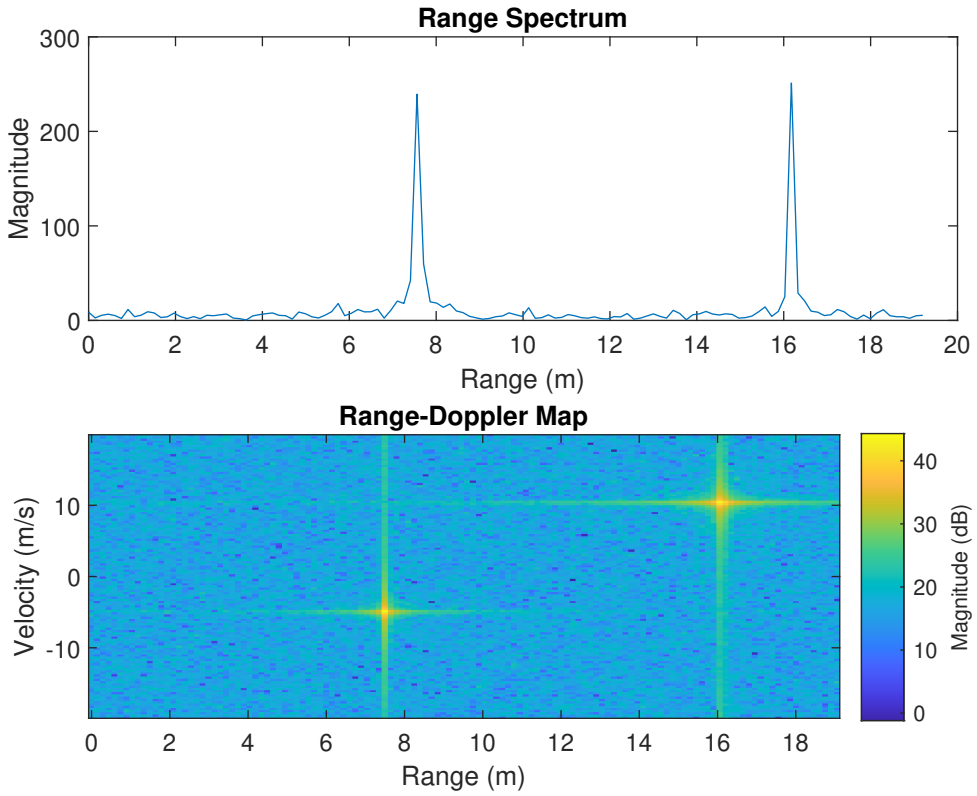


Figure 2.7: Range spectrum and range-Doppler map with two targets: T_1 ($r = 16, v = 10$) and T_2 ($r = 7.5, v = -5$)

2.2.3 The angle-FFT

For complete 3D localization, essential in applications such as autonomous driving, the Angle of Arrival (AoA) of each target must also be estimated. Most FMCW radar systems achieve this by using multiple spatially separated RX antennas. When a target reflects the transmitted signal, the wavefront reaches each RX antenna at slightly different times because of their physical separation. This delay introduces a phase shift in the IF signals received which can be analyzed to estimate direction. The phase difference ω is related to the AoA θ by:

$$\omega = \frac{2\pi d_a \sin(\theta)}{\lambda} \tag{2.27}$$

where d_a is the physical distance between the RX antennas. Solving for the angle:

$$\theta = \sin^{-1}\left(\frac{\lambda\omega}{2\pi d_a}\right) \tag{2.28}$$

Using this relationship, an angle-FFT can be applied to the sequence of complex phasors corresponding to the peaks in the 2D range-Doppler map. The result is a

range-Doppler-angle cube, often referred to as a radar data cube, which enables spatial localization of multiple objects in the environment. Angular estimation, commonly known as Direction of Arrival (DoA) estimation, further enhances radar capability by resolving targets that may overlap in range and Doppler but are spatially separated. This is essential for applications such as autonomous driving, gesture recognition, and medical monitoring [16].

2.3 FFT accelerators

In real-time FMCW radar applications, strict timing requirements must be met to ensure continuous environmental mapping and enable immediate and accurate decision-making. This means that the system must receive radar data, process it and return the necessary information within a time interval shorter than the duration between two consecutive radar chirps. This time constraint heavily depends on the specific application, which dictates the required resolution and the maximum detectable range and velocity.

Moreover, timing constraints are not the only ones that need to be addressed. FMCW radar systems are often deployed in small, low-power embedded platforms, making area utilization and power efficiency equally important design considerations.

The FFT accelerator forms a core component in real-time FMCW radar systems, as it is essential for extracting distance, speed, and angle information from the received signals. Specifically, as previously described in Section 2.2, a 3D-FFT is employed to process radar data. This involves first applying FFT along the rows, followed by another FFT along the orthogonal dimension to generate a range-Doppler map and finally the third FFT for the angular estimation.

Due to demanding performance and energy-efficiency requirements, general-purpose microcontroller-based solutions often fall short. As a result, there is a strong preference for dedicated hardware implementations that offer significantly higher performance with lower power consumption. Consequently, considerable effort has been dedicated to the development of specialized high-speed FFT accelerators capable of computing 3D-FFTs efficiently.

2.3.1 FFT hardware architectures

FFT hardware architectures can be broadly categorized into three main types: fully parallel, iterative, and pipelined architectures [17]. The choice among these depends primarily on throughput requirements, but power consumption and area occupation also vary significantly between these implementations.

Fully parallel FFT architectures represent a direct hardware mapping of the FFT flow graph. Each addition and multiplication operation is implemented with a

dedicated adder or multiplier, resulting in maximal resource usage, with hardware complexity on the order of $\mathcal{O}(N \log N)$ [17]. Although this approach provides the highest degree of parallelism, enabling maximum throughput and minimum latency, it also incurs the highest cost in terms of area and power.

Iterative FFT architectures, also known as memory-based or in-place FFTs, offer a more resource-efficient alternative by reusing Processing Elements (PEs) across multiple iterations. These architectures are based on a central memory or multiple memory banks from which data are read, then processed by a small number of butterfly units and rotators, and finally written back. This approach significantly reduces hardware complexity and is suitable for scenarios in which large data chunks are transmitted intermittently and the computation has to be performed between bursts. The throughput and latency of iterative architectures depend on the number of available PEs and their radix-r configuration. The number of required iterations is $\log_2 N / \log_2 r$ [17], which means that higher-radix Butterfly Units (BUs) can be used to increase throughput at the cost of higher hardware complexity. Efficient memory access is critical in such architectures, typically achieved through multi-bank memory structures and conflict-free access patterns, to avoid stalls and ensure sustained computation rates.

Lastly, pipelined FFT architectures consist of several stages connected in series, each performing a single FFT stage. These architectures can be further divided into serial-pipelined and parallel-pipelined types. The serial-pipelined version achieves a throughput of one sample per clock cycle and requires fewer hardware resources for arithmetic operations, typically on the order of $\mathcal{O}(\log N)$, while memory requirements are on the order of $\mathcal{O}(N)$. In contrast, parallel-pipelined architectures increase throughput by processing multiple samples in parallel, at the expense of higher hardware complexity and power consumption.

Each architectural style presents a trade-off among throughput, latency, power, and area. The selection of an appropriate architecture heavily depends on the specific constraints and performance goals of the target application.

2.3.2 Building blocks

Hardware implementations of FFT require mapping the algorithm onto specific structures designed to meet performance, area, and power consumption requirements. The overall operation is decomposed into smaller sub-operations, which are then mapped onto dedicated building blocks. These hardware blocks perform the operations more efficiently than general-purpose software solutions. A given building block can be implemented in different ways using various components, depending on the specific constraints of the application in terms of speed, power, and silicon area. For FFT computations, the primary building blocks are butterfly units, rotators, and shuffling circuits.

Butterfly units

The butterfly unit forms the foundation of the Cooley-Tukey FFT, as it combines the results of smaller DFTs into larger ones or, conversely, decomposes larger DFTs into smaller components, enabling recursive decimation as described in Section 2.1.1. Butterfly circuits are characterized by their radix. A radix-r butterfly has r inputs and r outputs and computes a r-point DFT. From a hardware perspective, butterfly units consist primarily of adders and rotators, implementing the operations described by Equations 2.9 and 2.10 for DIT and DIF FFTs, respectively.

However, radix-2 and radix-4 butterfly units typically do not require explicit rotators, since the corresponding rotation operations involve trivial multiplications. In the radix-2 case, the only necessary rotation is a multiplication by -1, which can be implemented by simply inverting the sign. In the radix-4 case, the additional $-j/j$ rotation can be achieved by signal routing without using multipliers. For these reasons, radix-2 and radix-4 butterfly units are widely used in hardware designs. In contrast, higher-radix butterflies require non-trivial complex multiplications, necessitating the inclusion of rotators and thereby increasing hardware complexity and cost.

Rotators and Twiddle Factor Generators

Despite the simplifications in lower-radix butterflies, rotations are still essential between FFT stages to apply the twiddle factors and implement non-trivial stages. These operations are handled by dedicated hardware blocks known as rotators, for which various implementation strategies have been proposed in [17].

The most direct implementation of a rotator follows Equation 2.8, which involves four multipliers and two adders:

$$\begin{aligned} X &= x\,C - y\,S \\ Y &= x\,S + y\,C \end{aligned} \tag{2.29}$$

However, this basic structure can be optimized to reduce arithmetic complexity, by rewriting the expressions to use only three multipliers:

$$\begin{aligned} X &= x\,(C + S) - (x + y)\,S \\ Y &= x\,(C - S) + (x + y)\,S \end{aligned} \tag{2.30}$$

The circuits used to implement complex multiplication for rotations are shown in Figure 2.8.

Nevertheless, rotators must be supplied with the appropriate twiddle factor at each stage of the FFT. There are two common approaches for this: using Read Only Memory (ROM)-based lookup tables or employing run-time Twiddle Factor

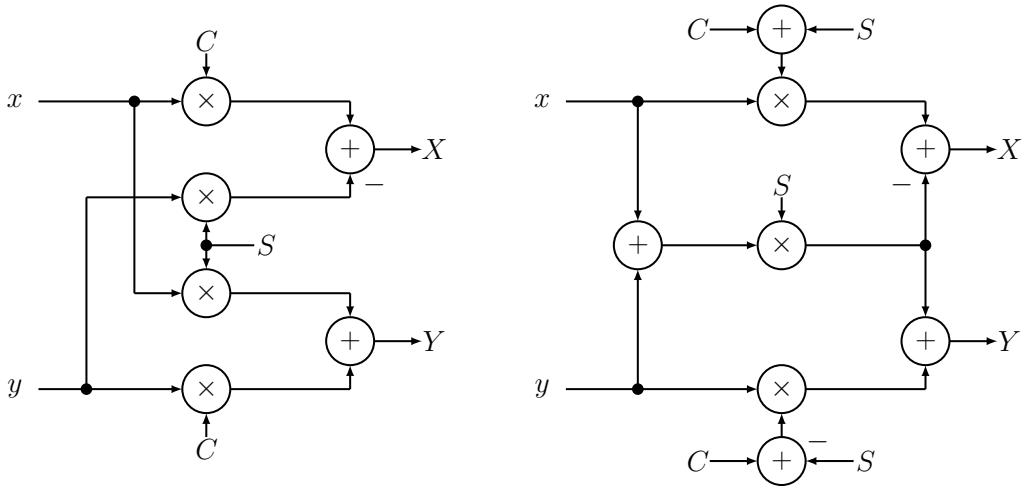


Figure 2.8: Standard complex multiplier based on 2.29 (left) and optimize complex multiplier with one multiplier less based on 2.30 (right)

Generators (TFGs), each with distinct trade-offs regarding memory usage, speed, computational load, and hardware overhead.

In ROM-based solutions, twiddle factors are precomputed and stored in permanent memory. These factors are indexed using the current butterfly count and the FFT stage. This approach is the simplest and most popular for short FFT lengths, where the number of twiddle factors is limited and memory usage remains small. However, for larger FFT sizes, the ROM requirement increases significantly, leading to higher power consumption and increased chip area.

To mitigate the large ROM footprint, designers have explored ways to reduce storage requirements by exploiting the inherent symmetry of twiddle factors. Using trigonometric identities, the total number of factors stored can often be reduced to $N/8 + 1$, as the rest can be generated through operations such as sign changes or exchanging real and imaginary parts [18]. An alternative method is proposed in [19], where a small ROM and minimal control logic is used to generate the required twiddle factors at run-time. This technique involves storing only positive angles and reconstructing the entire set of twiddle factors using quadrant-based rules, achieving a storage requirement of only $N/8$ values.

Another approach is to generate twiddle factors on the fly, almost eliminating memory requirements at the expense of computational complexity. Special hardware is designed to compute the twiddle factors in run-time exploiting Coordinate Rotation Digital Computer (CORDIC) algorithms, polynomial-based techniques, or other mixed approaches.

In [20] the proposed DRRA-based reconfigurable architecture for mixed-radix

FFT includes a novel TFG unit that minimizes memory requirements and simplifies hardware. The proposed design is based on a recursive feedback complex multiplication for computing sine and cosine functions. Unlike ROM-memory methods, this TFG computes initial values and seed values on-the-fly for each processing cell using complex multipliers. It also leverages Data Path Unit (DPU) to internally generate specific twiddle factors, further reducing TFG's workload. This approach leads to a reduction in the number of cycles required to compute a FFT, particularly for higher N values, compared to other ROM-less solutions. In addition, the complexity with respect to CORDIC based TFG architectures is lower, often requiring only two multiplication and two addition operations per output point.

However, recursive approaches present a major drawback, that is, the introduced error propagation in finite-precision computation, due to their feedback nature. A solution for this problem is proposed in [21] in which a conventional recursive sine/cosine function generation algorithm is combined with a small compensation lookup table. This compensation table solves the error propagation problem by replacing the LSBs of the generated output with correction values, guaranteeing full precision.

Shuffling circuits

FFT algorithms require data to be rearranged at each stage to ensure the correct pairing for subsequent butterfly computations. Specifically, in each stage of an $N = 2^n$ point FFT, butterfly units operate on data pairs whose indices differ in a specific bit position b_{n-s} for the current stage s. Therefore, correct selection and routing of the input data at each stage is crucial to guaranty the accuracy of the final result. This rearrangement is based on bit-dimension permutations, which reorder a group of 2^n data elements by permuting the n bits that represent their indices.

A systematic representation of bit-dimension permutations is introduced in [22]. Let σ denote a bit-dimension permutation that transforms a point in n-dimensional space, represented by its binary coordinates $u_{n-1}, u_{n-2}, \ldots, u_0$, into a new point $u_{\sigma(n-1)}, u_{\sigma(n-2)}, \ldots, u_{\sigma(0)}$ by permuting the bit positions. The original data position is denoted by P_0, and the output position by P_1, so that $\sigma(P_0) = P_1$.

Among the wide range of possible bit-dimension permutations, two are particularly important for FFT computations: the perfect shuffle and the bit reversal. Specialized hardware circuits have been designed to perform these permutations quickly and efficiently. Typically, these involve multiplexers that route data between memory locations, along with registers or buffers used for temporary storage and synchronization.

The perfect shuffle is a common permutation in FFT algorithms. It performs a circular shift of the bits of the index to the left by one position, i.e., $\sigma_{PS}(u_{n-1}, u_{n-2}, \ldots, u_1, u_0) = (u_{n-2}, \ldots, u_1, u_0, u_{n-1})$. This permutation can be decomposed into Elementary Bit Exchanges (EBEs), which are simpler bit-dimension permutations that involve only two dimensions at a time [22]. There are three types of EBEs that differ according to the domain of operation.

The first type, called parallel permutations, swaps data positions in space and can be implemented with only multiplexers, without the need for delay elements. The second type, known as serial-serial permutations, uses buffers of length L to delay one data stream by $\Delta t = L$, allowing reordering in the time domain. The third type, serial-parallel permutations, reorders data across both time and space domains by alternating delays and spatial swaps. This type of permutation involves interchanging groups of data streams by selectively delaying and rearranging them. The circuits used for these permutations are shown in Figure 2.9.

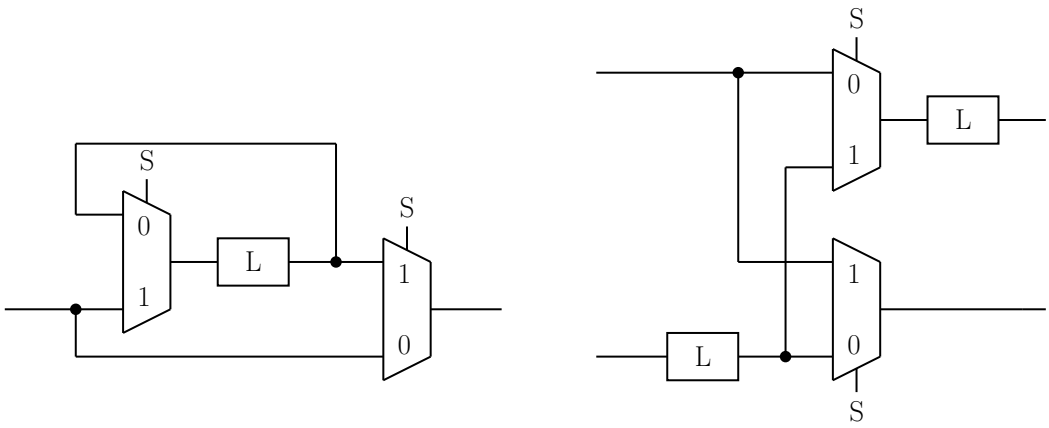


Figure 2.9: Shuffling circuit for serial-serial permutation (left) and serial-parallel permutation (right)

Another important permutation in FFTs is bit reversal, which is required either at the beginning or at the end of the computation depending on the chosen decimation method, as discussed in Section 2.1.1. This permutation reorders the index bits as follows: $\sigma(u_{n-1}, \ldots, u_1, u_0) = (u_0, u_1, \ldots, u_{n-1})$. In [23], an optimal hardware circuit for bit reversal for serial data is proposed using a sequence of blocks, each composed of two multiplexers and a buffer. The length of each buffer depends on the number of bits in the index. Alternatively, parallel-parallel implementations may be used, where the permutation is applied directly to data stored in memory or buffers. These implementations can be optimal in specific architectural scenarios.

2.3.3 FFT accelerators for mmWave FMCW radar

FFT accelerators are the computational backbone of mmWave FMCW radar systems. Their design significantly influences performance in terms of memory usage, execution time, and power consumption. Multiple architectural approaches have been proposed to optimize these aspects and adapt to the diverse constraints of various applications.

The primary requirement for FFT accelerators in FMCW radar is computational efficiency and high throughput, particularly in real-time scenarios where large volumes of data must be processed rapidly to ensure the responsiveness necessary in safety-critical systems. This section presents an overview of several FFT accelerator implementations, highlighting different architectural strategies aimed at optimizing speed, power efficiency, and silicon footprint through specialized hardware and integrated signal processing functions.

In [24], a processor designed for real-time FMCW radar applications is proposed. It integrates the entire signal processing pipeline, including a 2D-FFT accelerator, two-dimensional Constant False Alarm Rate (CFAR) detection, and Digital Beam-Forming (DBF). The system is implemented using a memory-based architecture, with intermediate data stored in single-port SRAMs. The radar system has four RX antennas, each with a dedicated single port SRAM where IF data are stored during the computation of 2D-FFT. Data is read, processed and rewritten to memory sequentially, minimizing memory usage. After FFT, CFAR and DBF modules detect peaks in the Doppler-range map and estimate the target AoA. The 2D-FFT module comprises a controller for address mapping, a FFT calculator, and four SRAMs operating in a pipelined scheme that eliminates transfer time overhead between stages. In total, six SRAMs are used: four to store FFT results and two for CFAR and DBF processing. The DBF module utilizes the CORDIC algorithm to estimate target angles, avoiding multipliers and thereby significantly reducing area and power consumption. The FFT calculator features 128 radix-2 butterfly units and handles 256-point range FFT and 128-chirp Doppler FFT. Butterflies are implemented using four real Booth multipliers that perform fixed-point arithmetic using 48-bit words, with 24 bits each for the real and imaginary components. A truncation logic reduces each component to 16 bits at the output.

An efficient memory-based FFT accelerator which supports variable lengths from 64 to 4096 points is presented in [7]. Like the previous design, it integrates the entire radar signal processing pipeline, including a Window Multiplication Unit (WMU), a Butterfly Unit (BFU), a Magnitude/Phase Calculation Unit (MPU), and an Accumulation Unit (ACU). Input data are first processed by the WMU, which performs windowing using coefficients stored in dedicated SRAM. The BFU executes radix-4/2 butterfly operations of the FFT for various transform lengths, iteratively reading and writing data to a dedicated FFT SRAM. After the FFT stage, the MPU

computes the magnitude using approximate techniques involving only shifters and adders, and calculates the phase using the CORDIC algorithm, avoiding multipliers. The results are accumulated in the ACU and stored in its corresponding memory. A notable feature of this design is its use of Half-Precision Floating-point (HPF) arithmetic. This format mitigates quantization noise, especially relevant in 2D FFT operations, by offering a balance between precision and hardware cost. The HPF adder and multiplier process the sign, exponent, and mantissa separately, employing complex alignment, normalization, and rounding steps.

An alternative approach is proposed in [25], which introduces a low-latency FFT accelerator tailored for 6G applications. This architecture targets fast 4096-point FFT computation through massive parallelism, employing 64 parallel branches in a memory-based setup. A memory bank consisting of 64 parallel memories stores data in natural order. To manage the complexity of parallelism, the perfect shuffle permutation is applied, allowing uniform addressing across all memories. The shuffle is implemented using a sequence of serial-serial, parallel-parallel, and serial-parallel permutations implemented using optimal circuits, as detailed in Section 2.3.2. Data are then fed into PEs, each consisting of a radix-2 butterfly unit and a rotator, implemented with low-area complex multipliers and placed only at odd-indexed terminals. Rotation coefficients are stored in ROM memories and shared between multiple stages and rotators, simplifying control logic and reducing the total rotation memory size to $N/2$.

In automotive applications, Multiple-Input Multiple-Output (MIMO) systems are employed to support highly detailed imaging requirements. MIMO radar systems require high antenna density and therefore the footprint of FFT accelerators must be limited. To this end, [26] presents a low-footprint FFT accelerator targeting Reduced Instruction Set Computer (RISC-V) platforms. This design opts for a memory-based architecture to reuse FFT resources and eliminate the area cost of the multiple stages of the pipelined approach. The memory system includes four SRAM banks connected through a custom crossbar that enables simultaneous read/write access. To further reduce area, the processor employs a dual-radix butterfly unit. For the range-FFT stage, a radix-4 unit processes 16-bit words to meet strict timing constraints with reduced resolution. For Doppler-FFT and angle-FFT stages, a radix-2 unit with 32-bit words provides higher accuracy at the expense of increased latency, since in this step the time requirements are less demanding. The area overhead is further reduced through simplified logic and ultra-compressed twiddle factor ROMs, which exploit trigonometric symmetries and encoding techniques.

Memory optimization is critical for energy efficiency in embedded systems. Many different techniques have been proposed to optimize memory usage and consequently increase power efficiency of FFT computation. A notable contribution in this direction is [8], which introduces an energy-efficient Sparse Fast Fourier

Transform (sFFT) accelerator. The design prioritizes reduced power and area through two strategies: zero-skipping with dynamic thresholding and compressed transpose memory. The zero-skipping technique filters out low-magnitude values using a dynamically calculated threshold, determined as 2σ from the CFAR mean, calculated using the previous 1D-FFT stage. This prevents unnecessary computation and speeds up processing. A sparse transpose memory is designed to compress sparse data after the initial 1D-FFT stage. This addresses the significant challenge of managing extensive datasets, within the area and power constraints of radar systems. By assuming and leveraging the sparse features in the data, the design effectively reduces memory usage by up to 75% compared to dense memory. The FFT accelerator features a variable-length architecture capable of handling from 128 to 2048 points. A first step involving bit reversed permutation arranges the data before feeding them to radix-2 butterfly units. The architecture employs horizontal unfolding, which doubles throughput by splitting computation across two blocks and using stride permutations to enable parallel execution.

In Table 2.1, the implementation details of some of the works discussed are presented.

Work	**# Points**	**Radix**	**Bitwidth**	**Data**	**Architecture**
[26]	1024	radix-2/4	16/32	fixed	memory
[24]	128-256	radix-2	32	fixed	memory
[7]	64-4096	radix-2/4	16	floating	memory
[8]	128-2048	radix-2	32	fixed	memory
[25]	4096	radix-2	32	fixed	memory
[27]	1024	radix-2	32	fixed	pipelined

Table 2.1: Implementation details of FFT accelerators in other works

Chapter 3

Methodology

This chapter presents the methodology and tools used to implement and simulate algorithms on the DRRA platform. It provides an overview of the architecture, the instruction set, and the component library, highlighting how these elements enable efficient mapping of algorithms on DRRA. The chapter also describes the CIS programming model and the Vesyla toolchain, which together facilitate the compilation, scheduling, and verification of algorithms on the reconfigurable fabric.

3.1 DRRA

The DRRA is a CGRA fabric developed to address several challenges in modern embedded system design, particularly in applications involving high computational or signal processing demands. Traditional hardware design approaches often struggle to achieve efficient resource utilization and require many computation cycles, which can reduce overall system performance. A major limitation of existing solutions is their use of computation-centric instruction models. These models process operations sequentially and are not well suited to express complex loop structures efficiently [11]. As a result, many computation cycles are wasted and the available hardware resources are not fully utilized.

Graphics Processing Units (GPUs) have been used to accelerate streaming workloads, but they lack the flexibility needed to adapt to the specific requirements of different streaming applications. Other platforms, such as ASICs and FPGAs, also present trade-offs. While ASICs provide high performance and energy efficiency for fixed tasks, they are expensive to design and lack adaptability. FPGAs offer more flexibility but suffer from lower silicon and computational efficiency, higher power consumption, and increased configuration overhead. These issues are mainly caused by their fine-grain reconfigurability and the large area required for their interconnect networks [9]. Moreover, the increasing complexity of Integrated Circuit

(IC) designs negatively affects development time and effort, contributing to the challenges of time to market [28].

CGRAs work at a coarser level of granularity compared to traditional reconfigurable architectures like FPGAs, mapping operations onto an array of tiles, each functioning as either a PE or a memory tile. They aim to find a balance between the efficiency of ASICs and the flexibility of FPGAs, offering better computational efficiency than FPGAs and reducing engineering effort compared to ASICs [9]. The DRRA follows this approach by adopting a resource-centric instruction model, where instructions are directly linked to hardware resources. This enables more efficient task scheduling and execution, allowing for better use of resources.

In addition to improving performance, DRRA supports low-cost customization. Its architecture is composed of predesigned modules that can be reused as Intellectual Property (IP) blocks. These modules follow a common design template, making it possible to compose the architecture, tailoring the system to specific application needs, such as the number and type of compute resources, the structure of the memory hierarchy, and the communication infrastructure.

In general, DRRA presents a flexible and efficient solution that addresses the limitations of traditional architectures. It helps to close the gap between flexibility and performance while supporting faster and more cost-effective hardware design. This makes it especially valuable during the early stages of system development, where architectural choices have a significant impact on final performance and implementation costs [9].

3.1.1 Architecture

The DRRA is a coarse-grain spatial computing fabric designed to be flexible and scalable, with the ability to dynamically reconfigure its internal resources based on the specific requirements of the mapped application. The fabric is organized as a matrix of DRRA cells, each interconnected via a nearest-neighbor Network-on-Chip (NoC) connection scheme [29]. In Figure 3.1 the structure of the fabric and the interconnections between cells are shown.

Each DRRA cell consists of a single controller, also referred to as sequencer, and a set of 16 resource slots that serve as placeholders for DRRA components. The controller is responsible for configuring and managing the internal components of the cell, issuing instructions to the resources, and controlling their operation during runtime. The slots are initially unoccupied and provide a function implementation space, an instruction decoder, and a standardized set of ports, forming the template on which all DRRA components are built.

DRRA components constitute the basic building blocks for vector processing. Typical resources include Register Files (RFs), SRAMs, Input/Outputs (IOs), and DPUs, but additional application-specific blocks can also be integrated. For

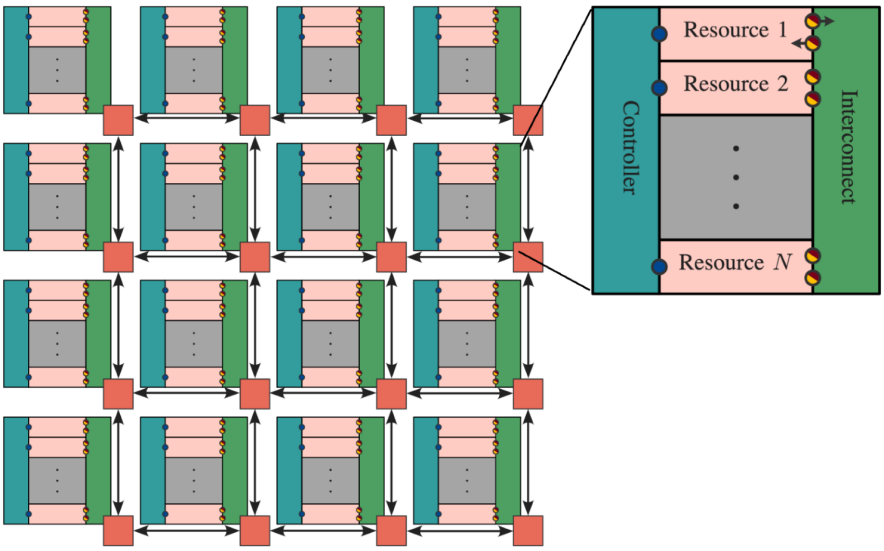


Figure 3.1: Architecture of DRRA-2 fabric [29]

instance, in our work, specialized FFT computation units have been designed and instantiated within the fabric. The seamless integration of different resources is enabled by a standardized interconnection interface, which ensures compatibility among components within a cell. The standard component interface, shown in Figure 3.2, consists of four input/output data ports, divided into word-level and bulk-level ports, typically 16 bits and 256 bits wide, respectively. Word-level ports are used for communication within a cell, while bulk-level ports handle communication between neighboring cells.

Figure 3.2: Port interface of a DRRA component

A resource can occupy one or more contiguous slots depending on its size, which is determined by factors such as the number of ports and the complexity of its function logic [29]. Designers can instantiate any supported resource, such as DPUs, RFs, or SRAMs, making the architecture highly adaptable to a wide range of application requirements.

3.2 Composable Instruction Set

The DRRA architecture adopts the CIS model, which is a resource-centric approach to instruction design [11]. Unlike traditional instruction-centric models that focus on controlling a central processing unit, the composable model targets individual hardware resources, such as interconnects, storage units, or computation modules. Complex operations are therefore built from the collaboration of many small, independent micro-threads, each controlling a specific hardware resource.

Each Pseudo-Assembly (PASM) instruction is associated with a particular resource, and the coordinated execution of multiple instructions produces the desired complex operations. A controller controls each cell, coordinating the activity of its resource slots and providing fine-grained control over resource behavior and communication.

The DRRA Instruction Set Architecture (ISA) is divided into two categories: controller instructions and resource instructions [29]. The cell-level sequencer decodes controller instructions, which manage the configuration and operation of the components in the cell. Resource instructions, on the other hand, are forwarded by the controller to a specific slot within the cell. Each slot contains a simple local decoder that only needs to recognize the instruction types relevant to the resource it hosts. This two-stage decoding process greatly reduces the complexity of the instruction logic [10], as the controller only performs basic dispatching, while the slot-level decoder executes the instruction.

Instructions in the DRRA are 32 bits wide and are structured as shown in Table 3.1. The `type` field indicates whether the instruction is a control or resource instruction, defining how it will be decoded. The `opcode` specifies the operation and enables the necessary hardware resources. For resource instructions, four bits indicate the slot number, allowing the sequencer to route the instruction payload to the corresponding resource. The remaining bits form the `payload`, containing the specific instruction content used by the target component.

type	opcode	slot	payload
31	30:28	27:24	23:0

Table 3.1: ISA format of the CIS used in the DRRA

3.2.1 Controller instructions

Two key controller instructions in the DRRA are `wait` and `act`. These instructions are used to control the execution of resources by activating them or imposing timing delays. They ensure proper synchronization of tasks and, consequently, the correct execution of the program. Unlike resource instructions, controller instructions are

automatically inserted by the compiler according to the timing constraints specified in the PASM program.

The `wait` instruction, summarized in Table 3.2, uses the `mode` field to define the event that triggers its completion. In mode 0, the instruction waits for a specific number of cycles, while in mode 1 it waits for one event, represented using a 1-hot encoding of event slots. The `cycle` field specifies the number of cycles to wait when mode 0 is selected.

mode	cycle	
28	27:1	

Table 3.2: `wait` instruction

The `act` instruction, shown in Table 3.3, enables the Finite State Machine (FSM) of the target port, activating the resource. The first field of the instruction specifies the ports to activate using 1-hot encoding. The `mode` and `param` fields define the activation pattern, allowing multiple ports to be enabled simultaneously. Not all combinations of ports are valid, so careful slot assignment is required when mapping the application.

ports	mode	param	
28:13	12:9	8:1	

Table 3.3: `act` instruction

3.2.2 Spatial and temporal composability

The DRRA's CIS model supports both spatial and temporal composability. Spatial composability allows complex operations to be constructed from multiple simple, atomic operations distributed across different hardware resources. Temporal composability, on the other hand, enables complex tasks to be built by combining simple instructions over time [11]. This is achieved through operators such as repetition (R), which defines loops, and transition (T), which inserts delays between instruction blocks. These operators allow the DRRA fabric to efficiently accelerate both simple loops and complex nested loops by modeling FSM-like transition patterns. In the CIS, the repetition and transition operators are implemented through two dedicated instructions: `rep/repx` and `fsm`.

The `rep/repx` instruction is used together with resource instructions to repeat operations according to multi-level loop sequences. Its fields, shown in Table 3.4, define the loop level, the number of iterations, the iteration step, and the delay between repetitions.

port	level	iter	step	delay	
24:23	22:19	18:13	12:7	6:1	

Table 3.4: `rep/repx` instruction

The `fsm` instruction is also associated with other resource instructions and triggers state transitions among a maximum of four resource configuration. Its fields define the delay between consecutive stages, as shown in Table 3.5.

port	delay_0	delay_1	delay_2	
24:23	22:16	15:9	8:2	

Table 3.5: `fsm` instruction

This instruction model has important implications for both hardware and compiler design. In hardware, each slot only needs to decode one or two simple instruction types, which simplifies control logic and reduces power consumption. Control units can also be placed physically close to the resources, reducing wire lengths and improving energy efficiency. On the compiler side, the model requires cycle-accurate instruction scheduling. The compiler must carefully order each PASM instruction to ensure synchronization between resources. Although this increases compiler complexity, it results in highly efficient execution [11].

3.3 DRRA component library

The DRRA fabric is assembled by filling the free slots of its cells with the resources available in the `drra-components` library, following the fabric architecture description file. The library contains the files for each available component. These include RTL descriptions, the architecture description file (`arch.json`), the resource-specific ISA file (`isa.json`), as well as support files such as C models for simulation and Bender files that define the hierarchy and dependencies.

The following subsections present some of the basic components available in the library, highlighting their design, operations, and PASM instructions.

Switchbox

The Switch Box (SWB) manages intra and inter-cell communication by creating interconnections between internal components and allowing routing to other cells. The SWB must always be placed in slot 0, as it has a special interface with 16 word-level ports, which ensure connectivity between all slots. Word-level channel connections are configured with the `swb` instruction. This instruction allows defining

the source and target slots between which a channel is established and supports up to four configuration options that can be switched at runtime using the `fsm` instruction, providing flexible and efficient connection reconfiguration.

option 24:23	channel 22:19	source 18:15	target 14:11	

Table 3.6: `swb` instruction

For bulk-level communication, each cell has four connection interfaces, one for each adjacent cell. The `route` instruction is used to configure the connections between bulk-level IO ports and intra-cell bulk channels. The `sr` field specifies whether the connection is for sending or receiving. On the send side, `source` indicates the slot number, while `target` is the 1-hot encoded direction, as depicted in Figure 3.3. On the receiving side, the roles are reversed, with the direction specified in the `source` and the slot in the `target` field. A single bulk intra-cell connection between two components is supported by setting both the target and the source to 4, indicating the "self" cell.

option 24:23	sr 22	source 21:18	target 17:2	

Table 3.7: `route` instruction

8 NW	7 N	6 NE
5 W	4 self	3 E
2 SW	1 S	0 SE

Figure 3.3: `route` direction configuration

Input/Output SRAM

Another important component is the IOSRAM. It serves as the main storage unit of DRRA enabling data transfers between the IO buffers and the fabric. IO buffers serve as external memory elements that provide test data. In synthesis, their

implementation may take the form of First-In First-Outs (FIFOs), RFs, or SRAM-based memory structures, depending on the application requirements.. They are connected to DRRA through the bulk-level input/output data ports of the fabric. Typically, the input buffer is connected to the top row of the fabric and provides the input data. The output buffer instead is connected to the bottom row of the fabric and is used to store the output data of the algorithm. As the SWB, also the IOSRAM must be placed in a predetermined slot in the cell, slot 1, which is the only one provided with an IO interface.

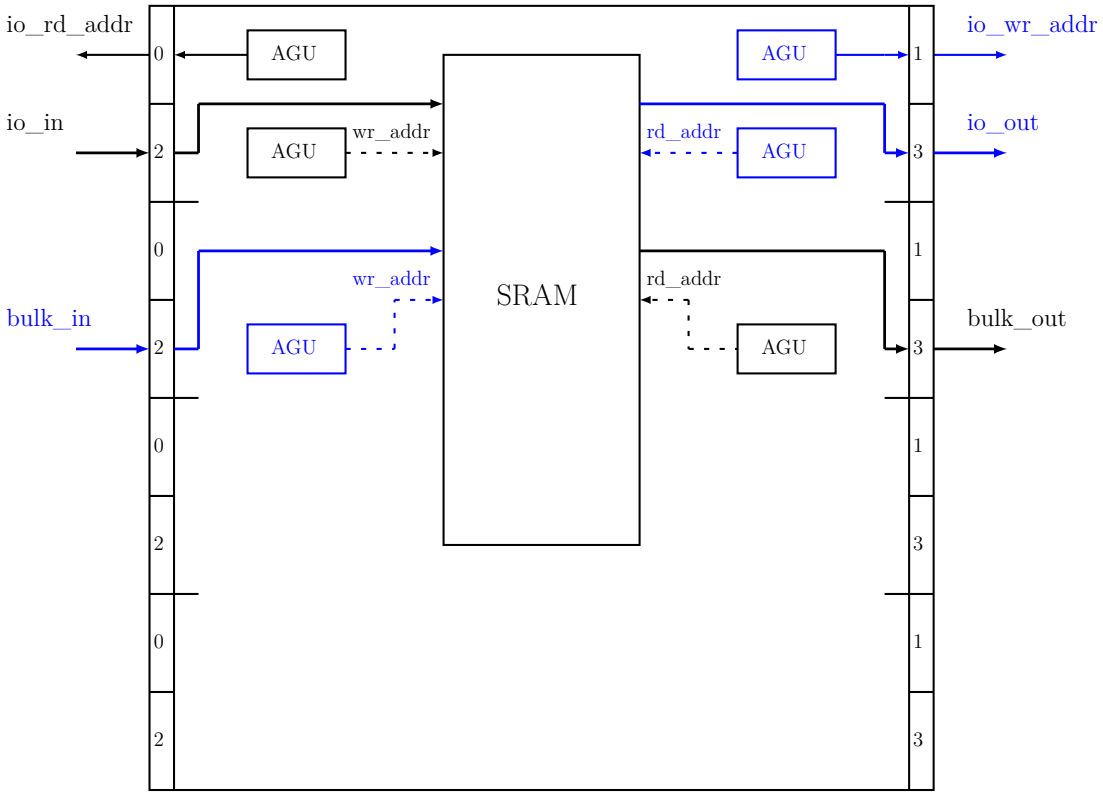


Figure 3.4: IOSRAM component. In black, the top cell path, in blue the bottom cell path

The component occupies four slots, primarily due to the large area required by the SRAM, although only a limited number of ports are actively utilized. Because read and write operations with the IO buffer depend on the specific cell in which the component is instantiated, three slightly different implementations are provided: one for reading data from the buffer, one for writing data to the buffer, and another supporting both interfaces. Figure 3.4 shows the schematic of the component, where the two interface types are highlighted.

The ports in the first slot manage IO communication with the buffers, while the

bulk-level ports in the second slot handle intra-cell data transfers. Word-level ports are used to transmit read/write addresses to the IO buffer, while bulk channels facilitate actual data exchange. Each port is equipped with an Address Generation Unit (AGU), responsible for generating addresses used to access the SRAM or transmitted via the word-level ports to the IO buffer. Each AGU is configured using the `dsu` instruction. It is used to set the initial address, which can be static or dynamic (defined via the `sd` field), and enables the configured ports. Complex address sequences are generated by combining `dsu` with the `rep/repx` instruction, as described in Section 3.2.2.

sd	init_addr	port	
24	23:8	7:6	

Table 3.8: `dsu` instruction

Register File

The RF component provides local storage within computational cells, enabling fast access for the DPU and other computational resources. The component occupies one slot and supports configurable word length and depth. The word length corresponds to the global parameter `WORD_BITWIDTH`, while the depth can be defined in the `arch.json` description file, in a range between 16 and 256 words.

Two word-level ports are used for read/write operations between resources in the cell, while the two bulk-level read/write ports are used to efficiently transfer data between cells, typically between the RF and the IOSRAM. Each port of the RF includes an AGU that generates addresses based on the access pattern specified using the `dsu` and `rep/repx` instruction combination as seen for the IOSRAM component.

Data Path Unit

The DPU is the core computational resource for most DRRA applications. It occupies two slots, allowing operations with two inputs and one output to be performed. It supports up to 32 different arithmetical and logic operations, defined by the `mode` field of the `dpu` instruction. Common arithmetical operations include `add`, `mult`, `mac`, `div`, and `exp`. Some of them require an immediate value, that is defined in the correspondent instruction field.

Loop behavior is defined using the `rep/repx` instruction, while up to four configuration options can be defined using the `option` field, and then switched at runtime using the `fsm` instruction. This combination enables fast and efficient computation for applications requiring multiple operation types.

option	mode	immediate	
24:23	22:18	17:2	

Table 3.9: `dpu` instruction

3.4 Vesyla toolchain

The methodology adopted in this work is based on the Vesyla toolchain, a domain-specific High-Level Synthesis (HLS) system targeting the DRRA. Vesyla translates high-level algorithmic descriptions into optimized instruction schedules and mappings for DRRA fabrics, providing a coherent methodology for fabric assembly, program compilation, and design simulation.

3.4.1 PASM compilation

The flow begins with the initialization of a DRRA-style project in Vesyla, containing all the necessary files to begin the implementation process. Among these, the `arch.json` is of particular importance, as it contains the description of the target architecture. This file is used to define the resources included in the DRRA fabric for the specific application needs. Another essential file is `main.cpp`, which provides a template for defining the functions that implement the algorithm logic. The final key component is the `pasm` folder, which holds the PASM code which forms the low-level representation of algorithms.

The structure of PASM programs reflects the dataflow-oriented nature of the DRRA. At the highest level, regions such as `loop`, `cond`, and `epoch` define the iteration and control structures of the program. Within `epochs`, the programmer specifies operation regions that can be of three types: resource operations (`rop`), such as register file reads or writes; control operations (`cop`), which are managed by the cell controllers; and constraints (`cstr`), which explicitly define timing and structural dependencies between `rops`. These constraints guide the scheduler in translating the high-level program into a realizable instruction timeline, ensuring that both data dependencies and resource availability on the DRRA fabric are respected. An example of a PASM program can be found in Appendix 7.1.

To translate algorithmic descriptions onto DRRA, Vesyla introduces the Control Address Data-Flow Graph (CADFG) as its central intermediate representation. Unlike traditional control-data flow graphs, CADFG explicitly separates control flow, data flow, and address computations. This decomposition permits efficient mapping of algorithmic structures to DRRA resources. CADFGs are refined further into Instruction Dependent Graphs (IDGs), which introduce edges for both data transfer and temporal dependencies, bridging the gap between algorithmic abstraction and instruction-level scheduling. A key step in this refinement is

dependency analysis, which identifies hazards such as Write-After-Write (WAW) and Write-After-Read (WAR) that could affect the correctness of vector operations. Scheduling builds upon this dependency analysis to produce correct and efficient instruction timelines. The result is a set of scheduled instructions that respect both temporal and resource constraints, optimized for the DRRA architecture.

3.4.2 Assemble the DRRA fabric

Once algorithms are compiled through the Vesyla flow, the DRRA fabric must be assembled to provide a simulation platform for deployment and verification. This process translates a user-defined architectural description into a fully realized hardware model, including both the ISA and Register Transfer Level (RTL) representations. It is implemented via a CMake-based build hierarchy that automates resource gathering, model compilation, and simulation setup.

The foundation of the assembly process is the `arch.json` file. It specifies the desired fabric configuration, detailing the resources to instantiate, their arrangement in the cells, and their specific parameters. Upon execution of the assembly command, Vesyla elaborates this description, expanding it with structural information of each component to create an elaborated architectural description file. A similar process is applied to the `isa.json` file, collecting all supported instructions from the instantiated components and generating the complete set of instructions available on the fabric, providing a comprehensive view of its computational capabilities.

The final phase of the assembly process is the generation of the RTL description, consisting of a collection of SystemVerilog files that describe the hardware implementation of the fabric instance. The elaborated `arch.json` guides this process, ensuring that the RTL description accurately reflects the structural and functional characteristics of the assembled fabric. The output is a set of SystemVerilog files that can be synthesized and simulated to verify the hardware design.

During the build phase, Vesyla also compiles Structural Simulation Toolkit (SST) behavioral and timing models of the DRRA components. These models are used by the instruction-level simulator to enable cycle-accurate simulation and by the instruction scheduler to ensure correct timing during compilation of the PASM program.

Finally, the algorithm is simulated on the assembled DRRA fabric. Input data are provided through the input buffer, and compiled PASM instructions are issued to each corresponding cell. After computation, the results are stored in the output buffer and written to an output memory file, which is then compared with results obtained from the software implementation of the algorithm. This process verifies the correctness of the algorithm execution on the DRRA platform.

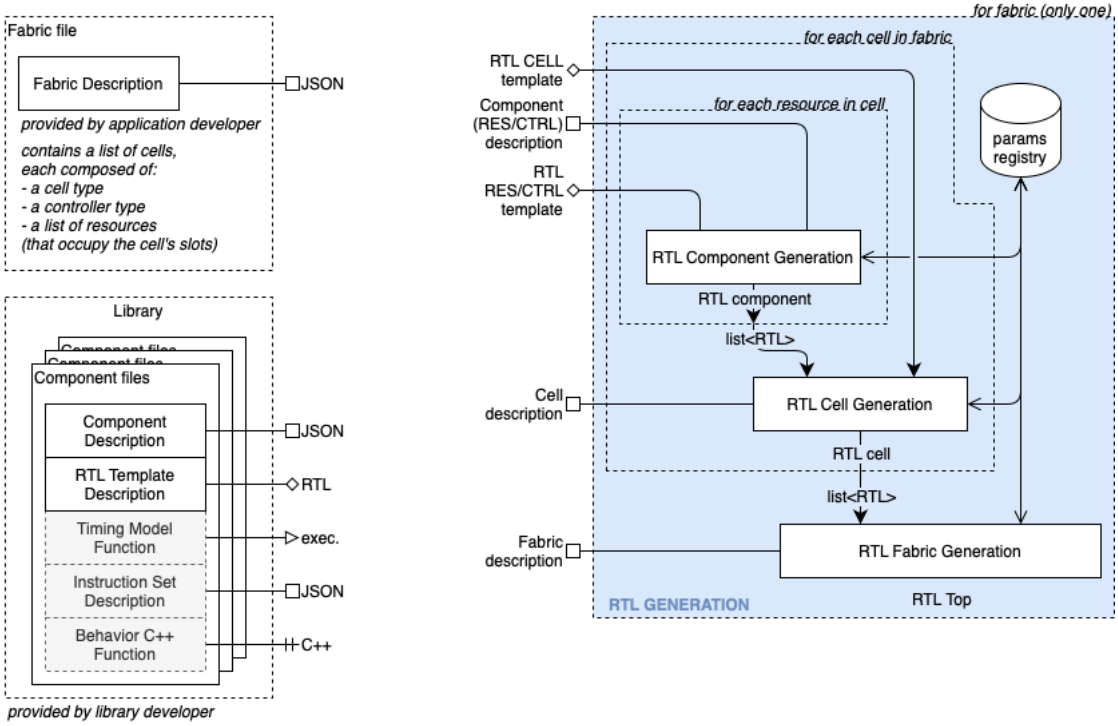


Figure 3.5: Component library structure and assemble workflow of the Vesyla toolchain [29]

Chapter 4

Implementation

This chapter presents the library components developed for FFT acceleration on DRRA. The proposed solutions have been designed and tested to meet the performance requirements of the mmWave FMCW radar application, while exploiting the advantages of the DRRA platform and addressing its inherent constraints.

The implemented FFT accelerator adopts a memory-based architecture and is built to support DIF radix-2 and radix-4 butterflies. Particular emphasis has been placed on the memory organization within the system, as memory represents a key element for efficient FFT computation. The design leverages existing DRRA components, exploiting their specific characteristics, while additional resources have been introduced to support the FFT functionality.

The developed components are presented in detail, starting from their theoretical foundation and progressing to the design decisions made during implementation. An overview of the complete FFT accelerator is then provided, covering the top, middle, and bottom cells. Finally, the developed PASM program is introduced, illustrating how the design is mapped and executed on the DRRA platform.

4.1 DRRA2 FFT architecture

The proposed solution relies on a RF to store the temporary FFT data during the computation. This approach offers several advantages, such as efficient access to individual memory words, flexibility in the number of ports, and a simplified read/write protocol. However, it also presents a significant limitation due to the restricted maximum number of registers. Single-word access requires complex address decoding hardware, making deeper memory implementations costly. Despite this limitation, the maximum depth of the RF, as defined in the synthesis macros, is 256 registers. This depth is sufficient to meet the requirements of the targeted application, which is based on a 256-points FFT.

4.1.1 Register File

The `rf_fft_r4` module implements the address generation and memory management unit for the FFT computation. Its design is based on the `rf` component already available in the component library, described in Section 3.3. The module, whose schematic is shown in Figure 4.1, provides multiple interfaces for word-level memory access, supports the bit-reversed addressing required by the FFT algorithms, and integrates several AGUs specialized for radix-2 and radix-4 butterfly operations. It consists of four slots, each provided with the standard interface that includes clock, reset, instruction, and enable signals, along with data ports. This structure enables four parallel and independent access to the register file from the BUs, thereby supporting efficient and high-throughput butterfly computations.

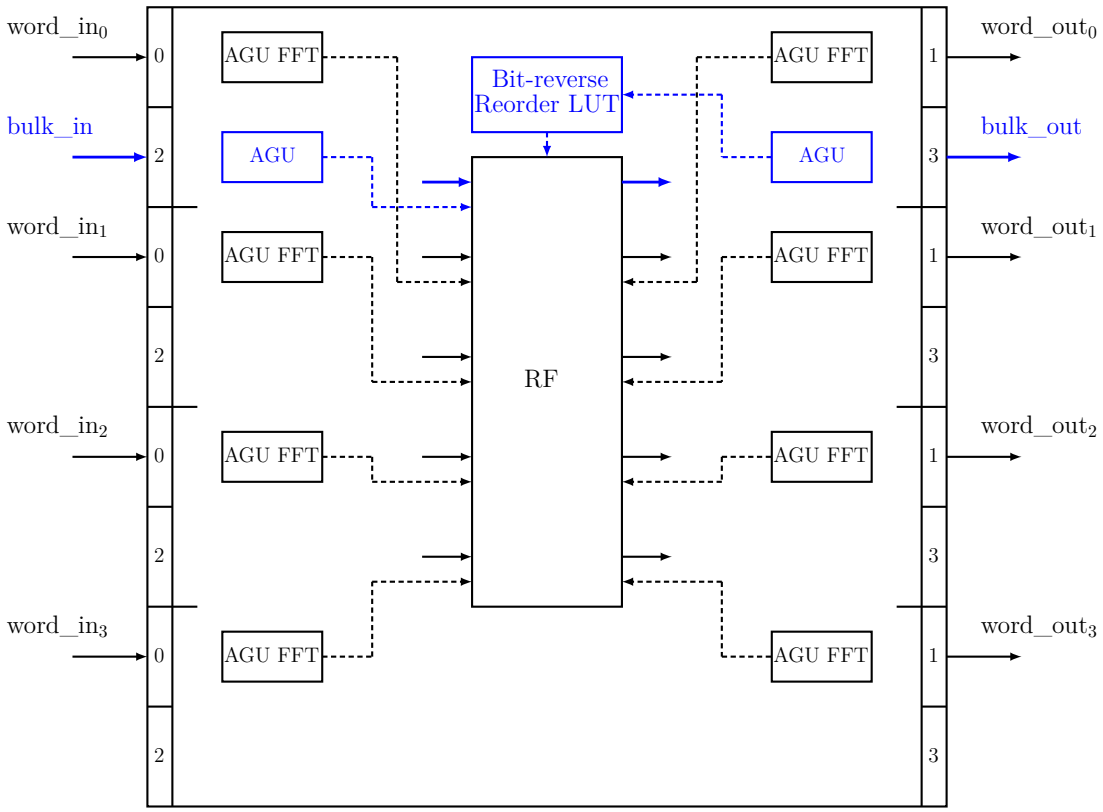


Figure 4.1: Block diagram of the FFT Register File

The register file is organized as a two-dimensional array with parametrisable bit width and depth. Each FFT point is a complex number consisting of a real part and an imaginary part. Both parts are stored in a single 32-bit register, with the first 16 bits corresponding to the real part, and the remaining 16 bits representing the imaginary part. To achieve an adequate resolution of the final result while

limiting power consumption and chip area, fixed-point representation has been adopted. The real and imaginary parts follow the $Q_{x.y}$ format, where x indicates the number of integer bits and y the number of fractional bits, whose value can be adjusted through a parameter. In this representation, `FRAC_BITWIDTH` bits are assigned to the fractional part, while the remaining 16−`FRAC_BITWIDTH` bits are reserved for the integer part.

Memory is updated synchronously on each clock edge, while combinational logic handles read and write arbitration. Bulk write operations are given priority, and mechanisms are in place to prevent simultaneous writes to the same address, ensuring correct and conflict-free memory access. Activation signals, `activate_n`, selectively enable the FSMs within each port, initiating the corresponding read and write operations.

The four word-level input/output ports of the component are used to supply and receive data to/from the BUs, enabling the execution of two radix-2 or one radix-4 computations per clock cycle. The design instantiates specialized AGUs to generate the appropriate FFT read and write address sequences. These sequences are determined on the basis of the parameters that are defined using the dedicated `fft` instruction, shown in Table 4.1.

port	n_points	radix	n_bu	mode	delay	
24:23	22:11	10:9	8	7	6:2	

Table 4.1: `fft` instruction

This instruction defines the port to configure and sets the number of points and radix of the current FFT computation, along with the number of butterflies (if radix-2 is used), the AGU operation mode and the delay between consecutive read and write operations. These settings are then used to determine the port index of the BU and thus generate the correct address sequence. This close integration between instruction and AGU makes the `rf_fft_r4` module highly reconfigurable, allowing it to adapt to different FFT sizes and radices. For bulk-level data transfers between RF and SRAM, only the bulk-level ports in slot 0 are used, while the bulk ports in the other slots remain disabled. In this case, the general purpose AGU component explained in Section 3.1.1 is used.

As discussed in Section 2.1.1, FFT algorithms require bit-reversed ordering of the data at the start or at the end of the computation, depending on the decimation. In this module, which implements the DIF algorithm, bit reversal is necessary after the final stage, when the data are written back to SRAM. This operation is performed using two dedicated look-up tables, `LUT_init` and `LUT_init_r4`, pre-initialized with the correct address sequences for radix-2 and radix-4 FFT, respectively. During bulk reads, the selected look-up table provides the bit-reversed

address sequence, which is used to access single words in the RF and create the bulk data packet.

4.1.2 FFT Address Generation Unit

The `agu_fft_r4` module provides a rotation-based scalable, radix-flexible address generation scheme for the FFT accelerator. It is responsible for controlling read and write access to the `rf_fft_r4` register file, as well as generating the address sequence for the TFG. The design is composed of a top-level AGU module and two supporting submodules: `mux_rotator_r4` and `twiddle_addr_r4`. Together, these submodules produce the correct address sequences according to the radix, the BU configuration, and the current FFT stage.

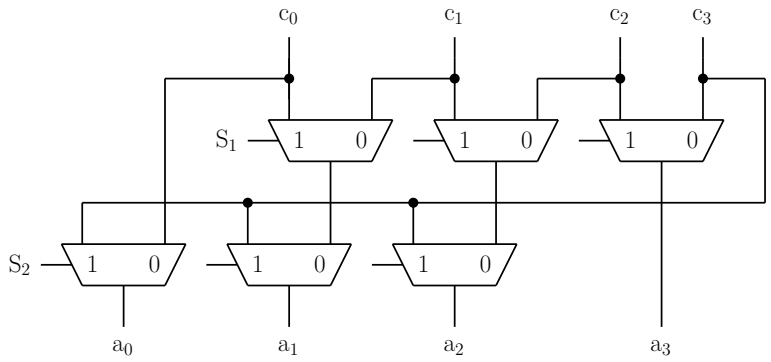
The AGU is controlled by an FSM that works in conjunction with internal counters to sequence the address generation process. The FSM cycles through three states: `IDLE`, `ADDR`, and `DELAY`. In the `IDLE` state, it waits for an activation signal, which is triggered by the `act` instruction issued by the controller. Once activated, the FSM loads all configuration parameters, obtained from the `fft` instruction, into internal registers. The `DELAY` state is used to insert instruction-configurable delays between successive address generations, allowing precise timing control for data movement.

The addressing sequence is primarily managed by internal counters, which track the current FFT stage and the address within the stage. The counter values are adjusted according to the transform length, the radix, and the number of butterflies in use. Based on the main address counter, the two submodules, `mux_rotator_r4` and `twiddle_addr_r4`, generate the addresses for the FFT data and the corresponding twiddle factor addresses, respectively. Depending on the mode specified by the `fft` instruction, the module then selects one of these outputs as the final address.

Mux Address Rotator

The `mux_rotator_r4` module is responsible for generating memory addresses for FFT data samples. Its operation is based on rotations of bits of the address counter received from the top-level module. This transformation maps sequential addresses into the required access patterns, allowing data to be read from the correct memory locations and passed to the BU and vice versa, throughout the computation stages. The core of the module consists of two cascaded rows of multiplexers, as illustrated in Figure 4.2.

The design supports both radix-2 and radix-4 operations using the same hardware. The number of multiplexers depends on the FFT size N and the radix r, corresponding to $2(\log_r N - 1)$ multiplexers. The circuit rotates groups of k bits,



Stage	S_1	S_2
1	000	000
2	001	001
3	011	010
4	111	100

Figure 4.2: Circuit used for address rotation for a 16-point radix-2 FFT and its corresponding selection signals.

where k is determined by the radix ($k = 1$ for radix-2 and $k = 2$ for radix-4). The upper row of multiplexers selects the length of the bit field to rotate, while the lower row routes the LSB to the correct output. Since the required rotation depends on the current FFT stage, the selection signals S_1 and S_2 for the upper and lower rows must be calculated accordingly, as reported in Table 4.2. The rotation operations for each stage of a 16-point radix-2 FFT are illustrated in Figure 4.3.

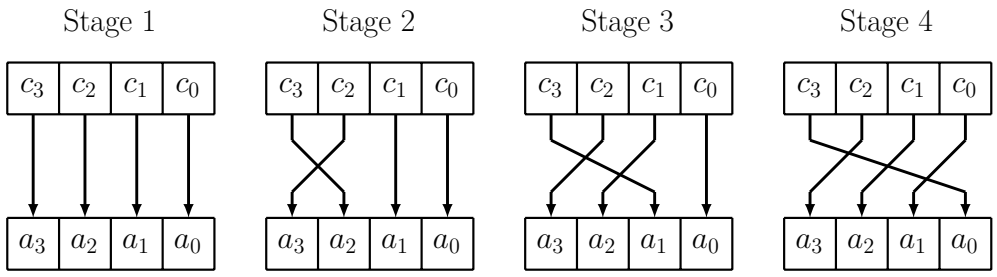


Figure 4.3: Rotation operations for each FFT stage

Each `rf_fft_r4` data port is associated with specific BU input and output ports, so each `mux_rotator` must generate the appropriate address sequence for its designated port. The `port_index` and `bu_index` are calculated in the top module and passed to the respective AGUs. For radix-2, the input sequence is divided into two parts when a single butterfly is used, or four parts when two butterflies are active. In radix-4, the input sequence is always divided into four equal parts. Table 4.2 shows the generated address sequence for a 16-point FFT with a single radix-2 butterfly, demonstrating how the sequence evolves according to rotation operations in Figure 4.3.

Address Counter ($c_3c_2c_1c_0$)	**Address Sequence**			
	Stage 1	Stage 2	Stage 3	Stage 4
0000	0000	0000	0000	0000
0001	0001	0001	0001	0010
0010	0010	0010	0100	0100
0011	0011	0011	0101	0110
0100	0100	1000	1000	1000
0101	0101	1001	1001	1010
0110	0110	1010	1100	1100
0111	0111	1011	1101	1110

Table 4.2: Address sequence generation for a radix-2 butterfly in a 16-point FFT (port 0)

Twiddle Address Generator

The `twiddle_addr_r4` module is responsible for generating addresses for the twiddle factors, which are used by the TFG component to provide the BU with the appropriate rotation coefficients at each clock cycle. The design supports both radix-2 and radix-4 butterflies, with addresses computed through a combination of shifting and masking operations applied to the address counter.

In the radix-2 case, each butterfly requires a single twiddle factor. The generator first adjusts the address to account for the number of butterflies within the BU. If a second butterfly is present, the counter max value is halved and an additional offset of $N/4$ is applied. This offset arises because, after the butterfly operation, only half of the FFT points require non-trivial twiddle factors, and this subset is further partitioned among the active butterflies. Once the adjustment is complete, the address is left-shifted and masked according to the current FFT stage.

For radix-4, each butterfly uses a set of three twiddle factors. The generator begins by masking the input address counter to compute a stage-dependent base address. This base address is then offset by the `port_index` of the butterfly, which selects the local twiddle factor within the three-element set. Finally, the resulting value is shifted by a number of bits determined by the current FFT stage.

To better illustrate the address sequence, Table 4.3 reports the generated twiddle factor addresses for a 16-point FFT in radix-2 mode, according to the FFT dataflow shown in Figure 2.3. The sequence highlights how the masking and shifting operations evolve across stages, ensuring that the correct coefficients are accessed at each step.

Address Counter ($c_3c_2c_1c_0$)	Twiddle Address Sequence			
	Stage1	Stage2	Stage3	Stage4
0000	000	000	000	000
0001	001	010	100	000
0010	010	100	000	000
0011	011	110	100	000
0100	100	000	000	000
0101	101	010	100	000
0110	110	100	000	000
0111	111	110	100	000

Table 4.3: Twiddle factor address generation for a 16-point DIF radix-2 FFT

4.1.3 Twiddle Factor Generator

The `tfg` module provides the complex exponential coefficients required in the FFT computation to combine partial results across stages. As discussed in Section 2.3.2, different design options are available for implementing a twiddle factor generator, mainly depending on memory usage and power constraints. In this work, a ROM-based TFG has been implemented, since the maximum FFT size to be supported is relatively small.

The design principle relies on the definition of twiddle factors. These coefficients, given by Equation 2.8, are complex numbers representing an angle in the unit circle through its sine and cosine values. In an N-point FFT, the angles are obtained by dividing the unit circle into N equal parts. An important observation is that the set of twiddle factors for an $N = 2^m$ FFT already contains all the coefficients needed for any smaller $N = 2^p$ FFT, where $m > p$. This makes it sufficient to store only the coefficients for the largest supported FFT size, reducing memory overhead.

In radix-4 FFTs, however, up to $3N/4$ twiddle factors are required, which could lead to high memory costs if all values were stored directly. To address this, the implementation exploits the symmetry properties of sin and cos, since the values of all angles on the unit circle can be derived from those contained in the first $\pi/8$ interval. Thus, only these base values need to be stored in memory, while the others are reconstructed by applying sign changes or swapping the real and imaginary components.

The proposed TFG relies on a ROM of size $N_{max}/8 + 1$, corresponding to $32 + 1$ complex coefficients of 32 bits each. The module is organized into three parallel slots, each equipped with an AGU and a ROM, as shown in Figure 4.4.

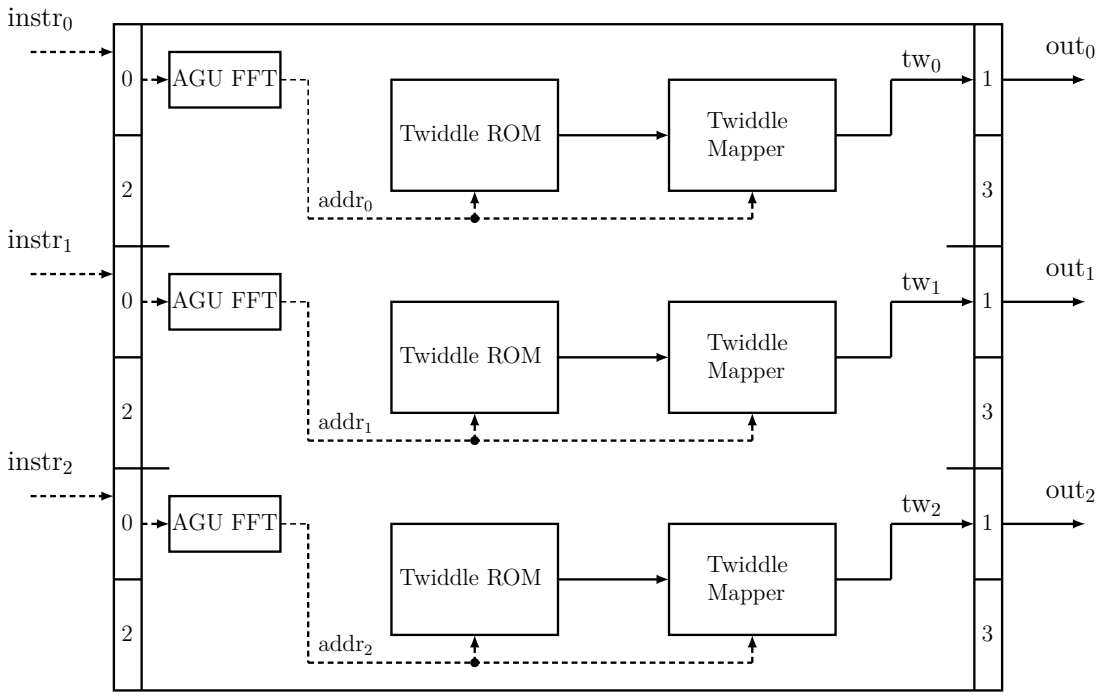


Figure 4.4: Block diagram of the Twiddle Factor Generator

This structure allows the simultaneous generation of three independent twiddle factor streams, which are required by radix-4 butterfly operations. The address generation for the ROM is handled by the same `agu_fft_r4` module described in Section 4.1.2, here operating in twiddle factor address mode. Alongside the AGU, two additional submodules, `twiddle_rom` and `twiddle_map`, are used to retrieve the stored values and apply the required transformations.

The component is configured through the `fft` instructions reported in Table 4.1. Each of the three data paths can be configured independently and supports different combinations of radix and number of butterflies. Once the activation signal is asserted, the AGU generates the addresses of the required twiddle factors for the selected port. The address is an 8-bit word, of which the five least significant bits r directly address the ROM, while the three most significant bits q identify the sector of the unit circle. To ensure correct alignment across FFT sizes, the generated address is shifted according to the `n_points` parameter. The retrieved base coefficient is then passed to the `twiddle_map` submodule, which applies the symmetry transformations dictated by q. The final coefficients are stored in pipeline registers for synchronization with the butterfly stage before being driven to the outputs.

Twiddle factors ROM

The `twiddle_rom` submodule implements a compact memory that stores the unique rotation coefficients of the first $\pi/8$ interval. The input address from the `agu_fft_r4` cannot be used directly, as the sin and cos values of the correspondent angle in adjacent circle sectors are specular to each other. As illustrated in Figure 4.5, given that the saved coefficients correspond to the angles in sector 0, this property repeats in all odd-indexed sectors. Distinguishing whether specular addressing is needed is straightforward and consists of checking the q_0 bit. If $q_0 = 1$, the address is mirrored by applying a two's complement operation. The effective memory address is, therefore, given by:

$$\texttt{rom\_addr} = \begin{cases} r, & \text{if } q_0 = 0 \\ \sim r + 1, & \text{otherwise} \end{cases} \tag{4.1}$$

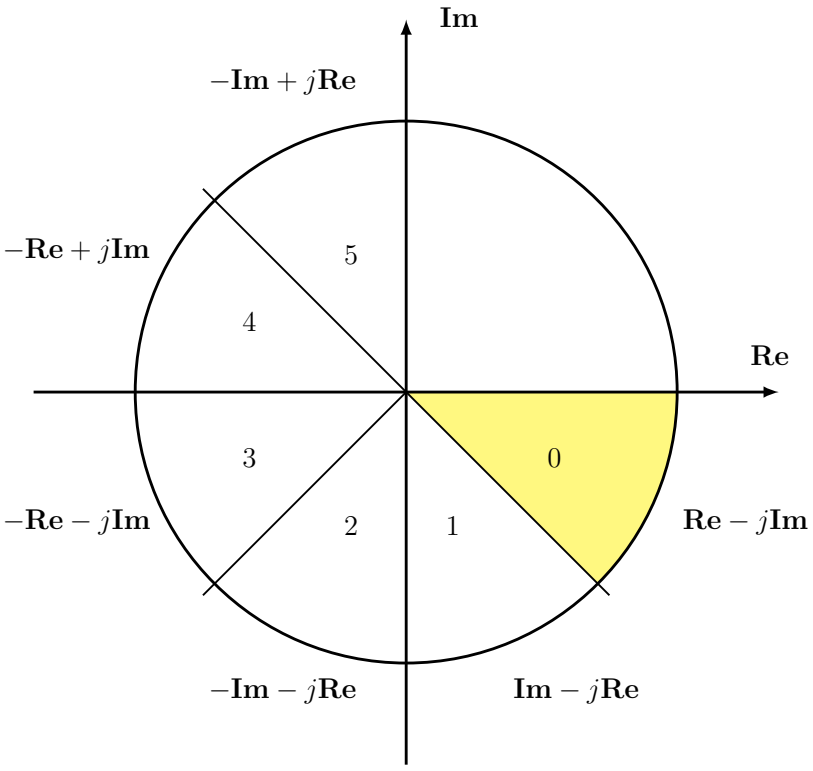
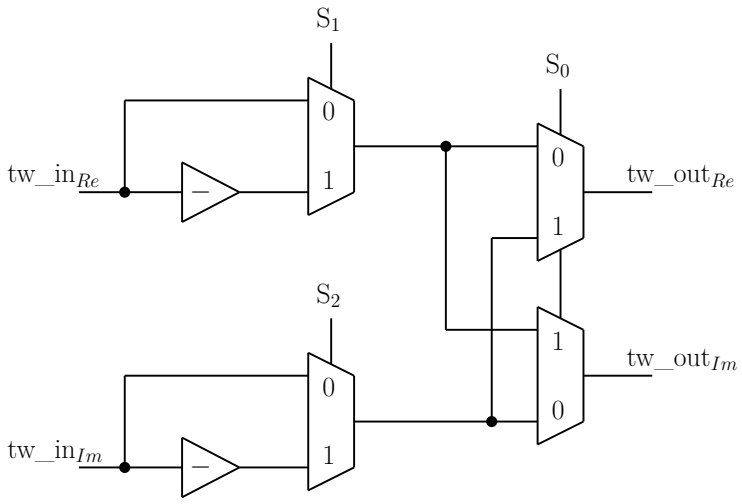


Figure 4.5: Twiddle factors sections with the symmetry transformations required

Twiddle factors mapper

The `twiddle_map` submodule reconstructs the complete twiddle set from the compact ROM contents. Reconstruction relies only on sign inversion and swapping between the real and imaginary parts. The MSB part of the address, q, indicates the section of the circle and, therefore, the operations that must be applied, as shown in Figure 4.5.

The remapping circuit, shown in Figure 4.6, consists of four multiplexers. Two of them select between the unchanged or two's complement versions of the real and imaginary parts, controlled by signals S_1 and S_2. The other two multiplexers handle swapping between real and imaginary components, both controlled by S_0. Table 4.6 lists the control signals required for each unit circle section.



Sec	$q_2q_1q_0$	S_0	S_1	S_2
0	000	0	0	0
1	001	1	1	1
2	010	1	1	0
3	011	0	1	0
4	100	0	1	1
5	101	1	0	0

Figure 4.6: Circuit used for twiddle factor remapping and corresponding selection signals.

4.1.4 Butterfly Unit

The `bu` module forms the computational core of the FFT accelerator and is designed to support both radix-2 and radix-4 butterflies in a flexible, parameterized manner. The component, shown in Figure 4.7, consists of seven slots, each equipped with the same input and output interface described in Section 3.1.1. These slots allow for input and output operations for different butterfly configurations. Specifically, slots 0 to 3 are dedicated to FFT data channels, while slots 4 to 6 are used for the input of twiddle factors.

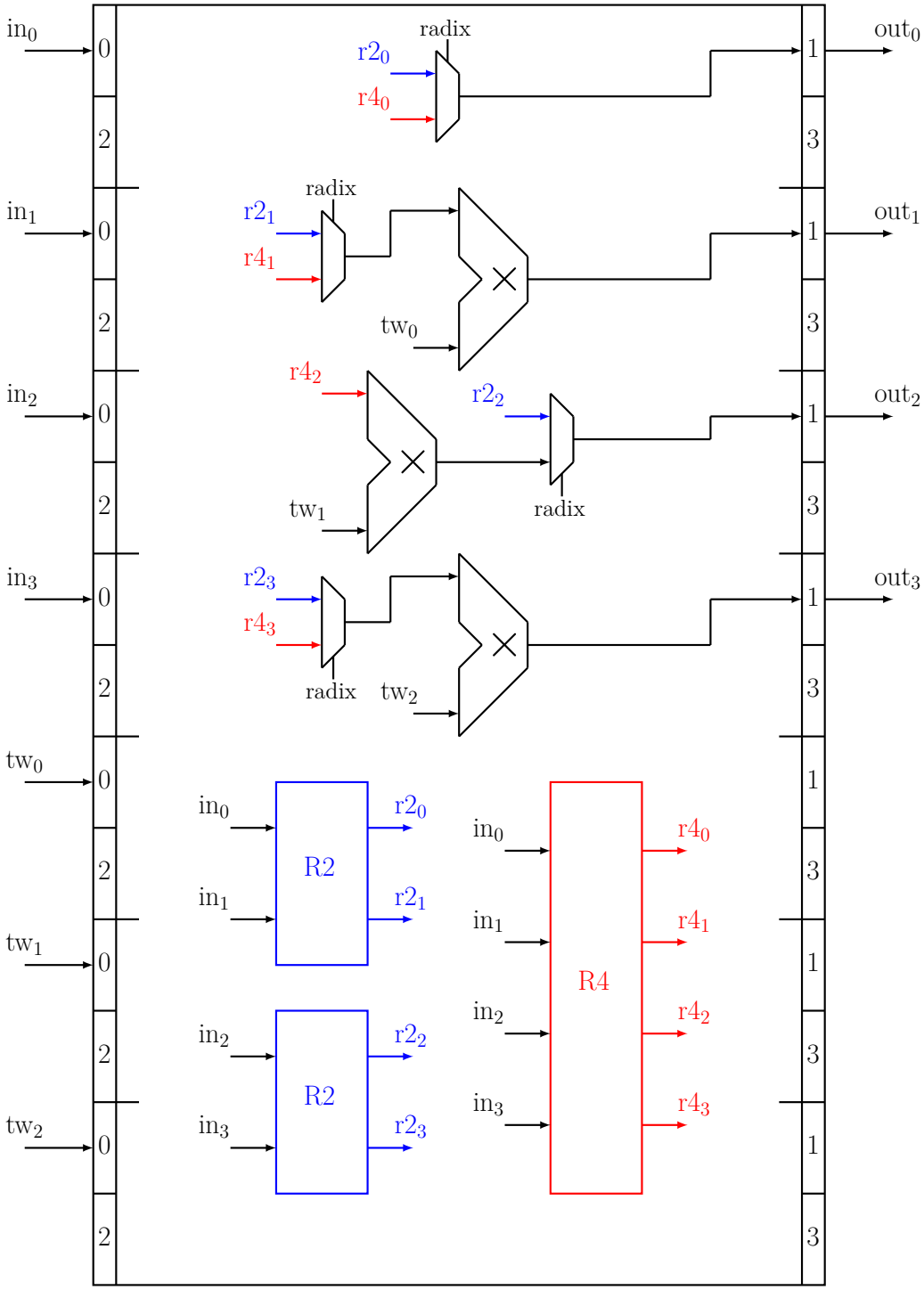


Figure 4.7: Block diagram of the Butterfly Unit

The arithmetic operations are carried out by dedicated radix-2 and radix-4 butterflies. The radix-2 butterfly operates on two input samples, producing sum and difference outputs as defined in Equation 2.10. Two parallel instances allow four input points to be processed simultaneously, fully utilizing the four data input ports and increasing throughput. The odd outputs of the butterflies are then multiplied by the twiddle factors to complete the calculation of the FFT stage. The radix-4 butterfly processes four input samples at the time, producing four transformed outputs according to the radix-4 algorithm defined in Equation 2.13. The outputs 1 through 3 are then multiplied by the twiddle factors. Three dedicated multipliers are instantiated, each receiving a butterfly output and the corresponding twiddle factor from input ports 4 to 6.

The routing of the signals to the appropriate butterflies and multipliers depends on the selected radix, which is provided to the module through the `bu` instructions. To ensure timing closure and support higher operating frequencies, intermediate butterfly results are stored in pipeline registers. These registers capture the results of the current cycle and provide stable inputs for the multiplier stage, maintaining throughput and computational efficiency.

The control logic of the `bu` block is instruction-driven. Each incoming instruction word is decoded to determine whether it corresponds to a butterfly operation or an FSM update. The `bu` instruction, shown in Table 4.4, sets the radix for the calculation, which is stored in the radix memory at the index specified in the `option` field. Configurations can be modified at run time using the `fsm` instruction, which drives the state evolution of the module, switching between different `bu` configurations according to the defined delays. This mechanism enables the accelerator to dynamically alternate between radix-2 and radix-4 execution, enhancing programmability and adaptability.

option 24:23	radix 22:21	

Table 4.4: `bu` instruction

Radix-2/4 Butterfly

The `radix2_bu` is a combinational block that implements radix-2 butterfly operations. It takes two complex inputs and produces two complex outputs by performing simple additions and subtractions. Each input is a signed fixed-point complex number packed into a 32-bit word. The real part occupies the upper half of the word, corresponding to `COMPLEX_BITWIDTH = WORD_BITWIDTH/2` bits, while the imaginary part occupies the lower half. The module is fully parameterized, allowing

flexibility in the word width (`WORD_BITWIDTH`), which automatically determines the width of the real and imaginary components (`COMPLEX_BITWIDTH`).

Each input word is split into real and imaginary components, and the butterfly operations are applied according to the equations:

$$\begin{aligned} out_0 &= in_0 + in_1 \\ out_1 &= in_0 - in_1 \end{aligned} \tag{4.2}$$

$$\begin{aligned} out_{0r} &= in_{0r} + in_{1r}, \quad out_{0i} = in_{0i} + in_{1i} \\ out_{1r} &= in_{0r} - in_{1r}, \quad out_{1i} = in_{0i} - in_{1i} \end{aligned} \tag{4.3}$$

This produces two signed fixed-point complex outputs, encoded in the same packed format.

The `radix4_bu` follows the same principle, but processes four inputs to generate four outputs according to the radix-4 Cooley-Tukey decomposition:

$$\begin{aligned} out_0 &= in_0 + in_1 + in_2 + in_3 \\ out_1 &= in_0 - j\,in_1 - in_2 + j\,in_3 \\ out_2 &= in_0 - in_1 + in_2 - in_3 \\ out_3 &= in_0 + j\,in_1 - in_2 - j\,in_3 \end{aligned} \tag{4.4}$$

Complex Multiplier

The `multiplier` module performs fixed-point complex multiplication, which is essential for applying twiddle factors in FFT accelerators. Each input is a packed signed fixed-point complex number, containing a real and imaginary part. The multiplier unpacks the inputs, performs the four required real multiplications, and reconstructs the output using the canonic formula for complex multiplication:

$$(a + jb) \cdot (c + jd) = (ac - bd) + j(ad + bc) \tag{4.5}$$

To support fixed-point arithmetic, truncation and rounding are applied after each multiplication. A fractional bit width parameter (`FRAC_BITWIDTH`) defines the binary point position, preserving the proper scaling across operations. Two's complement arithmetic is used to handle negative terms, such as $-bd$. The results are then recombined into a packed complex output.

The module also detects special rotation coefficients corresponding to $0°$ and $-90°$, which are used in both radix-2 and radix-4 algorithms. In the former case, the multiplication by 1 can be skipped entirely, since it leaves the value unchanged. This also resolves an issue of the $Q_{1.15}$ format, whose range extends from -1.0 to $+0.999969482$ and therefore cannot exactly represent 1. Because multiplications by 1 occur frequently during the FFT computation, even this small quantization

error could accumulate and degrade the accuracy of the final result. By skipping the multiplication, this problem is completely avoided. Multiplication by $-j$ can instead be implemented efficiently by simply changing the sign of the real part and swapping the real and imaginary components. In addition to improving numerical accuracy, these optimizations also help reduce power consumption by significantly reducing the utilization of the complex multiplier.

4.2 DRRA fabric

The final composition of the DRRA fabric for the RF-based FFT accelerator is illustrated in Figure 4.8.

As introduced in the previous sections, the DRRA architecture is organized into three rows of cells. The first row (Cell_0) integrates both a switch box and an IOSRAM, which together enable the acquisition of data from the input buffers and its temporary storage before transfer to the RF. The second row (Cell_1) is entirely dedicated to computation, with all resources allocated to FFT processing. Here, data are stored in the RF, supplied to the BU together with the twiddle factors generated by the TFG, and the results are written back into the RF. This process is repeated iteratively until the complete FFT is executed. The interconnections among the modules, as well as between adjacent rows, are managed by the SWB. The third row (Cell_2) mirrors the structure of Cell_0, but is instead used to transfer the processed data from the RF to the SRAM and finally to the output buffers. This arrangement ensures that data can be efficiently transferred to the fabric, stored locally, and made immediately available for processing in the computational row. The middle cell is optimized for throughput, combining RF, BU, and TFG, with the SWB managing intra-cell routing to minimize data movement overhead. This top-to-bottom organization establishes a streamlined and efficient data flow throughout the fabric.

It is important to note that this arrangement corresponds to a single column of the DRRA fabric. For more demanding applications that require higher throughput, the inherent scalability of the DRRA can be exploited by replicating this column multiple times, thereby enabling concurrent execution of several independent FFT computations. This property is particularly relevant for the FMCW radar application considered in this work, where large volumes of data are periodically acquired from the ADC of each antenna, delivering multiple chirps that must be processed within the short time interval preceding the next sampling period. The parallelism offered by the DRRA fabric provides an effective means of meeting these stringent real-time processing requirements.

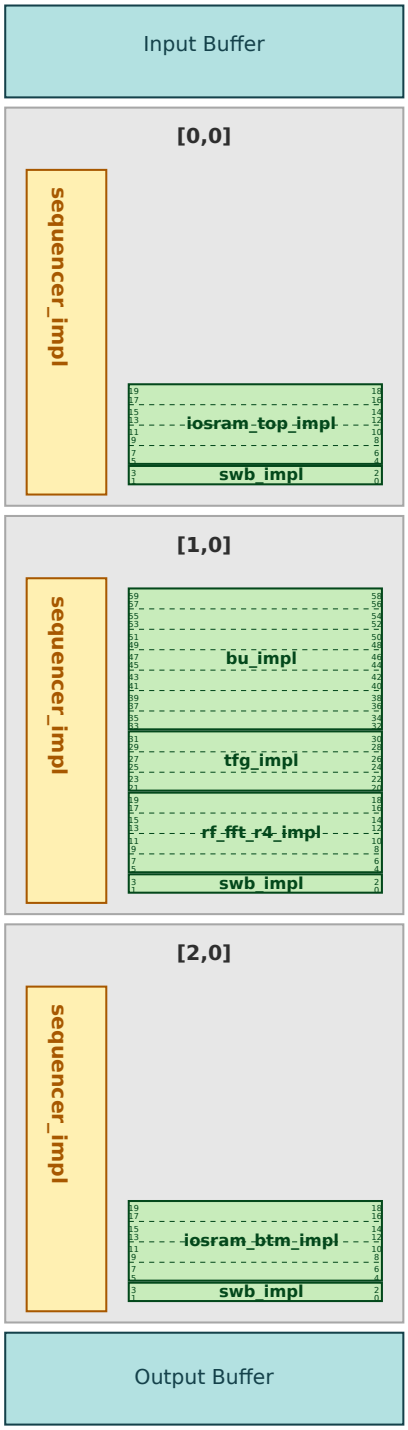


Figure 4.8: Layout of the DRRA fabric for the implementation of the FFT accelerator

4.3 PASM program

The detailed execution flow of the FFT accelerator is programmed through the PASM code reported in Appendix 7.1. The structure and semantics of PASM programs were introduced earlier in Section 3.2, where the notions of operation regions (`rop`) and constraints (`cstr`) were defined. Building on that foundation, each `rop` in the program corresponds here to one of the functional units instantiated in the DRRA column shown in Figure 4.8, while the constraints establish ordering and synchronization rules across them.

The program begins by configuring the routing and data stream units in the first cell using the instructions `route0r`, `input_r`, and `input_w`. Instruction `route0r` (Listing 4.1) establishes a connection between the IO cell input and the IO input port of the SRAM, enabling data exchange. The AGUs subsequently transmit the read address to the buffer and generate the write address for the SRAM.

```
1 rop <route0r> (row=0, col=0, slot=0, port=2){
2   route(slot=0, option=0, sr=0, source=2, target=0b010000000)
3 }
```

Listing 4.1: IO routing

Synchronization between read and write operations is enforced through constraints associated with individual `rop` instructions, as shown in Listing 4.2.

```
1 cstr("input_r == input_w")
```

Listing 4.2: IO read/write constraint

After data are transferred to the SRAM, they are moved to the RF in the computational cell. The connection between Cell_0 and Cell_1 is created using `route1wr`. The transfer process is managed by the AGUs, which generate the read and write addresses for the SRAM and the write address for the RF.

```
1 rop <route1wr> (row=1, col=0, slot=0, port=2){
2   route (slot=0, option=0, sr=1, source=1,
3     target=0b0000000000000010)
4   route (slot=0, option=0, sr=0, source=1, target=0b010000000)
5 }
```

Listing 4.3: Routing between SRAM in Cell_0 and RF in Cell_1

In the central cell, a sequence of `swb` instructions establishes all connections among the RF, TFG, and BU. Specifically, the first four ports of the RF are connected to the first four input ports of the BU, while the remaining three ports are connected to the TFG. The output ports of the BU are then connected to the inputs of the RF. The input and output AGUs of the RF and BU are configured for the specific FFT computation by specifying the required parameters.

```
1 rop <read_d1> (row=1, col=0, slot=1, port=1){
2   fft (slot=1, port=1, n_points=256, radix=0, n_bu=1, mode=1,
     delay=0)
3 }
4 rop <bu> (row=1, col=0, slot=8, port=0){
5   bu (slot=8, option=0, radix=0)
6 }
```

Listing 4.4: FFT configuration and start

The computation is then initiated, with the AGUs managing all read and write operations while the BU processes data until completion. Stream alignment is enforced by associated constraints; for example, twiddle factors must be read one cycle before the corresponding butterfly operation to compensate for the delay introduced by the TFG.

```
1 cstr("read_twid1 == read_d1 - 1")
```

Listing 4.5: Twiddle read contraint

Similarly, the writing process in the RF is delayed by two clock cycles to account for the computation latency in the BU.

```
1 cstr("read_twid1 == read_d1 - 1")
```

Listing 4.6: RF write delay

At the end of the computation, data in the RF are transferred back to the SRAM in the bottom cell and then to the output buffer, similarly to the reading process in the top cell. The completion of the FFT process is controlled by the constraint in Listing 4.7, which imposes a wait for a predetermined number of cycles before reading the results from the RF. These timing constants are automatically calculated and inserted by the compiler based on the timing models of the components.

```
1 cstr("read_res > write_d1 + 512")
```

Listing 4.7: Wait for the end of FFT computation

This programmatic description makes explicit the correspondence between the hardware architecture and the software-controlled execution model. It demonstrates how the fabric naturally maps the FFT process into a sequence of communication and computation regions, with synchronization points ensuring correct execution order. Compared to a purely hardwired accelerator, this approach allows the same fabric to be easily reconfigured to support different FFT sizes, radices, or numbers of butterflies by simply modifying the PASM program.

Chapter 5

Results

The final DRRA-based FFT accelerator design has been evaluated through simulation using the Vesyla framework. To assess its correctness and accuracy, the results obtained from the hardware model were first compared against a golden reference model developed in MATLAB. In addition, a more detailed validation was carried out by comparing the accelerator outputs with those produced by a Python implementation based on standard numerical libraries, as well as with the results obtained from a Texas Instruments DSP library. This multi-level comparison ensures both functional correctness and numerical reliability, while also providing a benchmark against widely used software and hardware solutions. Finally, the design has been synthesized, allowing the extraction of information about area, power consumption, and timing, thus completing the evaluation from both functional and implementation perspectives.

5.1 Simulation

The functional behavior of the proposed FFT accelerator has been evaluated using the Vesyla toolchain, which provides a complete simulation and verification environment for DRRA-based designs, as described in Section 3.4. Starting from the PASM program presented in the previous section, Vesyla translates the abstract program representation into a cycle-accurate simulation model while simultaneously assembling the fabric. The final simulation is executed in Questasim, using the RTL description along with the generated data and instruction memories.

In standard Vesyla test cases, which use predefined library components, the output buffer from the simulation is compared to the outputs generated by `model_0` in C++. A bit-for-bit match indicates that the hardware implementation behaves consistently with the high-level description, validating functional correctness. However, during the development of the FFT components, this method proved challenging

due to fixed-point representations. Differences in rounding and truncation between the two implementations caused small discrepancies in the least significant fractional bits. These minor differences were interpreted as functional mismatches, resulting in false verification failures.

To address this limitation, a golden model was implemented in MATLAB. The input signal was generated and sampled to produce N input samples, which were then applied to both the MATLAB model and the DRRA implementation. The outputs were plotted for comparison, allowing verification of functional correctness while ignoring minor LSB differences. Figure 5.1 shows the comparison between the MATLAB golden model and the DRRA FFT results.

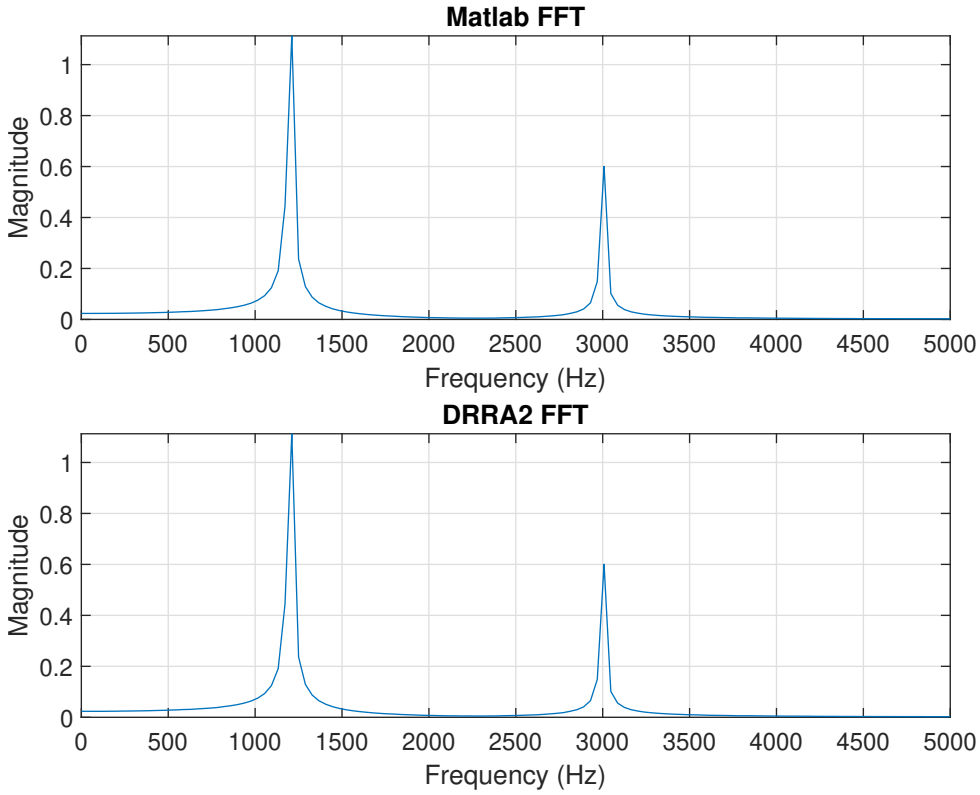


Figure 5.1: Results comparison between the MATLAB golden model and the DRRA FFT results

The sampled signal is defined as:

$$x = 0.01 \sin(2\pi \cdot 1200\, t) + 0.005 \sin(2\pi \cdot 3000\, t) \tag{5.1}$$

representing two sinusoidal components with frequencies $f_1 = 1200\,\text{Hz}$ and $f_2 = 3000\,\text{Hz}$, which correspond to the peaks observed in Figure 5.1.

While this method assesses functional correctness, it does not quantify numerical accuracy. For this purpose, a Python-based evaluation was conducted using two reference models. The first model uses NumPy library functions to produce complex64 FFT outputs with 32-bit floating-point precision. The second model employs the TI DSPLib DSP_fft16x16, which implements a mixed-radix (radix-2 and radix-4) FFT using Q_{15} fixed-point format.

The outputs from these models, along with the DRRA FFT results, were compared to reference data from the Texas Instruments radar Hardware Accelerator (HWA) FFT module [30]. The comparison is summarized in Table 5.1.

Model	**Max. Difference**	**Relative Error** [%]
FFT_NumPy	9.4868	0.0581
FFT_16x16	53.0000	0.3243
DRRA_FFT_2r2	23.0000	0.1408
DRRA_FFT_r4	14.0000	0.0857

Table 5.1: Results comparison between the models and the reference Radar HWA FFT data

The results demonstrate the accuracy of the DRRA-based FFT implementations. Among the two variants, the radix-4 implementation (DRRA_FFT_r4) achieves the closest match, with a maximum difference of 14.0 and a relative error of 0.0857%. The radix-2 implementation (DRRA_FFT_2r2) shows slightly higher deviation, with a maximum difference of 23.0 and a relative error of 0.1408%. This difference arises because the radix-4 FFT requires fewer computational stages, reducing the accumulation of quantization errors introduced in fixed-point arithmetic.

Comparisons with the other models provide further insight. The NumPy-based FFT exhibits the smallest relative error (0.0581%), due to its 32-bit floating-point precision. Conversely, the TI DSPLib FFT_16x16 shows the largest discrepancy (0.3243%) because of its reliance on Q_{15} fixed-point representation. Overall, the DRRA FFT solutions achieve accuracy levels close to the floating-point implementation while maintaining the efficiency of fixed-point arithmetic, with the radix-4 variant offering the best trade-off between precision and computational cost.

The input signal and corresponding FFT outputs are illustrated in Figure 5.2.

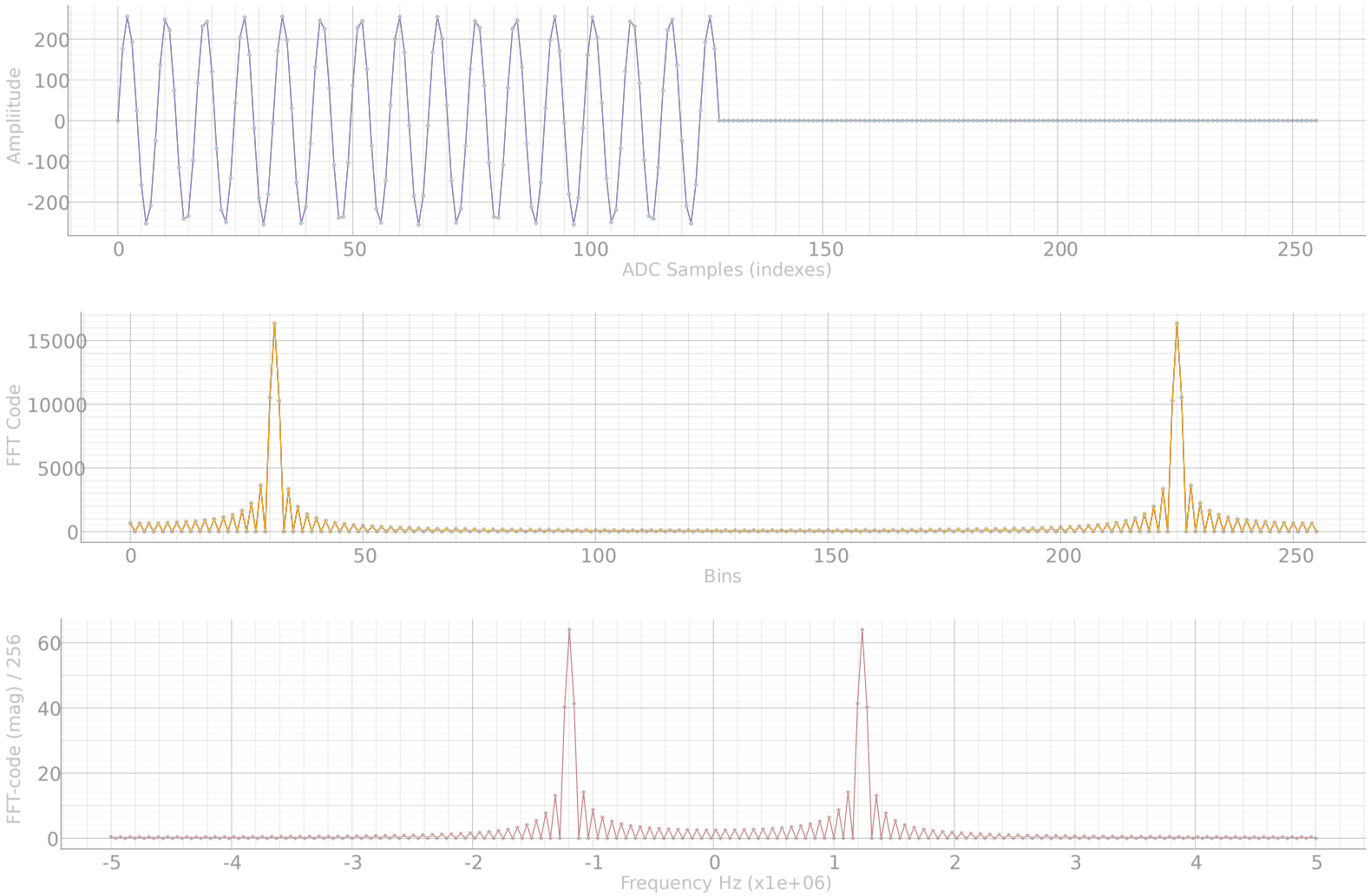


Figure 5.2: Input signal and FFT output, with x-axis corresponding to frequency bins and frequency values

5.2 Synthesis

The FFT accelerator, composed of 3 cells in a single column, was synthesized using Cadence Genus targeting the GlobalFoundries (GF) 22FDX 22 nm Fully-Depleted Silicon-on-Insulator (FD-SOI) process technology, with a nominal supply voltage of 0.8 V. The synthesis flow included standard steps of logical optimization, gate-level mapping, and technology-specific cell selection, allowing for the estimation of area, power, and timing.

The design was synthesized for a target clock period of 1 ns, corresponding to a maximum operating frequency of 1 GHz. Static timing analysis confirmed that this constraint was fully met, with zero slack along the critical path. The critical path was identified within the computational unit of the accelerator: it extends from the `state_reg[0]` register in the finite-state machine of the `bu` resource to the `bu_reg2_reg[15]` register in the same module. This path is associated with the input selection of the radices, which depends on the radix option specified by the instruction. Consequently, the maximum achievable operating frequency of this design is limited to 1 GHz.

Instance	**Module**	**Cell Count**	**Total Area** [μm^2]
fabric	fabric	236763	236721.479
cell_0_0	cell	19022	69932.216
controller	controller	7826	5820.273
resource_0	swb	6991	1927.711
resource_1	iosram_top	4199	62178.242
cell_1_0	cell	200529	97288.622
controller	controller	7007	5574.866
resource_0	swb	11376	4106.486
resource_1	rf_fft_r4	137022	67040.497
resource_5	tfg	4967	2412.268
resource_8	bu	40087	18139.996
cell_2_0	cell	17084	69483.601
controller	controller	8019	5855.683
resource_0	swb	4952	1452.805
resource_1	iosram_btm	4107	62169.123

Table 5.2: Cell count and area breakdown of the DRRA fabric

The total area of the synthesized fabric is reported in Table 5.2. Among

the three cells, the central computational cell (`cell_1_0`) dominates the overall area occupation, accounting for 41.10% of the fabric. The top and bottom cells (`cell_0_0` and `cell_2_0`) contribute 29.54% and 29.36%, respectively, as shown in Figure 5.3. In these two peripheral cells, the area is almost entirely driven by the integrated SRAM blocks, which have a width of 256 bits and a depth of 64 words, representing roughly 85% of their footprint.

Within the central computational cell, the detailed area breakdown is presented in Figure 5.4. The register file resource `rf_fft_r4` is by far the main contributor, occupying 68.33% of the cell area. This is explained by its relatively large depth of 256 words and the complexity of its associated address decoding logic. The (`bu`), where for adders and multipliers are instantiated, accounts for 20.04%. The (`tfg`), implemented with a compact ROM-based architecture, represents only 2.47% of the cell area. The remaining fraction is split between the `controller` and the `swb`. This area distribution reflects the strong impact of memory in FFT implementations, particularly in memory-based architectures where large data sets must be preserved in-place during processing.

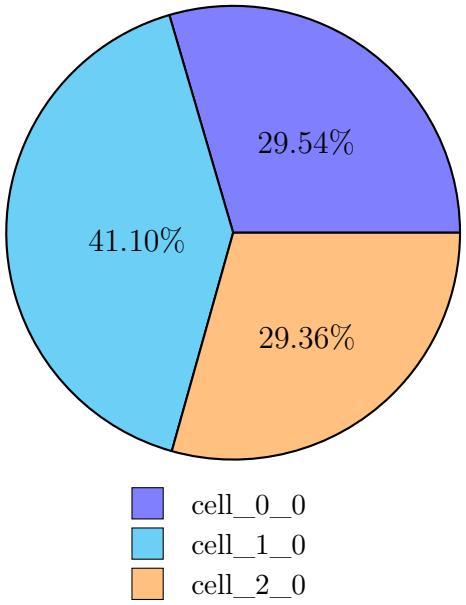


Figure 5.3: Area distribution of top-level cells

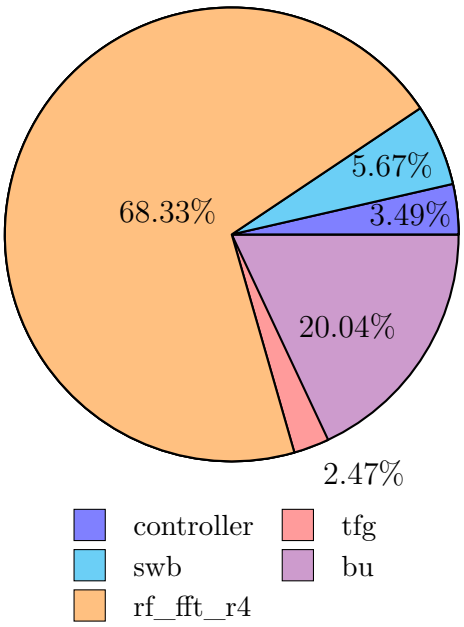


Figure 5.4: Area distribution of cell_1_0

The power characterization of the synthesized design is summarized in Table 5.3. At a clock frequency of 1 GHz, the total power consumption amounts to 98.14 mW. Registers are the primary contributors, responsible for 57.55% of the total power,

followed by logic resources with 33.73%, and memory blocks with 8.72%, as shown in Figure 5.6. When considering the type of power consumption, internal power is dominant at 78.13% of the total, switching power accounts for 21.71%, and leakage power is negligible at only 0.16% (Figure 5.5). The marginal impact of leakage is expected, since the relatively high operating frequency emphasizes dynamic contributions (internal and switching) over static consumption.

Category	**Leakage** [mW]	**Internal** [mW]	**Switching** [mW]	**Total** [mW]
memory	0.100	8.452	0.008	8.562
register	0.014	53.18	3.284	56.48
logic	0.040	15.05	18.02	33.10
Subtotal	0.154	76.68	21.31	98.14

Table 5.3: Power report of the fabric, for clock frequency of $f = 1\,\mathrm{GHz}$

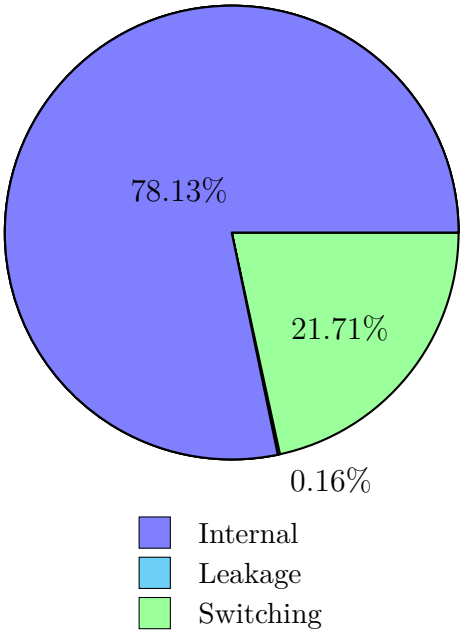


Figure 5.5: Power distribution of the fabric, by category

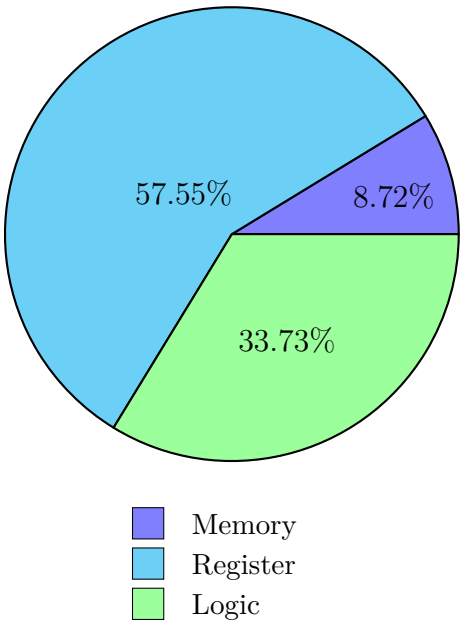


Figure 5.6: Power distribution of the fabric, by resource

A comparison between the proposed FFT accelerator and other designs implementing memory-based variable-size FFT architectures is summarized in Table 5.4.

The comparison considers only designs that focus solely on the FFT computation, excluding works that integrate additional modules. The analysis focuses on 256-point FFT computations and highlights key factors to evaluate efficiency and performance. To provide a fairer comparison between different technology nodes, the area and power have been normalized using the scaling tool described in [31], taking the 22-nm technology used in this work as the reference node.

Moreover, the energy per FFT operation has been computed taking into account both clock frequency and cycle counts as follows:

$$E_{eff} = \frac{P}{f} \cdot N_{cycles} \tag{5.2}$$

Design	**Technology** [nm]	**Voltage** [V]	**Frequency** [MHz]	**Area** [mm^2]	**Latency** [cycles]	**Power** [mW]	**Energy** [nJ/FFT]
This work	22	0.8	1000	0.24	258	98.14	25.3
Chen et al. [32]	45	0.9	1000	0.53	277	58.09	16.09
Wang et al. [33]	16	0.8	940	1.02	284	224.89	67.95
Guo et al. [34]	65	1.0	500	0.20	288	173.78	100.09
DRRA1, Shami [35]	65	-	500	0.22	536	81.61	87.49

Table 5.4: Comparison of FFT accelerator designs

The proposed design, implemented in a 22 nm technology, achieves a silicon footprint of only 0.24 mm^2. This is smaller than [32] (0.53 mm^2) and significantly more compact than [33] (1.02 mm^2), while [34] achieves an even smaller area of 0.20 mm^2.

In terms of performance, all designs complete the computation in a comparable number of cycles, ranging from 258 (this work) to 288 [34]. The proposed design achieves the lowest cycle count using the radix-4 configuration, considering only the cycles for FFT computation and excluding input/output operations. Coupled with a 1 GHz operating frequency, this results in high throughput with minimal overhead.

At 98.14 mW, the proposed design has a higher power consumption than [32] (58.09 mW), but is significantly better than both [33] (224.89 mW) and [34] (173.78 mW). When considering energy efficiency, the proposed accelerator achieves 25.3 nJ/FFT, which is higher than [32] (16.09 nJ/FFT), but demonstrates substantial advantages over [33] (67.95 nJ/FFT) and [34] (100.09 nJ/FFT), with energy reductions of 63% and 75%, respectively.

Compared to the previous FFT architecture implemented in the first-generation DRRA (DRRA-1 [35]), the proposed design demonstrates substantial performance gains. Despite occupying a nearly identical silicon area (0.22 mm^2) and achieving slightly lower power consumption (81.61 mW), the DRRA-1 FFT requires 536 clock

cycles to execute a 256-point FFT, more than twice the latency of the new design. Consequently, the proposed implementation delivers more than 3× improvement in energy efficiency. Overall, the proposed solution achieves a balanced trade-off between silicon area, performance, and energy efficiency, while leveraging the flexibility of the reconfigurable DRRA architecture.

Chapter 6

Conclusions

This thesis has addressed the challenges imposed by mmWave FMCW radar systems, specifically the stringent requirements for efficient, low-latency, and energy-aware execution of the FFT. Within this context, a reconfigurable FFT accelerator has been designed and implemented on the DRRA, a coarse-grained reconfigurable fabric. The proposed solution demonstrates how such an architecture can effectively meet the demands of real-time radar signal processing, where conventional platforms such as DSPs, FPGAs, or ASICs either lack the necessary flexibility or struggle to sustain the required trade-offs among performance, area, and power.

The accelerator integrates dedicated architectural blocks, such as RF, BU, and TFG modules, while leveraging the spatial programmability of the DRRA and the resource-centric configurability enabled by the CIS programming model. This approach allows for efficient mapping of FFT kernels, taking advantage of the inherent parallelism and reduced control overhead.

Extensive functional validation confirmed the correctness of the design by comparison against reference models, demonstrating near floating-point accuracy while maintaining the efficiency of fixed-point arithmetic. Post-synthesis evaluation in 22 nm CMOS technology revealed operation at 1 GHz, with a compact silicon footprint of 0.24 mm^2 and a power consumption of 98.14 mW, corresponding to an energy efficiency of 25.3 nJ per 256-point FFT. When compared to prior DRRA-based FFT solutions, this implementation delivers more than a threefold improvement in energy efficiency, highlighting a favorable balance among throughput, power, and area.

These results confirm the viability of the DRRA as a platform for domain-specific acceleration in radar systems. The work illustrates how reconfigurable fabrics can bridge the gap between the peak efficiency of ASICs and the flexibility of FPGAs, ultimately offering a scalable, reusable, and adaptable framework for emerging high-performance embedded applications.

6.1 Future work

Future developments may extend this research in several directions. Scaling to larger FFT sizes will require novel memory hierarchies and addressing mechanisms to partition data into manageable subsets, while minimizing latency and bandwidth bottlenecks. Hardware-efficient matrix transposition and reordering schemes must be explored to enable 3D-FFT implementations for full range-Doppler-angle processing, thereby unlocking complete radar data cube generation directly on the accelerator. At the same time, exploiting higher degrees of parallelism could further reduce execution time for multi-antenna radar scenarios or to increase the resolution.

In conclusion, this thesis has demonstrated that DRRA-based accelerators can deliver compact, reconfigurable, and energy-efficient FFT processing, establishing a promising foundation for next-generation FMCW radar and related high-performance embedded applications.

Chapter 7

Appendix

7.1 PASM program for DRRA FFT

```
epoch <fft256_2r2> {
  rop <route0r> (row=0, col=0, slot=0, port=2){
    route(slot=0, option=0, sr=0, source=2, target=0b010000000)
  }
  rop <input_r> (row=0, col=0, slot=1, port=0){
    dsu(slot=1, port=0, init_addr=0)
      rep(slot=1, port=0, level=0, iter=31, step=1, delay=0)
  }
  rop <input_w> (row=0, col=0, slot=1, port=2){
    dsu(slot=1, port=2, init_addr=0)
      rep(slot=1, port=2, level=0, iter=31, step=1, delay=0)
  }
  rop <route1wr> (row=1, col=0, slot=0, port=2){
    route (slot=0, option=0, sr=1, source=1, target=0
   b0000000000000010)
    route (slot=0, option=0, sr=0, source=1, target=0b010000000)
  }
  rop <read_val> (row=0, col=0, slot=2, port=3){
    dsu(slot=2, port=3, init_addr=0)
      rep(slot=2, port=3, level=0, iter=31, step=1, delay=0)
  }
  rop <write_val> (row=1, col=0, slot=1, port=2){
    dsu (slot=1, port=2, init_addr=0)
      rep (slot=1, port=2, iter=31, step=1, delay=0)
  }
  rop <swb_0> (row=1, col=0, slot=0, port=0){
    swb (slot=0, option=0, channel=8, source=1, target=8)
    swb (slot=0, option=0, channel=9, source=2, target=9)
    swb (slot=0, option=0, channel=10, source=3, target=10)
    swb (slot=0, option=0, channel=11, source=4, target=11)
```

```
30      swb (slot=0, option=0, channel=12, source=5, target=12)
31      swb (slot=0, option=0, channel=13, source=6, target=13)
32      swb (slot=0, option=0, channel=1, source=8, target=1)
33      swb (slot=0, option=0, channel=2, source=9, target=2)
34      swb (slot=0, option=0, channel=3, source=10, target=3)
35      swb (slot=0, option=0, channel=4, source=11, target=4)
36    }
37    rop <read_d1> (row=1, col=0, slot=1, port=1){
38      fft (slot=1, port=1, n_points=256, radix=0, n_bu=1, mode=1,
       delay=0)
39    }
40    rop <read_d2> (row=1, col=0, slot=2, port=1){
41      fft (slot=2, port=1, n_points=256, radix=0, n_bu=1, mode=1,
       delay=0)
42    }
43    rop <read_d3> (row=1, col=0, slot=3, port=1){
44      fft (slot=3, port=1, n_points=256, radix=0, n_bu=1, mode=1,
       delay=0)
45    }
46    rop <read_d4> (row=1, col=0, slot=4, port=1){
47      fft (slot=4, port=1, n_points=256, radix=0, n_bu=1, mode=1,
       delay=0)
48    }
49    rop <read_twid1> (row=1, col=0, slot=5, port=1){
50      fft (slot=5, port=1, n_points=256, radix=0, n_bu=1, mode=0,
       delay=0)
51    }
52    rop <read_twid2> (row=1, col=0, slot=6, port=1){
53      fft (slot=6, port=1, n_points=256, radix=0, n_bu=1, mode=0,
       delay=0)
54    }
55    rop <write_d1> (row=1, col=0, slot=1, port=0){
56      fft (slot=1, port=0, n_points=256, radix=0, n_bu=1, mode=1,
       delay=0)
57    }
58    rop <write_d2> (row=1, col=0, slot=2, port=0){
59      fft (slot=2, port=0, n_points=256, radix=0, n_bu=1, mode=1,
       delay=0)
60    }
61    rop <write_d3> (row=1, col=0, slot=3, port=0){
62      fft (slot=3, port=0, n_points=256, radix=0, n_bu=1, mode=1,
       delay=0)
63    }
64    rop <write_d4> (row=1, col=0, slot=4, port=0){
65      fft (slot=4, port=0, n_points=256, radix=0, n_bu=1, mode=1,
       delay=0)
66    }
67    rop <bu> (row=1, col=0, slot=8, port=0){
68      bu (slot=8, option=0, radix=0)
```

```
69    }
70    rop <read_res> (row=1, col=0, slot=1, port=3){
71      dsu (slot=1, port=3, init_addr=0)
72        rep (slot=1, port=3, level=0, iter=31, step=1, delay=0)
73    }
74    rop <route2w> (row=2, col=0, slot=0, port=2){
75      route (slot=0, option=0, sr=1, source=1, target=0
       b0000000000000100)
76    }
77    rop <write_res> (row=2, col=0, slot=2, port=2){
78      dsu (slot=2, port=2, init_addr=0)
79        rep (slot=2, port=2, iter=31, step=1, delay=0)
80    }
81    rop <output_r> (row=2, col=0, slot=1, port=3){
82      dsu (slot=1, port=3, init_addr=0)
83        rep (slot=1, port=3, iter=31, step=1, delay=0)
84    }
85    rop <output_w> (row=2, col=0, slot=1, port=1){
86      dsu (slot=1, port=1, init_addr=0)
87        rep (slot=1, port=1, iter=31, step=1, delay=0)
88    }
89
90    cstr("input_r == input_w")
91    cstr("route0r < input_r")
92    cstr("input_w < read_val")
93    cstr("route1wr < read_val")
94    cstr("read_val.e0[0] == write_val.e0[0]")
95    cstr("write_val.e0[31] < read_d1")
96    cstr("swb < read_d1")
97    cstr("bu < read_d1")
98    cstr("read_d2 == read_d1")
99    cstr("read_d3 == read_d1")
100   cstr("read_d4 == read_d1")
101   cstr("read_twid1 == read_d1 - 1")
102   cstr("read_twid2 == read_twid1")
103   cstr("write_d1 == read_d1 + 2")
104   cstr("write_d2 == read_d2 + 2")
105   cstr("write_d3 == read_d3 + 2")
106   cstr("write_d4 == read_d4 + 2")
107   cstr("bu != route1wr")
108   cstr("bu != swb")
109   cstr("read_res > write_d1 + 512")
110   cstr("write_res == read_res + 1")
111   cstr("output_r > write_res")
112   cstr("output_r == output_w")
113 }
```

Listing 7.1: FFT-256 dual radix-2 PASM program

Bibliography

[1] Jonas Fuchs, Markus Gardill, Maximilian Lübke, Anand Dubey, and Fabian Lurz. «A Machine Learning Perspective on Automotive Radar Direction of Arrival Estimation». In: *IEEE Access* 10 (2022), pp. 6775–6797. DOI: 10.1109/ACCESS.2022.3141587 (cit. on p. 1).

[2] Yuwei Cheng, Jingran Su, Mengxin Jiang, and Yimin Liu. «A Novel Radar Point Cloud Generation Method for Robot Environment Perception». In: *IEEE Transactions on Robotics* 38.6 (2022), pp. 3754–3773. DOI: 10.1109/TRO.2022.3185831 (cit. on pp. 1, 2).

[3] Khushi Gupta, Srinivas M. B., Soumya J, Om Jee Pandey, and Linga Reddy Cenkeramaddi. «Automatic Contact-Less Monitoring of Breathing Rate and Heart Rate Utilizing the Fusion of mmWave Radar and Camera Steering System». In: *IEEE Sensors Journal* 22.22 (2022), pp. 22179–22191. DOI: 10.1109/JSEN.2022.3210256 (cit. on p. 2).

[4] Gen Li, Yun Ge, Yiyu Wang, Qingwu Chen, and Gang Wang. «Detection of Human Breathing in Non-Line-of-Sight Region by Using mmWave FMCW Radar». In: *IEEE Transactions on Instrumentation and Measurement* 71 (2022), pp. 1–11. DOI: 10.1109/TIM.2022.3208266 (cit. on p. 2).

[5] Muhammet Emin Yanik and Sandeep Rao. «Radar-Based Multiple Target Classification in Complex Environments Using 1D-CNN Models». In: *2023 IEEE Radar Conference (RadarConf23)*. 2023, pp. 1–6. DOI: 10.1109/RadarConf2351548.2023.10149609 (cit. on p. 2).

[6] Dian T. Nugraha, Andre Roger, and Romain Ygnace. «Integrated FFT accelerator and inline bin-rejection for automotive FMCW radar signal processing». In: *2015 European Radar Conference (EuRAD)*. 2015, pp. 564–567. DOI: 10.1109/EuRAD.2015.7346363 (cit. on pp. 2, 4).

[7] Jinmoo Heo, Yongchul Jung, Seongjoo Lee, and Yunho Jung. «FPGA Implementation of an Efficient FFT Processor for FMCW Radar Signal Processing». In: *Sensors* 21.19 (2021). ISSN: 1424-8220. DOI: 10.3390/s21196443. URL: https://www.mdpi.com/1424-8220/21/19/6443 (cit. on pp. 3, 4, 24, 26).

[8] Tourangbam Harishore Singh, Po-Tsang Huang, Kung-Shuo Kao, Chih-Shiang Cheng, Kuei-Ann Wen, and Li-Chun Wang. «Energy-Efficient Sparse FFT and Compressed Transpose Memory for mmWave FMCW Radar Sensor System». In: *IEEE Transactions on Instrumentation and Measurement* 73 (2024), pp. 1–11. DOI: 10.1109/TIM.2024.3385823 (cit. on pp. 3, 25, 26).

[9] M. A. Shami. «Dynamically Reconfigurable Resource Array». PhD thesis. Stockholm: KTH Royal Institute of Technology, 2012 (cit. on pp. 5, 27, 28).

[10] Yu Yang Jordi Altayo and Ahmed Hemani. «Tile-based Heterogeneous Reconfigurable Architecture Template and Its Instruction Set». In: *Open-Source Computer Archietcture Research Workshop 2024*. Buenos Aires, Argentina, June 2024 (cit. on pp. 5, 30).

[11] Yu Yang, Jordi Altayó González, Paul Delestrac, and Ahmed Hemani. *CIS: Composable Instruction Set for Data Streaming Applications*. 2025. arXiv: 2407.00207 [cs.AR]. URL: https://arxiv.org/abs/2407.00207 (cit. on pp. 5, 27, 30–32).

[12] E. O. Brigham and R. E. Morrow. «The fast Fourier transform». In: *IEEE Spectrum* 4.12 (1967), pp. 63–70. DOI: 10.1109/MSPEC.1967.5217220 (cit. on p. 8).

[13] James W. Cooley and John W. Tukey. «An algorithm for the machine calculation of complex Fourier series». In: *Mathematics of Computation* 19 (1965), pp. 297–301. URL: https://api.semanticscholar.org/CorpusID:121744946 (cit. on p. 8).

[14] P. Duhamel and Henk Hollmann. «'Split radix' FFT algorithm». In: *Electronics Letters* 20 (Feb. 1984), pp. 14–16. DOI: 10.1049/el:19840012 (cit. on p. 12).

[15] Lakshmi Santhosh and Anoop Thomas. «Implementation of radix 2 and radix 22 FFT algorithms on Spartan6 FPGA». In: *2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*. 2013, pp. 1–4. DOI: 10.1109/ICCCNT.2013.6726840 (cit. on p. 13).

[16] Sandeep Rao. *Introduction to mmwave Sensing: FMCW Radars*. Texas Instruments. URL: https://www.ti.com/content/dam/videos/external-videos/ko-kr/2/3816841626001/5415203482001.mp4/subassets/mmwaveSensing-FMCW-offlineviewing_0.pdf (cit. on pp. 14–16, 18).

[17] Mario Garrido, Fahad Qureshi, Jarmo Takala, and Oscar Gustafsson. «Hardware architectures for the fast Fourier transform». In: Oct. 2018, pp. 613–647. ISBN: 978-3-319-91733-7. DOI: 10.1007/978-3-319-91734-4_17 (cit. on pp. 18–20).

[18] Tomasz Patyk, Fahad Qureshi, and Jarmo Takala. «Hardware-Efficient Twiddle Factor Generator for Mixed Radix-2/3/4/5 FFTs». In: *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*. 2016, pp. 201–206. DOI: 10.1109/SiPS.2016.43 (cit. on p. 21).

[19] Qing Liu, Yanping Yu, and kuang Wang. «A novel method of twiddle factor generation of the FFT processor for OFDM system». In: *2006 IET International Conference on Wireless, Mobile and Multimedia Networks*. 2006, pp. 1–3. DOI: 10.1049/cp:20061402 (cit. on p. 21).

[20] Reeshita Kallapu, Dimitrios Stathis, Srinivas Boppu, and Ahmed Hemani. «DRRA-based Reconfigurable Architecture for Mixed-Radix FFT». In: *2023 36th International Conference on VLSI Design and 2023 22nd International Conference on Embedded Systems (VLSID)*. 2023, pp. 25–30. DOI: 10.1109/VLSID57277.2023.00020 (cit. on p. 21).

[21] Jen-Chuan Chi and Sau-Gee Chen. «An efficient FFT twiddle factor generator». In: *2004 12th European Signal Processing Conference*. 2004, pp. 1533–1536 (cit. on p. 22).

[22] Mario Garrido, Jesús Grajal, and Oscar Gustafsson. «Optimum Circuits for Bit-Dimension Permutations». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.5 (2019), pp. 1148–1160. DOI: 10.1109/TVLSI.2019.2892322 (cit. on pp. 22, 23).

[23] Mario Garrido, Jesús Grajal, and Oscar Gustafsson. «Optimum Circuits for Bit Reversal». In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 58.10 (2011), pp. 657–661. DOI: 10.1109/TCSII.2011.2164141 (cit. on p. 23).

[24] Mohan Guo, Dixian Zhao, Qisong Wu, Jiarui Wu, Diwei Li, and Peng Zhang. «An Integrated Real-Time FMCW Radar Baseband Processor in 40-nm CMOS». In: *IEEE Access* 11 (2023), pp. 36041–36051. DOI: 10.1109/ACCESS.2023.3265814 (cit. on pp. 24, 26).

[25] Zeynep Kaya and Mario Garrido. «Low-Latency 64-Parallel 4096-Point Memory-Based FFT for 6G». In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 70.10 (2023), pp. 4004–4014. DOI: 10.1109/TCSI.2023.3298227 (cit. on pp. 25, 26).

[26] Hector A. Gonzalez, Marco Stolba, Bernhard Vogginger, Tim Rosmeisl, Chen Liu, and Christian Mayr. «A Low-footprint FFT Accelerator for a RISC-V-based Multi-core DSP in FMCW Radars». In: *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2024, pp. 1–5. DOI: 10.1109/ISCAS58744.2024.10558386 (cit. on pp. 25, 26).

[27] Lingbo Ai, Zhentao Li, Yang Yu, and Menglin Zeng. «Design of High-Performance Millimeter-Wave Radar Digital Processing Accelerator». In: *2024 International Conference on Microwave and Millimeter Wave Technology (ICMMT)*. Vol. 1. 2024, pp. 1–3. DOI: 10.1109/ICMMT61774.2024.10672454 (cit. on p. 26).

[28] Ahmed Hemani. «The SiLago Method: Next Generation VLSI Architectures and Design Methods». In: *Proceedings of the Third ACM International Workshop on Many-Core Embedded Systems*. MES '16. Seoul, Republic of Korea: Association for Computing Machinery, 2016, p. 1. ISBN: 9781450342629. DOI: 10.1145/2934495.2936779. URL: https://doi.org/10.1145/2934495.2936779 (cit. on p. 28).

[29] SiLago Team. URL: https://silago.eecs.kth.se/docs/ (cit. on pp. 28–30, 38).

[30] Texas Instruments. *Hardware Accelerator (HWA) 2.0 overview 1*. URL: https://www.ti.com/content/dam/videos/external-videos/en-us/8/3816841626001/6282120424001.mp4/subassets/hwa_2.0_slides_-_final_version.pdf (visited on 09/09/2025) (cit. on p. 59).

[31] Satyabrata Sarangi and Bevan Baas. «DeepScaleTool: A Tool for the Accurate Estimation of Technology Scaling in the Deep-Submicron Era». In: *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2021, pp. 1–5. DOI: 10.1109/ISCAS51556.2021.9401196 (cit. on p. 64).

[32] Xiaowen Chen, Yuanwu Lei, Zhonghai Lu, and Shuming Chen. «A Variable-Size FFT Hardware Accelerator Based on Matrix Transposition». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26.10 (2018), pp. 1953–1966. DOI: 10.1109/TVLSI.2018.2846688 (cit. on p. 64).

[33] Angie Wang et al. «A 0.37mm2 LTE/Wi-Fi compatible, memory-based, runtime-reconfigurable 2n3m5k FFT accelerator integrated with a RISC-V core in 16nm FinFET». In: *2017 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. 2017, pp. 305–308. DOI: 10.1109/ASSCC.2017.8240277 (cit. on p. 64).

[34] Lei Guo, Yuhua Tang, Yuanwu Lei, Yong Dou, and Jie Zhou. «Transpose-free variable-size FFT accelerator based on on-chip SRAM». In: *IEICE Electronics Express* 11.15 (2014), pp. 20140171–20140171. DOI: 10.1587/elex.11.20140171 (cit. on p. 64).

[35] Muhammad Ali Shami, Muhammad Adeel Tajammul, and Ahmed Hemani. «Configurable FFT Processor Using Dynamically Reconfigurable Resource Arrays». In: *Journal of Signal Processing Systems* 91.5 (May 2019), pp. 459–473. DOI: 10.1007/s11265-017-1326-7 (cit. on p. 64).