Hardware Architecture for Accelerating Frequency-Domain Ultrasound Image Reconstruction

by

Pooriya Navaeilavasani

B.Sc., Shahid Beheshti University, Iran, 2019

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

University of Victoria

Hardware Architecture for Accelerating Frequency-Domain Ultrasound Image Reconstruction

by

Pooriya Navaeilavasani

B.Sc., Shahid Beheshti University, 2019

Supervisory Committee

Dr. Daler N. Rakhmatov, Supervisor

Department of Electrical and Computer Engineering

Dr. David Capson, Departmental Member

Department of Electrical and Computer Engineering

Abstract

Ultrasound is a widely employed biomedical imaging modality enabling non-invasive, low-cost, and real-time diagnostics. In a typical ultrasound system, a multi-channel transducer emits sound waves into the medium and then records returning echo signals that are subsequently converted into an image of the subsurface structure. Coherent plane-wave compounding (CPWC) is one of the latest ultrasound imaging techniques that involves emitting multiple plane-wave pulses at various angles and then combining angle-specific reconstructed image data into a final frame. This approach offers high data acquisition rates (e.g., hundreds or even thousands of raw data frames per second) that are crucial for capturing fast-changing phenomena in the imaged medium.

High data acquisition rates should be matched with fast data processing to increase the frame rate of reconstructed, or beamformed, image frames. One example of highly efficient plane-wave beamforming methods is the Temme-Mueller algorithm that operates in the Fourier domain. This thesis describes a novel pipelined hardware architecture for accelerating the execution of this algorithm. The proposed design has been coded in VHDL and implemented on a modern Xilinx® field-programmable gate array (FPGA), taking advantage of Xilinx® intellectual property (IP) core reuse to reduce development time.

Our architecture is capable of producing over 1,300 beamformed frames per second, where each frame contains 256K complex-valued data points using the 32-bit floating-point representation for both real and imaginary parts. The correctness of our FPGA-based beamformer has been verified by comparing its output to the reference software-based implementation of the Temme-Mueller algorithm. This verification was done on an experimental ultrasound dataset available as part of the public-domain PICMUS evaluation framework. Our evaluation results demonstrate that the proposed design provides a promising alternative to the conventional GPU-based approach to high-frame-rate ultrasound image reconstruction, paving the way for future algorithmic and architectural enhancements.

Table of Contents

List of Tables

List of Figures

Glossary

ASIC	Application-specific integrated circuits
AXI	Advanced eXtensible Interface
BlockRAM™	Embedded block RAM resources in Xilinx® FPGAs
CPU	Central processing unit
CPWC	Coherent plane-wave compounding
DAS	Delay-and-sum
DDS	Direct Digital Synthesizer
DSPS	Double spectrum and phase shift
FFT	Fast Fourier transform
FPGA	Field programmable gate arrays
FPS	Frames per second
GPU	Graphics processing unit
HFR	High frame rate
IP	Intellectual property (core)
ITFFT	Inverse temporal fast Fourier transform
IXFFT	Inverse spatial fast Fourier transform
LUT	Look-up table
MSE	Mean square error
NData	The number of processed real and imaginary data words
NtFFT	Number of temporal FFT points
NxFFT	Number of spatial FFT points
PLUT	Phase Look-up table
PS	Phase shift
PSNR	Peak signal-to-noise ratio
PW	Plane wave
PWI	Plane wave imaging
PICMUS	Plane wave imaging challenge in medical ultrasound
RF	Radio frequency
RLUT	Remap Look-up table
SSIM	Structural similarity index
TFFT	Temporal fast Fourier transform
TM	Temme-Mueller (migration)
UltraRAM™	Dense embedded RAM resources in Xilinx® FPGAs
VHDL	VHSIC Hardware Description Language
XFFT	Spatial fast Fourier transform

Chapter One

Introduction

1.1 Ultrasound Imaging

Modern medical ultrasonography has advanced significantly in the last 30 years, inspired by Paul Langevin's pioneering work in early twentieth-century acoustic imaging [1]. Ultrasound imaging involves the emission of acoustic waves by an ultrasound system, which then captures and converts returning echoes into visual representations of the body's internal structure. Ultrasound waves have frequencies exceeding the range of human hearing, typically surpassing 20 kHz [2] (e.g., a commonly used value is a few MHz). Ultrasound is a widely employed medical imaging modality that provides low-cost and real-time diagnostics, distinguishing it from techniques like CT and MRI. Beyond their well-known application in medical imaging, ultrasound technology finds utility in a diverse array of fields such as industrial nondestructive testing, structural health monitoring, geological exploration, obstacle detection and ranging, and precision measurements [3].

1.1.1 Ultrasound Imaging Systems

The block diagram illustrating the implementation of a conventional multi-focus ultrasound system is depicted in Figure 1.1.

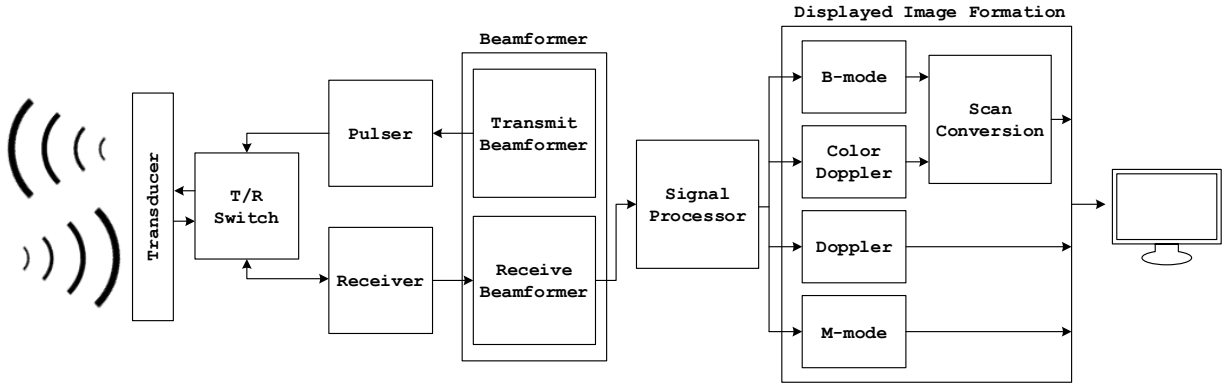


Figure 1.1: Conventional multi-focus ultrasound system (based on [4] and [5]).

The transducer is responsible for generating and receiving ultrasound waves. It contains piezoelectric elements that convert electrical energy into ultrasound waves (transmit mode) and vice versa (receive mode). The pulser is an electronic circuit that generates high-voltage electrical pulses to excite the transducer elements during the transmit mode. The receiver circuit amplifies and conditions the weak returning echoes received by the transducer during the receive mode. The beamformer is a critical part of the ultrasound system that shapes the transmitted ultrasound beam and returning echoes. The transmit beamformer controls the time delay applied to individual transducer elements to focus the beam and steer it in different directions. Its settings determine the

transmitted beam's focus depth, width, and orientation. The receive beamformer, which is the subject of this thesis, processes returning echo signals to reconstruct the reflectivity map of the insonified medium. It is followed by the signal processor that performs tasks like filtering, demodulation, signal correlation, and envelope detection to extract relevant information from the beamformed data. The displayed image formation block converts the output of the signal processor into one or more displayed images, such as B-mode, M-mode, or Doppler depending on the imaging mode [4].

1.1.2 Physics of Ultrasound

The acoustic impedance of tissues is a fundamental property that characterizes how ultrasound waves interact with the imaged medium as they travel through it. This interaction includes the reflection, refraction, scattering, absorption, and attenuation of ultrasound energy (see Figure 1.2) [6]:

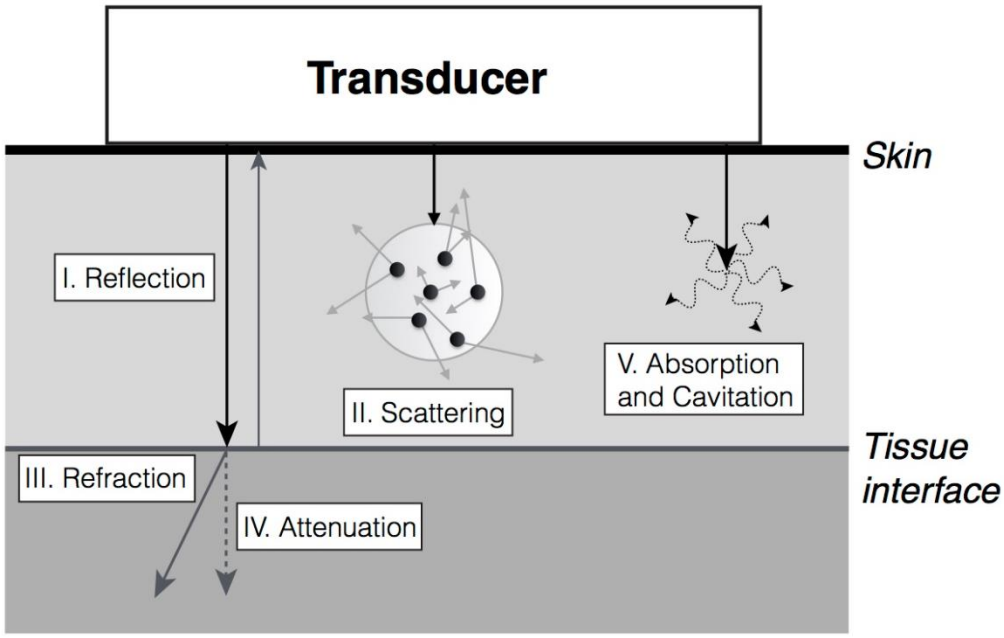


Figure 1.2: Ultrasound and tissue interactions [6].

I. Ultrasound waves reflect when they encounter tissue interfaces with different acoustic impedances, redirecting energy back to the transducer. The intensity of the reflected signal depends on the contrast between tissues, with regions with significant impedance differences producing brighter echoes in ultrasound images. Reflection is the primary mechanism used in ultrasound imaging to visualize tissue interfaces and structures.

II. Scattering occurs when ultrasound waves intersect with microscopic structures or interfaces within tissues, about the size of the ultrasound wavelength or smaller. These microscopic irregularities disperse ultrasound waves in multiple directions, creating a "speckle" pattern in ultrasound images. In other words, a random speckle pattern of bright

and dark spots is a visual representation of the interference of scattered ultrasound waves with small tissue structures.

III. Refraction occurs when an ultrasound wave changes direction as it passes from one tissue into another with a different acoustic velocity. Consequently, the image resolution and contrast can be impacted by this change in the ultrasound beam's trajectory.

IV. Attenuation is the gradual decrease in ultrasound signal strength as it passes through tissue, influenced by factors like absorption, scattering, and beam divergence. Attenuation is frequency-dependent, with low frequencies having better penetration due to less attenuation compared to higher frequencies.

V. Absorption is the conversion of ultrasound energy into thermal heat in biological tissue, which depends on the tissue's composition and properties. Softer tissues like muscles have lower absorption, while denser structures like bones and calcified formations have higher absorption. Excessive absorption can reduce wave propagation capabilities and limit imaging depth.

Ultrasound imaging modes, such as A-mode, B-mode, M-mode, and Doppler (as illustrated in Figure 1.3Figure 1.6), visualize specific information about tissues and structures within the body. Some of the commonly employed ultrasound modes include [7]:

A-mode:

A-mode ultrasound is the simplest and earliest ultrasound mode in medical imaging. It utilizes a single-element transducer and displays a received echo signal on the horizontal axis to indicate depth and on the vertical axis to represent their amplitude. This mode is primarily used for distance measurements and detection of strong reflections.

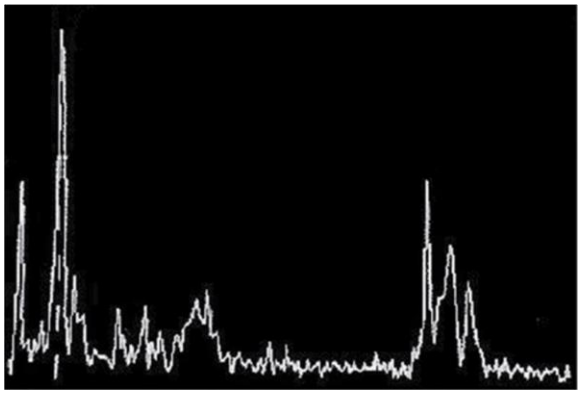


Figure 1.3: A-mode ultrasound image [8].

B-mode:

B-mode ultrasound, the most common mode in medical imaging, typically utilizes a multi-element transducer array to scan the body. It presents two-dimensional grayscale images on the screen, with pixel brightness related to the intensity of received reflections. B-mode is used for visualizing the shape, size, and internal structure of organs, tissues, and abnormalities.

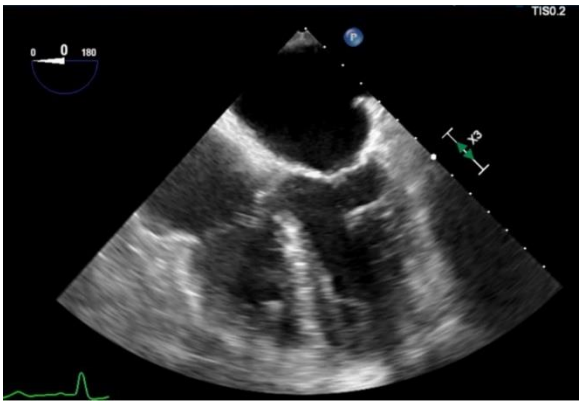


Figure 1.4: B-mode ultrasound image [9].

M-mode:

M-mode ultrasound involves selecting a single scan line and then displaying a real-time graph of motion over time, allowing for dynamic representation of moving structures. This technique is commonly used in cardiology to assess heart motion and valve function.

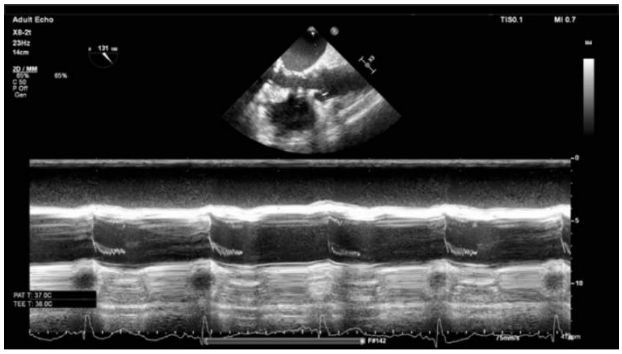
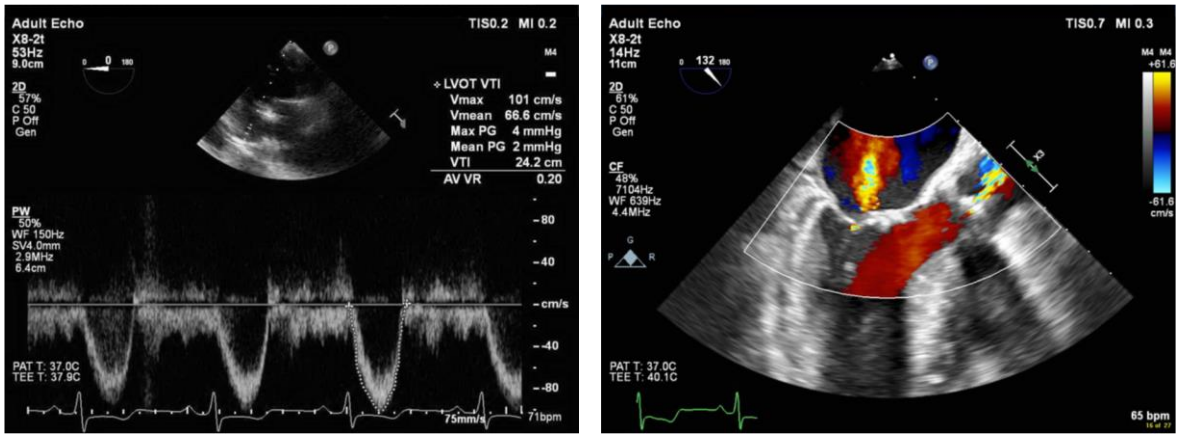


Figure 1.5: B-mode ultrasound image [9].

Doppler:

Doppler ultrasound relies on the Doppler effect to measure blood flow within the body. It measures the velocity and direction of blood flow within vessels. There are two primary types: Color Flow Doppler, which displays blood flow direction using colors, and Pulsed-Wave Doppler, which measures velocity at a specific point within a vessel, providing information about blood flow speed and direction.



(a) (b)

Figure 1.6: Doppler ultrasound image: (a) Pulsed-Wave Doppler (PWD) and (b) Color Flow Doppler (CFD) [9].

1.2 Ultrafast Ultrasound Imaging

1.2.1 Plane-Wave Imaging (PWI)

Ultrafast, or high frame rate (HFR), ultrasound imaging has gained attention for its data acquisition speed increase, reaching up to 20,000 frames per second (fps) when compared to traditional focused imaging [10]. In conventional ultrasound, a focused beam is emitted in a single direction, which yields a single line of a reconstructed image. HFR ultrasound, on the other hand, uses diverging (fan-shaped) or plane (flat-shaped) waves emitted at a fixed angle to insonify a large section simultaneously. This enables rapid imaging of fast-moving structures like the heart [1].

This thesis focuses on ultrafast plane-wave imaging (PWI). Each emitted plane wave (PW) propagates through and interacts with structures, which produces echoes that are received by the ultrasound transducer array. The received echo signals are often referred to as radio-frequency (RF) channel data. For a given array element having a horizontal coordinate x_1, the recorded signal over time t can be expressed as RF (x_1, t).

Reconstructing ultrasound images from an untilted (i.e., zero-angle) PW emission requires the calculation of traveling time, a critical parameter for determining the precise location of structures within the imaged anatomy. Given parameters such as the horizontal position of the receiving element x_1, and the speed of sound in the medium c, the traveling time τ from the transducer to a specific point (x, z) and back, as shown in Figure 1.7, is calculated using the formula:

$$\tau(x_1, x, z) = (z + \sqrt{z^2 + (x - x_1)^2})/c \tag{1.1}$$

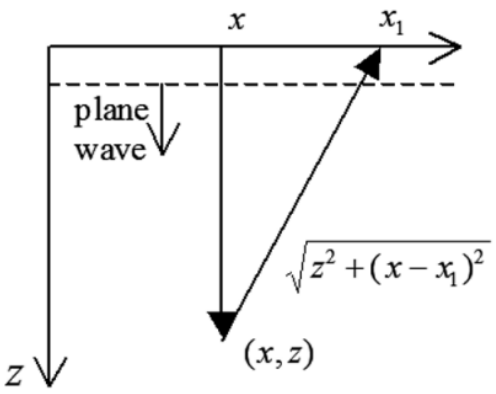


Figure 1.7: Time delay for a zero-angle plane wave [11].

The formation of each pixel value *s*(*x*, *z*) in the image entails the coherent addition of contributions from individual scatters. This process includes delaying $RF(x_1, t)$ signals by $\tau(x_1, x, z)$ followed by their integration along the array direction x_1.

$$s(x,z) = \int RF\big(x_1, \tau(x_1, x, z)\big)dx_1 \tag{1.2}$$

Due to the use of unfocused beams, PWI typically results in lower image resolution compared to traditional focused-beam techniques. This limitation can be mitigated by employing Coherent Plane-Wave Compounding to enhance image quality [12].

1.2.2 Coherent Plane-Wave Compounding (CPWC)

Coherent Plane-Wave Compounding (CPWC) involves transmitting multiple plane waves from different angles and compounding the resulting angle-specific beamformed data to create a high-quality image. This method effectively enhances image resolution, reduces artifacts, and provides a more precise representation of the imaged anatomy [13]. CPWC can offer improvements in image quality, but a user must also consider the trade-off between decreased frame rate and increased resolution, which is influenced by the number and range of the PW angles [14]. This technique is especially valuable in scenarios where high image resolution is important, including applications in obstetrics, cardiology, and musculoskeletal imaging [15].

While the previous section emphasized calculating traveling time for zero-angle plane waves during beamforming, CPWC extends this approach to scenarios with tilted PWs. When a plane wave is transmitted with an inclination angle α, as depicted in Figure 1.8, the total traveling time from the transducer to a specific point (*x*, *z*) within the medium, and back to the receiving element at x_1, can be written as:

$$\tau(\alpha, x_1, x, z) = ((z\cos\alpha + x\ \sin\alpha) + (z + \sqrt{z^2 + (x - x_1)^2}))/c \tag{1.3}$$

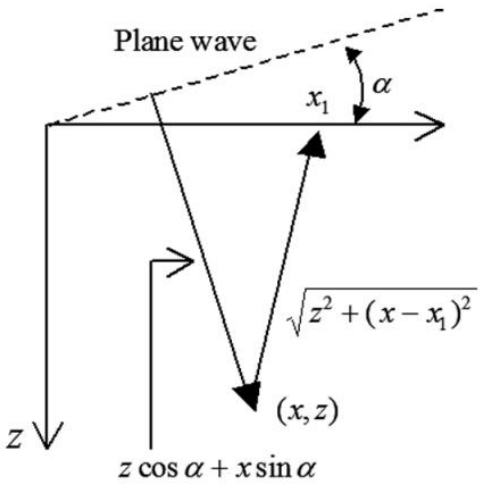


Figure 1.8: Time delays for a plane-wave transmission at an angle α [11].

Subsequently, the beamformed data points are computed using the same equation (1.2) but incorporating the new delay calculated by (1.3) for each PW angle α.

1.3 Receive Beamforming

The transition from channel data to beamformed data is done by the receive beamformer. The most common method in ultrasonic imaging is the Delay-and-Sum (DAS) beamforming [16]. This technique operates in the time domain, and its basic idea has been illustrated in the previous section. Here, we also briefly introduce more advanced (adaptive and frequency-domain) beamformers.

1.3.1 Time-domain beamforming

Conventional DAS beamforming is a well-known time-domain method that involves two basic steps (performed for each beamformed data point): applying delays given by (1.3) to the received RF signals recorded by the individual transducer elements (channels), and then summing the resulting values across the transducer array. Equation (1.2) illustrates the second step, but this summation is typically weighted. The weights employed in the conventional DAS beamformer remain constant throughout the entire beamforming process, thus characterizing it as a non-adaptive technique [17]. In a configuration consisting of M elements in a transducer array, the output of the beamformer can be written as:

$$s_\alpha(x,z) = \sum_{i=1}^{M} w_i \, RF(x_i, \tau(\alpha, x_i, x, z)) \tag{1.4}$$

where w_i is a weight corresponding to the element i. Note that in the CPWC case, final $s(x,z)$ is computed by summing individual $s_\alpha(x,z)$ over all used PW angles α.

Conventional DAS beamforming has widespread adoption in medical ultrasound imaging for several practical reasons, including its theoretical simplicity, straightforward implementation, and

numerical stability, making it a popular choice for various medical imaging applications [18]. However, it employs fixed weights, leading to low image resolution, prominent artifacts, and weak contrast [19].

Adaptive beamforming, on the other hand, dynamically computes weights in real-time based on the statistical characteristics of the channel data. This adaptability allows for improved image quality in terms of resolution and contrast by suppressing off-axis interference and optimizing receiver element weighting in the desired signal directions [18] [19].

1.3.2 Adaptive beamforming

Adaptive beamforming techniques, such as Minimum Variance methods, find widespread application in diverse fields such as radar, sonar, wireless communications, microphone array speech processing, and medical ultrasound imaging [20, 21, 22]. These approaches frequently involve the construction of constrained optimization problems employing Lagrange multipliers to derive weight vectors. For example, these vectors can minimize beamformer output, while ensuring unity gain in the designated signal directions [23].

However, adaptive beamforming is computationally demanding due to the need of creating and inverting matrices associated with the delayed array data vectors. In practical applications, achieving real-time processing remains challenging [22, 24].

1.3.3 Frequency-domain beamforming

The exploration of frequency-based beamforming for ultrasonic imaging dates back to the early 1980s [25, 26]. Fourier-domain receive beamforming emerges as a compelling high-speed alternative to traditional time-domain counterparts. The frequency-based beamforming has also demonstrated the capability to generate high-quality images with fewer input samples, promising enhanced data acquisition efficiency [27, 28].

The influence of synthetic aperture radar/sonar (SAR/SAS) applications has steered the field towards more computationally efficient methods exploiting Fast Fourier Transforms (FFT). These methods adeptly address the focusing challenge through interpolations in the Fourier domain [29, 30, 31].

In this study, we have employed the Fourier-domain beamforming technique known as Temme-Mueller (TM) migration [32], which is an extension of well-known Stolt's algorithm [33]. Compared to alternative Fourier-domain algorithms, TM migration is similar to Lu's method (e.g., see [34]), imposing a minimal burden in terms of the amount of computation required. Its detailed explanation can be found in the next chapter.

1.4 Thesis Contributions and Organization

In this thesis, we present a novel hardware architecture designed to accelerate Fourier-domain image reconstruction using the Temme-Mueller algorithm. Our primary goal is to achieve high frame rates during data reconstruction, exceeding 1,000 frames per second for a single PW angle,

at a power consumption cost significantly less than that of GPUs. This goal is inspired by [41], which demonstrated the feasibility of achieving a data acquisition rate of approximately 500 frames per second for 11 PW angles, which translates into several thousands of frames per second for a single PW angle. Also, given that GPUs typically consume more than 100 W during computation, we aim at a lower power consumption by an order of magnitude. Our motivation stems from the need to increase the efficiency of ultrasound image reconstruction, particularly in the context of Coherent Plane-Wave Compounding (CPWC).

The starting point of our work is a software baseline implementation written in MATLAB®, which serves as a reference for our hardware-based solution. The hardware design is realized through VHDL coding, incorporating the integration of Xilinx® IP cores, to harness the potential of FPGA technology and IP reuse. To assess the effectiveness of our hardware, we conduct a comprehensive evaluation on two levels. First, we utilize Xilinx Vivado® tools to analyze the FPGA implementation characteristics, addressing aspects such as performance and resource utilization. Then, we compare the beamforming results achieved by our proposed hardware architecture to the software output to verify correct operation. This analysis aids in establishing the effectiveness and correctness of our hardware solution in the context of ultrasound image reconstruction.

This thesis is organized as follows. Chapter 2 delves into the background material. It first discusses the fundamentals of Fourier-domain Temme-Mueller migration and explores the resource requirements essential for the proposed hardware design. This chapter also reviews related work in the field, highlighting the contributions of this work. Chapter 3 addresses data organization, external/internal storage, and external/internal data handling. Chapter 4 describes the proposed computational architecture. It presents a detailed overview of the essential computational components. Furthermore, this chapter discusses interconnection structures and elaborates on key performance features, such as computation and communication overlap and multi-frame pipelining. Chapter 5 discusses proposed control mechanisms, covering global and stage-level controls, as well as flexibility features of the architecture. Chapter 6 offers an overview of the FPGA IP cores used for key operations, such as FFT, handling cosine and sine functions, utilizing RAM and DSP blocks for remapping, and defining the I/O interface. Chapter 7 presents the evaluation results. It introduces the test datasets and discusses FPGA implementation characteristics, including resource utilization, performance metrics, and power consumption. This chapter also compares FPGA-based beamforming results with software-based images, employing metrics such as SSIM, PSNR, and MSE. Chapter 8 concludes the thesis and outlines future work.

Chapter Two

Background

In this chapter, we provide a background on key concepts related to Fourier-domain migration, specifically the Temme-Müller algorithm. Additionally, we investigate the resource requirements necessary for implementing this method, including computational aspects, memory management, and communication issues. To contextualize our contributions, we conduct a review of related work in the field, highlighting the unique and novel aspects of our study in comparison to existing literature.

2.1 Fourier-Domain Migration

The primary objective of migration is to transform a raw data frame in the t - x domain (Figure 2.1 (a)) for a particular PW angle into the corresponding beamformed (BF) data frame in the z - x domain (Figure 2.1 (b)) to be compounded over all PW angles in use. In the initial t - x domain, t represents time, and x represents the lateral coordinates of individual transducer elements on the surface. The resulting beamformed data is a subsurface image, where z represents the axial distance, or depth, given by $z = ct/2$, where c is the speed of sound in the imaged medium. The process of Fourier-domain migration in our case is illustrated in Figure 2.2 and described next.

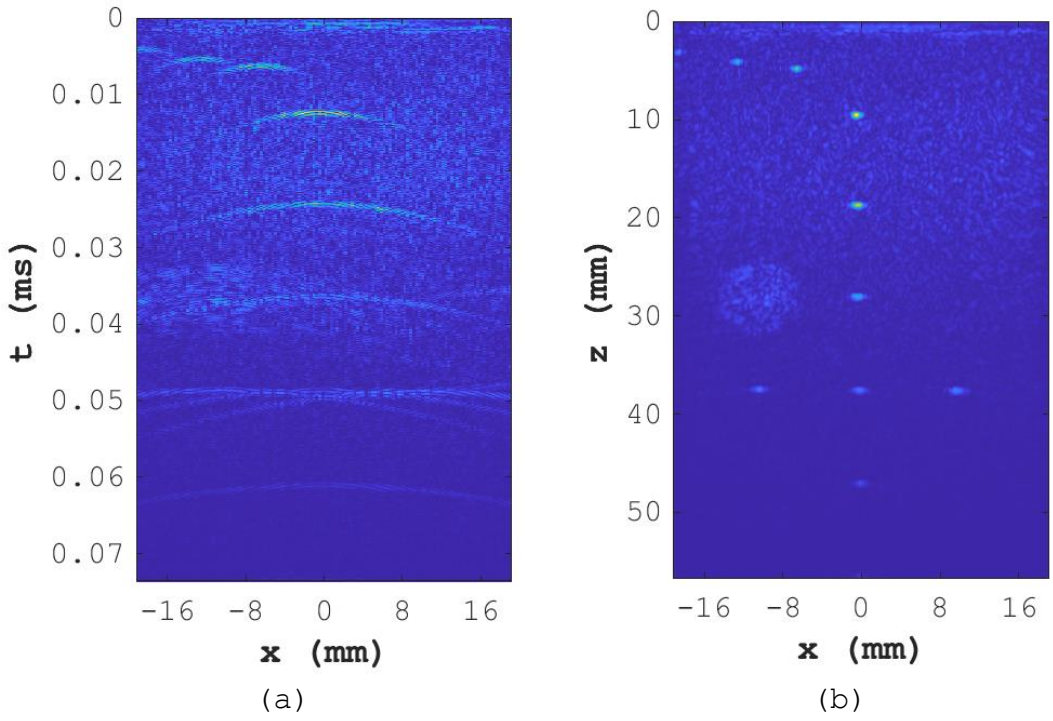


Figure 2.1: Example of: (a) 2D raw data in the *t-x* domain, (b) 2D beamformed data in the *z-x* domain.

Consider $P_\theta(t,z,x)$ as the representation of the acoustic wavefield generated by a PW emission at an angle θ, where t stands for time, while z and x indicate the axial (depth) and lateral (sensor) coordinates, respectively. According to the so-called "exploding reflector model" [33], which assumes that backscattered echoes result from reflectors "exploding" at $t = 0$, the reconstruction of the beamformed dataset $P_\theta(0,z,x)$ at $t = 0$ involves processing the channel dataset $P_\theta(t,0,x)$ recorded by the transducer array on the surface at $z = 0$.

Let $\Psi_\theta(\omega,0,k_x)$ denote the 2D Fourier transform of $P_\theta(t,0,x)$, where ω is the temporal frequency, and k_x is the lateral spatial frequency. The Fourier transform of $P_\theta(0,z,x)$, which is represented as $\Psi_\theta(0,k_z,k_x)$, is obtained by remapping $\Psi_\theta(\omega,0,k_x)$ to $\Psi_\theta(0,k_z,k_x)$ and then taking the 2D inverse Fourier transform [33], where k_z is the axial spatial frequency. Thus, our migration goal is to obtain $\Psi_\theta(\omega,0,k_x)$ from $\Psi_\theta(0,k_z,k_x)$, which is achieved through spectral remapping, i.e., interpolation in the frequency-wavenumber domain.

The Temme-Mueller (TM) algorithm is based on the scalar wave equation model expressed as follows [32]:

$$\frac{d^2\Psi_\theta}{dz^2}+\left(\frac{\omega^2}{c^2}-k_x\right)\Psi_\theta=0 \tag{2.1}$$

where

$$\Psi_\theta(\omega,z,k_x)=\iint P_\theta(t,z,x)e^{-j(\omega t+k_x x)}dxdt \tag{2.2}$$

$$P_\theta(t,z,x)=\frac{1}{4\pi^2}\iint \Psi_\theta(\omega,z,k_x)e^{j(\omega t+k_x x)}dk_x d\omega \tag{2.3}$$

The general solution to Equation (2.1) is

$$\Psi_\theta(\omega,z,k_x)=Ce^{j\left(\frac{\omega^2}{c^2}-k_x\right)^{\frac{1}{2}}z}+De^{-j(\frac{\omega^2}{c^2}-k_x)^{\frac{1}{2}}z} \tag{2.4}$$

The first and second terms, with respective coefficients *C* and *D*, correspond to waves traveling in the upward and downward directions, respectively, along the negative z-axis. We choose $D = 0$, focusing solely on the downward continuation of the upward-traveling reflected wave. The coefficient *C* can be calculated from the boundary condition at $z = 0$:

$$\mathrm{C}=\Psi_\theta(\omega,0,k_x) \tag{2.5}$$

By applying Equation (2.4) to Equation (2.3), we obtain:

$$P_\theta(t,z,x)=\frac{1}{4\pi^2}\iint \Psi_\theta(\omega,0,k_x)e^{j\left(\omega t+k_x x+\left(\frac{\omega^2}{c^2}-k_x\right)^{\frac{1}{2}}z\right)}dk_x d\omega \tag{2.6}$$

where t must be set to $(z\cos\theta + x\ \sin\theta)/c$ [32]. However, calculating the above equation directly is expensive. To enable the use of the FFT, we need to replace ω with k_z. In TM migration, this is achieved through remapping from the (ω, k_x) to (k_z, k_x) grid as follows [32]:

$$\omega(k_z) = \frac{c}{2}\left(\frac{k_z\left(1+\left(\frac{k_z}{k_x}\right)^2\right)}{\cos\theta + \sin\theta\ \left(\frac{k_z}{k_x}\right)}\right) \tag{2.7}$$

$$\alpha_\theta = \frac{c}{2}\left(\frac{\cos\theta\left(1-\left(\frac{k_z}{k_x}\right)^2\right)+2\sin\theta\ \left(\frac{k_z}{k_x}\right)}{\left(\cos\theta + \sin\theta\ \left(\frac{k_z}{k_x}\right)\right)^2}\right) \tag{2.8}$$

By applying Equation (2.7) and (2.8) in Equation (2.6), the final solution of Temme-Mueller migration is given below.

$$P_\theta(0,z,x)\ = \frac{1}{4\pi^2}\iint \alpha_\theta \Psi_\theta(\omega(k_z),0,k_x)e^{j(k_x x + k_z z)}dk_x dk_z \tag{2.9}$$

In the case of CPWC involving multiple PW emissions at different angles θ, coherent compounding can be done in the (ω, k_x) domain:

$$\Psi(0,k_z,k_x)\ = \sum\nolimits_\theta \alpha_\theta \Psi_\theta(\omega(k_z),0,k_x) \tag{2.10}$$

Then, taking the 2D inverse FFT of $\Psi(0,k_z,k_x)$ will produce the CPWC beamformed data frame $P(0,z,x)$. This implies that the inverse Fourier transform is applied only once.

Figure 2.2 summarizes the key steps of the Fourier-domain migration process, with 'F' and 'IF' denoting 1D direct and inverse Fourier transforms, respectively. Notably, for real-valued input data, computational load for the temporal FFT can be halved due to inherent symmetry in the f - x spectrum, where the negative-f half of the spectrum can be discarded without loss of information. In Figure 2.2 and in the sequel, we use f instead of ω to represent the temporal frequency, given that $\omega = 2\pi f$.

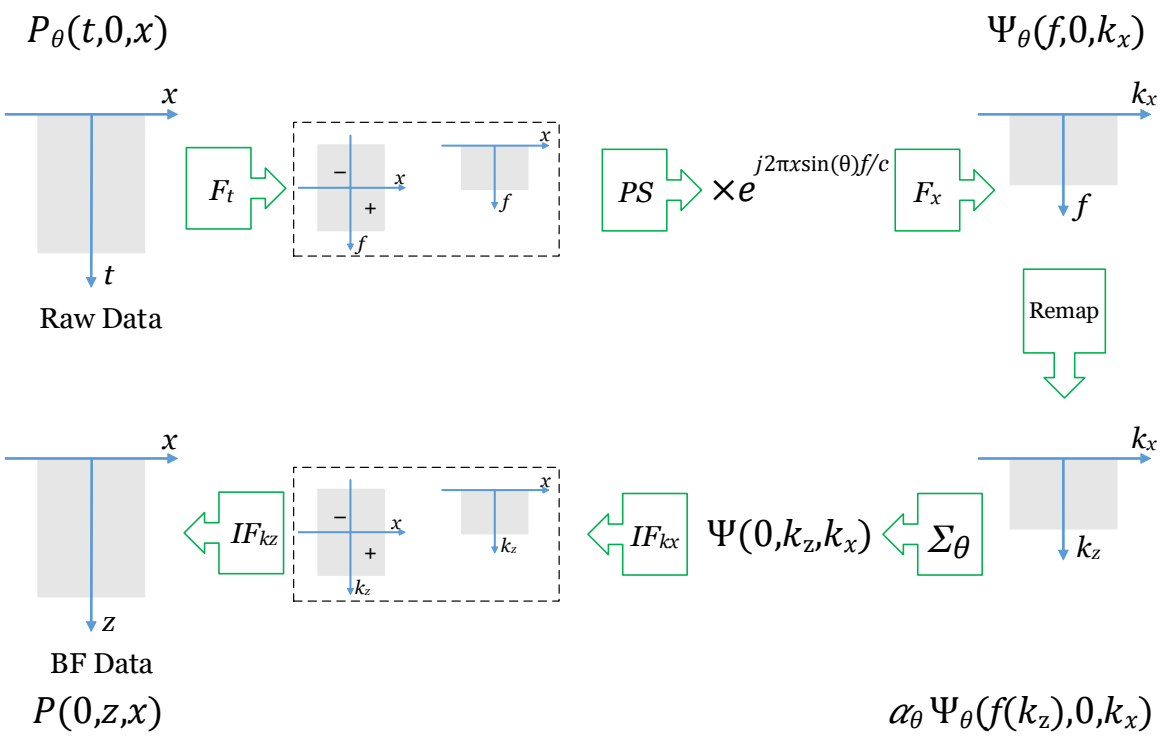


Figure 2.2: Key steps in Temme-Mueller (TM) migration process [35].

2.2 Resource Requirements

The successful implementation of the TM algorithm hinges on several critical resource requirements. In this section, we will explore these requirements to provide insights into the design challenges.

2.2.1 Fast Fourier Transform (FFT)

The TM algorithm heavily relies on the forward and inverse Fourier transforms to achieve the desired wavefield transformations. Therefore, efficient FFT implementations are essential to optimize the algorithm's performance. One of the most widely used FFT algorithms is the Cooley-Tukey algorithm, which employs recursive decomposition of the Discrete Fourier Transform into smaller Discrete Fourier Transforms. This algorithm has the worst-case computational complexity of $O(N \log N)$, where N represents the number of data points [36].

2.2.2 Memory Requirements

The TM algorithm involves processing entire data frames at once, particularly during the Fourier transformation and remapping steps. As a result, substantial memory resources are required to store intermediate results and perform computations.

In PW ultrasound imaging, the frame size is determined by the number of temporal samples and the number of transducer elements. After the 2D Fourier transform, the frame size is constrained by the maximum supported number of temporal FFT points (NtFFT) and maximum supported number of spatial FFT points (NxFFT). In this study, NtFFT is set to 4096, and NxFFT is set to 256, which is sufficient for typical use cases involving a 128-element transducer array and a few thousand temporal samples. Consequently, each frame becomes quite large, especially considering that the algorithm involves multiple stages, necessitating the storage of intermediate data for stage-by-stage processing.

Each data point consists of two parts, the real and imaginary components, represented in the IEEE single-precision format. As a result, the final size of one frame amounts to 8 MB, calculated as (4096 * 256 * 2 * 32 bits). Additional memory is necessary for storing interpolation points, scaling factors, and phase shifts, which will be detailed in the next chapter.

2.2.3 Communication and Data Handling

As previously mentioned, the algorithm consists of multiple stages, each offering the potential for parallel processing and pipelining. Whether employing pipelining or parallel processing setups, maintaining effective communication among processing nodes is vital to ensure efficient data exchange and seamless computation coordination. Additionally, resource-intensive tasks like input data management demand efficient data handling and storage solutions to ensure timely accessibility and processing. Thus, the utilization of high-speed interconnects and communication protocols becomes critical for minimizing communication overhead.

2.3 Related Work

In the domain of ultrasound beamforming methods, extensive research has been conducted over the years, spanning various implementation approaches, including software, hardware, and hybrid solutions.

Several researchers have harnessed the computational power of Graphics Processing Units (GPUs) to accelerate image reconstruction processes, capitalizing on GPUs' parallel processing capabilities to enhance computational speed [37, 38]. For instance, in [39], an ultrasound real-time imaging system employing GPU-based Synthetic Aperture Focusing Technique processing with a PCIe interface delivering a frame rate of 450 fps for images at a resolution of 256x256 pixels. In another study [40], the utilization of two GTX-480 GPUs for beamforming and an additional GTX-470 GPU for recursive compounding showcased the immense capabilities of GPU acceleration. Their beamformer successfully computed compounded 512x255-pixel plane wave and synthetic aperture images at throughputs exceeding 4700 fps and 3000 fps, respectively.

The real-time implementation of GPU-accelerated ultrasound imaging faces significant technical challenges, primarily due to the substantial volume of data involved and the complexities of data transfer. These challenges result in unwanted latency and delays, exacerbated when dealing with larger frame sizes.

To illustrate the scale of the challenge, let's consider a high-end system equipped with 128 channels operating at a 40MHz RF sampling rate (12 bits per sample). In this setup, each transmission event generates raw data totaling a massive 2,212MB. This data volume corresponds to the imaging depth of 7.7cm assuming the sound speed of 1540m/s. In scenarios requiring rapid imaging, the data movement can surge to 18.8GB/s. Even with the integration of an embedded block for PCIe Gen3x16, offering a capacity of approximately 12GB/s, it falls short of meeting the required transfer rate [41]. Furthermore, to ensure accessibility of the raw data for both CPU-driven control flow management and GPU-based algorithmic processing, the data must undergo multiple storage and transportation cycles. This repeated handling introduces delays that constrain real-time performance.

On the other hand, beamforming can be viewed as a dataflow operation where RF data undergoes processing in stages, and each stage is piped onto the next. This characteristic makes modern field-programmable gate array (FPGA) devices a suitable choice for managing data storage and movement across different stages of the beamforming process by leveraging local memories. However, it's worth noting that many FPGA-based implementations have been primarily focused on time-domain techniques, such as DAS beamforming [41].

In the context of frequency-domain beamforming techniques, an FPGA-based hardware has been mentioned in [34]. The proposed approach involved spatial compounding, where different limited-diffraction beams were added either coherently to enhance resolution or incoherently to reduce speckle. Their system was equipped with 128 independent channels, 40-MHz/12-bit converters used for both transmission and reception, and RAM resources of up to 512 MB per channel. However, it's important to note that the FPGA resources were used as part of the data acquisition hardware, while the image reconstruction process itself was done in software on a host PC. In other words, [34] did not address FPGA-based beamformer design aspects.

Additionally, [42] proposed a hardware-oriented algorithm for fixed-point CPWC image reconstruction. However, it should be noted that their approach is based on a method from [28] that is different from Temme-Mueller migration. Their method is more expensive, as it involves compounding in the k_z - x domain and requires an inverse Fourier transform in the spatial domain for every single angle, which adds to the overall computational load. Furthermore, their implementation study focuses on fixed-point data representation and did not report FPGA design-related details.

In this context, we introduce a novel and unique hardware architecture for accelerating the Temme-Mueller algorithm execution, offering the advantage of eliminating redundant inverse Fourier transforms, as this operation is applied only once. Furthermore, our proposed hardware implementation uses standard IEEE single-precision representation, a factor that contributes to enhancing the final image quality. To the best of our knowledge, there have been no prior hardware implementations of this kind reported in the literature.

Chapter Three

Proposed Storage Architecture

In this chapter, we explore data organization within our system and outline the system's data flow. We discuss the intricacies of external and internal storage, shedding light on how data is managed and accessed for both external and internal data handling.

3.1 Data Organization

The entire data storage organization is illustrated in Figure 3.1. To initiate the process, the Ultrasound Data Acquisition block obtains the RF channel data frames from the transducer array and stores them in **`Data Memory`**. Concurrently, the software performs precomputation of essential interpolation points, scaling factors, and phase shifts, storing them in dedicated **`LUT1`** and **`LUT2`** memories.

Once the storage of this information is complete, the `Start` signal is asserted, thus communicating to the FPGA that the data is now accessible and ready for further processing. The FPGA then retrieves this information from the external memories and processes it during image reconstruction. The communication and data read/write requests are facilitated through the AXI Interface/Memory Controller layer. Throughout the processing stages, the hardware utilizes its internal memories for managing intermediate data, ultimately outputting only the final beamformed data, which is stored back in **`Data Memory`**.

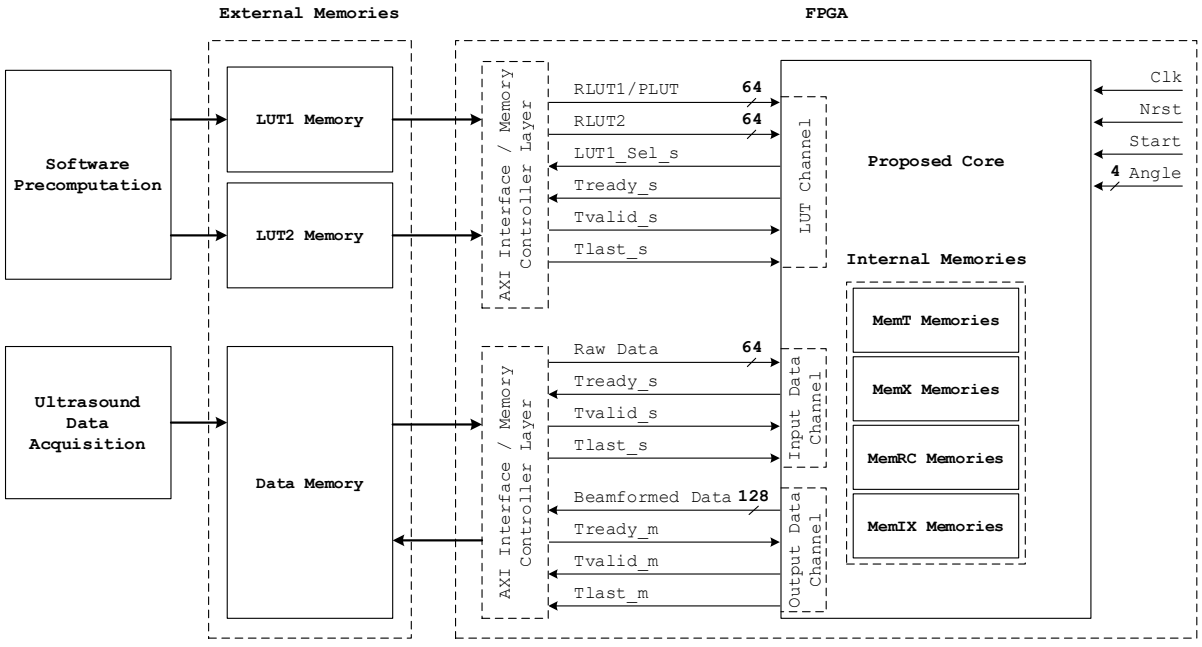


Figure 3.1: Data storage organization.

3.2 External Data Handling

As mentioned earlier, the system software executes several preprocessing steps to calculate frequency interpolation points, scaling factors and phase shifts. The phase shift data is referred to as the Phase Lookup Table (PLUT), and the combination of interpolation points and scaling factors is referred to as the Remap Lookup Table (RLUT). After the preprocessing stage is completed, the software stores PLUT and the first half of RLUT in **`LUT1 Memory`**, and second half of RLUT in **`LUT2 Memory`**.

All memory units, whether internal or external, have the same word size of 64 bits. In general, each data point is stored as a single word having two components: the upper 32 bits represent the imaginary part, while the lower 32 bits represent the real part. A similar structure is applied to the RLUT entries, with each entry containing interpolation points (found in the higher 32 bits) along with associated scaling factors (found in the lower 32 bits). Meanwhile, each PLUT entry is organized to contain pairs of 32-bit phase shift values, compactly stored within a single word.

The minimum required storage for each of these external memories is outlined in Table 3.1. Note that both the raw RF input and the beamformed output (each having 4096*256 data points) are stored in **`Data Memory`** simultaneously.

Table 3.1: Minimum external storage requirement.

Memory	Size (64-bits words)
`Data Memory`	2* 4096 * 128
`LUT1 Memory`	2048 * 128 (Half RLUT) + 128 (PLUT)
`LUT2 Memory`	2048 * 128 (Half RLUT)

For external communication, the proposed core is equipped with AXI Interface/Memory Controller layer. The interfaces for these memories are constructed using the Xilinx® AXI™ bus IP and Memory Interface Generator IP. The communication between proposed core and the interface layer is a simple handshake protocol. All read and write requests for these AXI interfaces are managed by a Global controller.

In Figure 3.1, within the LUT and Input Data channels, the core initiates a data read request by setting `Tready_s`, signaling the interface layer its readiness to accept data. The interfaces then sequentially read data from the external memories, raising `Tvalid_s` to indicate the validity of incoming data. The `Tlast_s` signal is raised on the last sample of each frame, emphasizing the ending of the data transfer for that frame. Given that **`LUT1 Memory`** contains both PLUT and RLUT, the core employs the `LUT1_Sel_s` signal to specify the portion of memory to be read sequentially (i.e., via address incrementing). Additionally, the core utilizes `Tready_s` to either pause or resume data read.

In the Output Data channel, the core utilizes a similar handshake protocol to communicate with the interface layer for data write operations to **`Data Memory`**. Upon the interface layer setting `Tready_m`, the core starts output data traffic by raising `Tvalid_m` and asserts `Tlast_m` high

at the last sample of each frame. The core has control over the pause and resumption of sequential data write through `Tvalid_m`.

3.3 Internal Data Handling

The Temme-Mueller migration algorithm consists of several sequential stages of data processing. To facilitate the transition between these stages, intermediate data must be temporarily stored for use in subsequent processing steps. As previously mentioned, this intermediate data is stored in the core's local memories and is seamlessly loaded when required. Each processing stage employs a pair of memories, which enables the formation of a pipeline structure. Table 3.2 provides an overview of these internal memories' sizes. The **`MemTs`**, **`MemXs`**, **`MemRCs`**, and **`MemIXs`** serve as intermediate buffers for internal communications among computational components. These memories are organized in pairs, and each pair functions as a large ping-pong buffer. When one paired memory is in read mode, the other is in write mode. Switching between these paired memories is accomplished with simple 2:1 multiplexers and 2:1 demultiplexers. The switching behavior is controlled by a Global controller. The specifics of our pipeline structure will be elaborated in the next chapter.

Table 3.2: Internal memories storages.

Memory	Size (64-bit words)
$MemT_1$ / $MemT_2$	2048*128 each
$MemX_1$ / $MemX_2$	2048*256 each
$MemRC_1$ / $MemRC_2$	2048*256 each
$MemIX_1$ / $MemIX_2$	2048*256 each

3.4 Design Process

The development process began with focusing on the proposed core functionality itself, encapsulating its primary inputs and outputs into a testbench. This phase involved simulating single-angle and multi-angle reconstruction scenarios, examining various raw data sets, and supplying precomputed PLUT and RLUT values.

Extending beyond the proposed core, the additional interface signals were incorporated to ensure compatibility with the AXI specifications. By validating such compatibility, we confirmed that the proposed core could be integrated into larger AXI-compliant systems.

The FPGA implementation process for our core unfolded through a series of simulations, including behavioral, post-synthesis, and post-placement-and-routing ones. This multi-step approach allowed for a thorough examination of the design at different stages to verify its correctness and assess its efficiency. The subsequent step of generating and loading a configuration file onto the physical FPGA and conducting tests on a commercial FPGA board was omitted. In other words,

the scope of this work was limited to creating and evaluating a virtual FPGA prototype. Such prototyping allowed us to gain insights into resource requirements and performance implications, which would be valuable when porting our designed architecture from the FPGA to the application-specific integrated circuit (ASIC) implementation fabric.

Chapter Four

Proposed Computational Architecture

We begin this chapter by presenting the implementation steps required for Temme-Mueller migration. Subsequently, we provide an extensive overview of the essential computational resources, explore the interconnection structures, and elaborate on key performance features that are pivotal for efficient computation and communication within the system.

4.1 Temme-Mueller Migration Steps

The algorithm at the core of our system involves seven basic steps depicted in Figure 4.1.

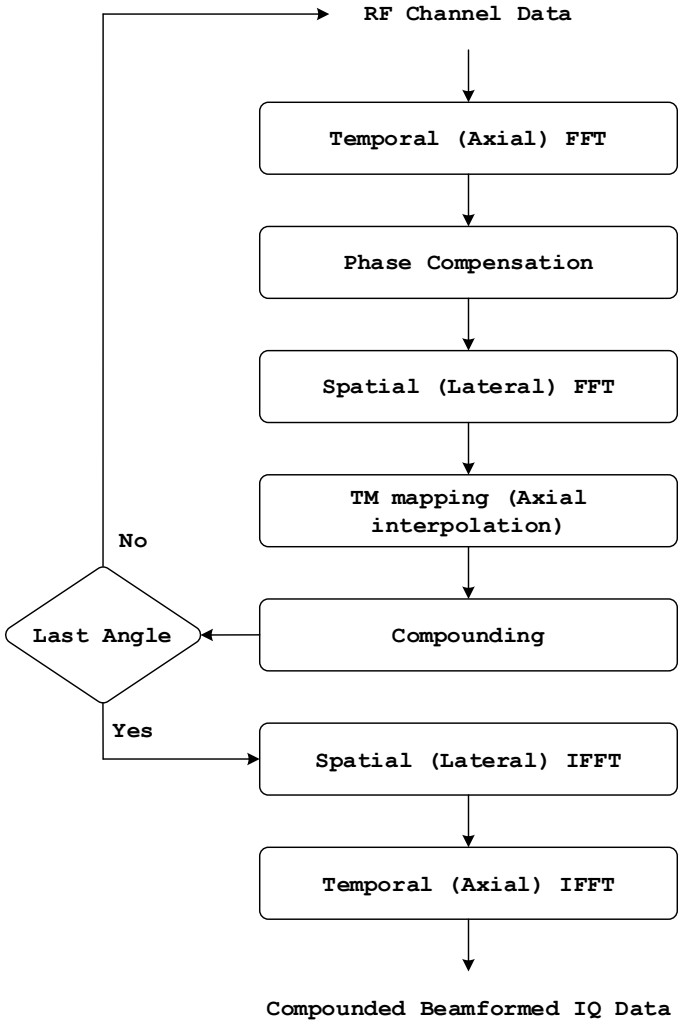


Figure 4.1: Algorithm steps of the core.

Initially, the raw data undergoes temporal Fourier transformation via the 1D FFT. It's important to highlight that, in the case of real-valued input data, we can effectively reduce the number of computations required for the temporal FFT by half. This reduction in computational load stems from the inherent symmetry present in the f - x spectrum of real-valued data. This symmetry implies that the negative frequency half of the f - x spectrum contains the same information as in the positive frequency half. Consequently, it becomes possible to replace an even-length real-input FFT with a half-length complex-input FFT. In this representation, the real and imaginary components of the complex FFT correspond respectively to the even- and odd-indexed elements of the original real data.

Consider an even-length real-valued input sequence, denoted as $x[n]$, with a length of N, where n ranges from 0 to *N-1*. We can derive a complex sequence, $X[k]$, of length *N/2*, where k ranges from 0 to *N/2-1*. The relationship between the real and imaginary parts of $X[k]$ and the original real input is defined as follows:

$$X_{Re}[n] = x[2k] \tag{4.1}$$

$$X_{Im}\left[\frac{N}{2} - k\right] = -x[2k+1] \tag{4.2}$$

Now, if $Y'[k]$ represents the FFT result of $X[k]$, we can reconstruct the FFT result for the original real-valued input $x[n]$, which is denoted as $Y[k]$, as follows [43]:

For $k = 0$:

$$Y_{Re}[0] = Y'_{Re}[0] \tag{4.3}$$

$$Y_{Im}[0] = Y'_{Im}[0] \tag{4.4}$$

For $k > 0$:

$$Y_{Re}[k] = \frac{Y'_{Re}[k] + Y'_{Re}\left[\frac{N}{2} - k\right]}{2} \tag{4.5}$$

$$Y_{Im}[k] = \frac{Y'_{Im}[k] - Y'_{Im}\left[\frac{N}{2} - k\right]}{2} \tag{4.6}$$

Following the reconstruction of the temporal FFT (TFFT) result for the original real-valued input, the next step involves applying the necessary phase shifts, given by $e^{j2\pi k_x \sin(\theta)}$, to the resulting f - x spectrum. The subsequent step involves performing a spatial FFT (XFFT) to transform the data from the f - x domain to the f - k_x domain. In this transformation, the f - k_x half-spectrum is computed using a 1D FFT along the lateral axis.

In the mapping stage, the data undergoes a transformation from the f domain to the k_z domain, followed by scaling. This process is accomplished through linear interpolation followed by scaler

multiplication. The interpolation points for frequency remapping and scaling factors (in RLUT) are pre-calculated by software based on equations (2.7) and (2.8).

In the subsequent stage, the remapped data from various angles are summed together in the k_z - k_x domain, yielding a compounded spectrum. Then, the compounded data is transformed back to the k_z - x domain along the k_x axis using a an inverse spatial FFT (IXFFT) with the same characteristics as XFFT, but in inverse mode. Finally, an inverse temporal Fourier (ITFFT) transform is applied along the k_z axis with the same characteristics as the TFFT, but in inverse mode. It should be noted that before computing the last inverse Fourier transform, the data is reshaped along the k_z axis as described in [44]. After such reshaping, the result of the ITFFT is the analytic IQ data in the z - x domain.

4.2 Main Computational Resources

The top view of the proposed core, based on the algorithm depicted in Figure 4.1, is shown in Figure 4.2.

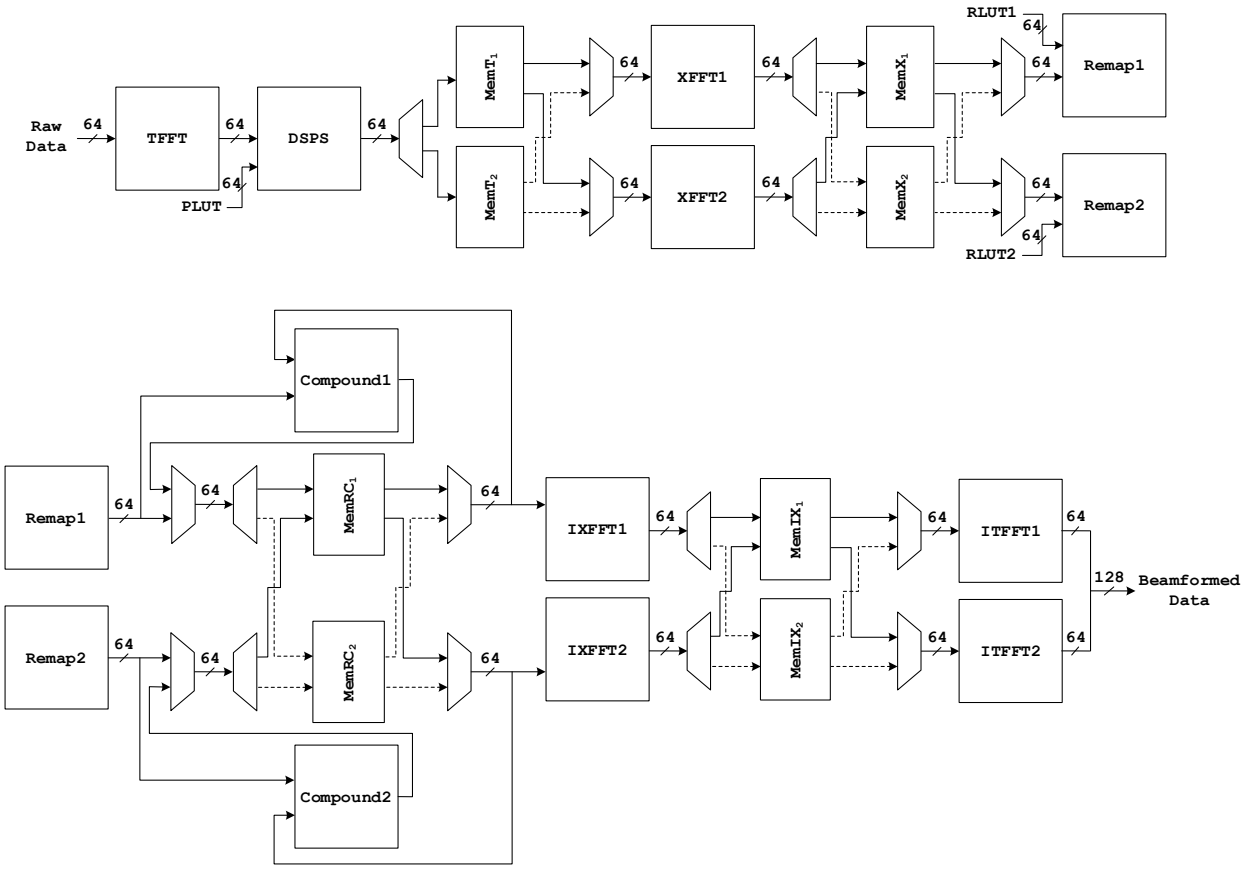


Figure 4.2: Architecture of the proposed core.

Each input data frame has dimensions of 4096 temporal rows and 256 spatial columns. Therefore, the Temporal FFT (**`TFFT`**) is a 4096-point Fourier transform, implemented using the Xilinx Vivado® FFT core, which is further elaborated upon in Chapter 6.

To process one raw data frame, the **TFFT** module iterates 64 times, which corresponds to half the size of the spatial dimension. This optimization is possible because an even-length real-input FFT can be represented as a complex FFT with half the length. Data access for **TFFT** is organized in column order, meaning that data is loaded into **TFFT** column by column. Prior to being written to memory (either $\mathbf{MemT_1}$ or $\mathbf{MemT_2}$), the data passes through the **DSPS** module. As the data ordering required for that module is also in column order, there's no need to reorder the data, and the **TFFT** output can be directly connected to **DSPS**.

The **DSPS** module consists of two parts: the double spectrum formation and phase shift correction. The first part reconstructs the spectrum using Equations (4.3) - (4.6), while the second part applies the appropriate phase shifts (PLUT) to the resulting f - x half-spectrum. This phase adjustment is explained in section 4.2.2.

Following this, the data is written to memory for spatial FFT (**XFFT**), which is a 256-point Fourier transform. The architecture used for this module is similar to that for **TFFT**, but the memory access relies on row ordering.

Since the Temme-Mueller remapping employs temporal frequency interpolation, memory access required by the **Remap** module follows a column order. Therefore, before the data can be utilized by the **Remap** module, the output of **XFFTs** is written into either $\mathbf{MemX_1}$ or $\mathbf{MemX_2}$. As depicted in Figure 4.1, the **Remap** module retrieves the f - k_x data along with precomputed interpolation points and scaling factors (RLUT) and outputs the k_z - k_x mapped data.

The subsequent step is compounding. If there is only one PW angle, this step is skipped; otherwise, the beamformed data frames from different angles of the same compounded frame are summed before inverse Fourier transformation. The **Compound** module simply accumulates remapped data from different angles into a single frame in the k_z - k_x domain. The execution flow for the two different scenarios are detailed in section 4.2.1.

Next, the resulting data frame is written to either $\mathbf{MemRC_1}$ or $\mathbf{MemRC_2}$. It is read row-wise by the inverse lateral FFT (**IXFFT**) modules that transform the data into the k_z - x domain and then write it to either $\mathbf{MemIX_1}$ or $\mathbf{MemIX_2}$. **IXFFTs** utilize the same IP as **XFTTs** but with different setup configurations, as explained in Chapter 6.

In the final step, the data is reshaped based on [44] and then the inverse axial FFT (**ITFFT**) blocks transform the data into the z - x domain. The memory access by **ITFFTs** follows a column ordering. Due to such reshaping, the output of this module consists of the real and imaginary parts of the "analytic" IQ data. **ITFFTs** utilizes the same IP as **TFFT** but with different setup configurations, as detailed in Chapter 6.

To establish a pipeline between various processing stages, different sets of memories have been implemented. Table 4.1 provides information about the size of these memories and their respective read and write patterns. The data_a and data_b symbols refer to the input (when reading) or output (when writing) of a pipeline stage. Note that there are two memories of the same size, which is due to two reasons. First, the read and write patterns are not the same, and most of the time, one

follows a column-wise pattern while the other follows a row-wise pattern and vice versa. Second, although Xilinx® URAM™ (see Chapter 6) supports large memory sizes, it does not support simultaneous read-write operations (only simultaneous read-read or write-write operations). Each pair of memories shares the same switching mechanism, which operates as follows: when the first memory is in write mode to store data from the previous process, the second memory is in read mode to load data for the next process. Once the calculation of one frame is completed, they switch roles, with the first memory changing to read mode and the second one transitioning to write mode, and this pattern continues.

Table 4.1: Internal memories.

Memory	Size (64-bit words)	Write address generation pattern	Read address generation pattern

$MemT_1$
$MemT_2$
	2048*128	**for** i = 0 **to** 127 **do**
		  for j = 0 **to** 2047 **do**
		  address = 127 * j + i
		  MemT(address) = data
		  end for
		end for
			for i = 0 **to** 1024*128-1**do**
			  address_a = i
			  address_b = i + 1024 * 128
			  data_a = MemT(address_a)
			  data_b = MemT(address_b)
			end for

$MemX_1$
$MemX_2$
	2048*256	**for** i = 0 **to** 1023 **do**
		  for j = 0 **to** 255 **do**
		  address_a = 2048 * j + i
		  address_b = 2048 * j + i +1024
		  MemX(address_a) = data_a
		  MemX(address_b) = data_b
		  end for
		end for
			for i = 0 **to** 128*2048-1**do**
			  address_a = i
			  address_b = i + 128*2048
			  data_a = MemX(address_a)
			  data_b = MemX(address_b)
			end for

$MemRC_1$
$MemRC_2$
	2048*256	**for** i = 0 **to** 127 **do**
		  for j = 0 **to** 2047 **do**
		  address_a = 256 * j + i
		  address_b = 256 * j + i + 128
		  MemRC(address_a) = data_a
		  MemRC(address_b) = data_b
		  end for
		end for
			Remap Process:
			for i = 0 **to** 1024*255-1**do**
			  address_a = i
			  address_b = i + 1024 * 255
			  data_a = MemRC(address_a)
			  data_b = MemRC(address_b)
			end for
			Compound Process:
			for i = 0 **to** 127 **do**
			  for j = 0 **to** 2047 **do**
			  address_a = 256 * j + i
			  address_b = 256 * j + i + 128
			  data_a = MemRC(address_a)
			  data_b = MemRC(address_b)
			  end for
			end for

$MemIX_1$
$MemIX_2$
	2048*256	**for** i = 0 **to** 1023 **do**
		  for j = 0 **to** 255 **do**
			for i = 0 **to** 1024*255-1**do**
			  address_a = i

		address_a = 2048 * j + i address_b = 2048 * j + i +1024 MemIX(address_a) = data_a MemIX(address_b) = data_b **end for** **end for**	address_b = i + 1024 * 255 data_a = MemIX(address_a) data_b = MemIX(address_b) **end for**

4.2.1 System Pipeline Timing

Table 4.2 provides details about data processing requirements for the **TFFT**, **XFFT**, **Remap**, **IXFFT**, and **ITFFT** modules per data frame. The *NData* is the number of processed real and imaginary data words. Notably, the total amount of data processed by **TFFT** is 50% less than for the other modules (due to the fact that the RF channel data is real-valued). This observation emphasizes the importance of maintaining a 1:2 ratio between **TFFT** module instantiations and the rest of the modules. For example, if an application requires doubling the frame rate compared to the current design, the implementation should include two **TFFT** modules, alongside four **XFFTs**, four **Remaps**, four **IXFFT** and four **ITFFT** modules.

Another key observation for the **XFFT** and **ITFFT** modules is that the second half of their input data is zero. To conserve memory resources, this portion of data is not stored in the internal memories. Instead, once the read address exceeds 127 (for **XFFT**) or 2047 (for **ITFFT**), it is no longer incremented. Subsequently, the remaining inputs are filled with zeros until the current iteration concludes. Afterward, data reading and address incrementation resume as usual for the next iteration.

Table 4.2: Data processing requirements.

Process	Number of Iterations	NData
TFFT	64	4096
XFFT	2048	256
REMAP	256	2048
COMPOUND	256	2048
IXFFT	2048	256
ITFFT	128	4096

In the single-angle scenario with no compounding, as depicted in the Figure 4.3, the output of **TFFT** for the first frame is written to $\mathbf{MemT_1}$. Subsequently, in the next stage, the **XFFT** module reads data from $\mathbf{MemT_1}$ and writes the output to $\mathbf{MemX_1}$, while **TFFT** writes the data for the second frame to $\mathbf{MemT_2}$. Simultaneously, the **Remap** module reads data from $\mathbf{MemX_1}$ and writes the output to $\mathbf{MemRC_1}$. While all this is happening, the **TFFT** module writes the data for the third frame to $\mathbf{MemT_1}$, and the **XFFT** module writes the data for the second frame to $\mathbf{MemX_2}$. This switching pattern continues throughout the entire process.

When **TFFT** processes the sixth input data frame, the first beamformed data frame is ready, which is indicated by the `Tvalid` signal from the core becoming high. Importantly, as the entire process

operates in a pipelined manner, subsequent reconstructed frames follow one another without any gaps, ensuring a continuous and uninterrupted output stream.

Figure 4.3: High-level timing diagram for single-angle migration.

Figure 4.4 and Figure 4.5 illustrate the timing for a multi-angle scenario with compounding. Specifically, each reconstructed frame comprises a composite of three different angles. The timing process and switching patterns closely resemble those of the single-angle scenario up until the remap process. However, there are notable differences afterwards.

It takes one stage to load one angle-specific input data frame, which means that the three-angle input data (to form one reconstructed frame via migration and compounding) takes three stages to be fully loaded into the core. However, the most significant variation occurs in the compounding process. As illustrated in the figure, the **Remap** module reads angle-1 input for frame 1 (being reconstructed) from $\mathbf{MemX_1}$ and writes the output to $\mathbf{MemRC_1}$. In the subsequent stage, **Remap** reads angle-2 input for frame 1 (being reconstructed) from $\mathbf{MemX_2}$. However, instead of writing the output directly to $\mathbf{MemRC_2}$, it sends it to the **Compound** module. Concurrently, the **Compound** module reads data from $\mathbf{MemRC_1}$, which contains the remapped angle-1 data for frame 1. As a result, the **Compound** module accumulates angle-1 data (stored in memory) and angle-2 data (directly from

Remap), producing a partially compounded output, which is then written to **$MemRC_2$**. In the next stage, **Remap** processes angle-3 input while the **Compound** module compounds data from **$MemRC_2$** with angle-3 remapped data to form the final version of frame 1, written to **$MemRC_1$**.

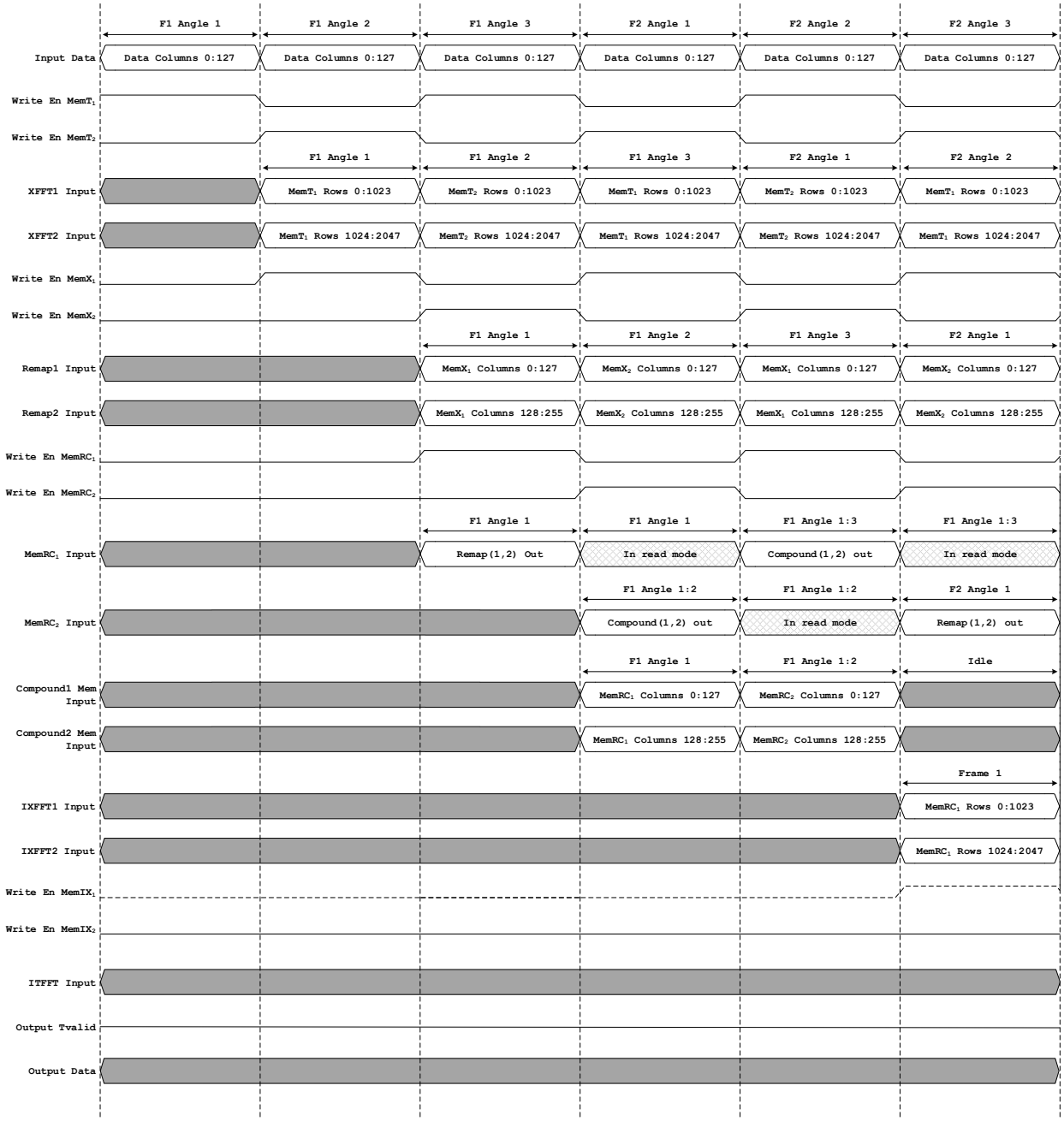


Figure 4.4: High-level timing diagram for three-angle compounded migration (part one).

In the last stage in Figure 4.4 (continued in Figure 4.5), **Remap** is already writing remapped angle-1 data of frame 2 (being reconstructed) to **$MemRC_2$** while the **Compound** module is idle. Meanwhile, **IXFFT** processes compounded frame 1 stored in **$MemRC_1$** and writes its output to **$MemIX_1$**. The remainder of the process is similar to the single-angle scenario, except for the presence of a 2-stage

gap between each reconstructed frame. In general, processing N_a angles will result in a gap of $N_a - 1$ stages.

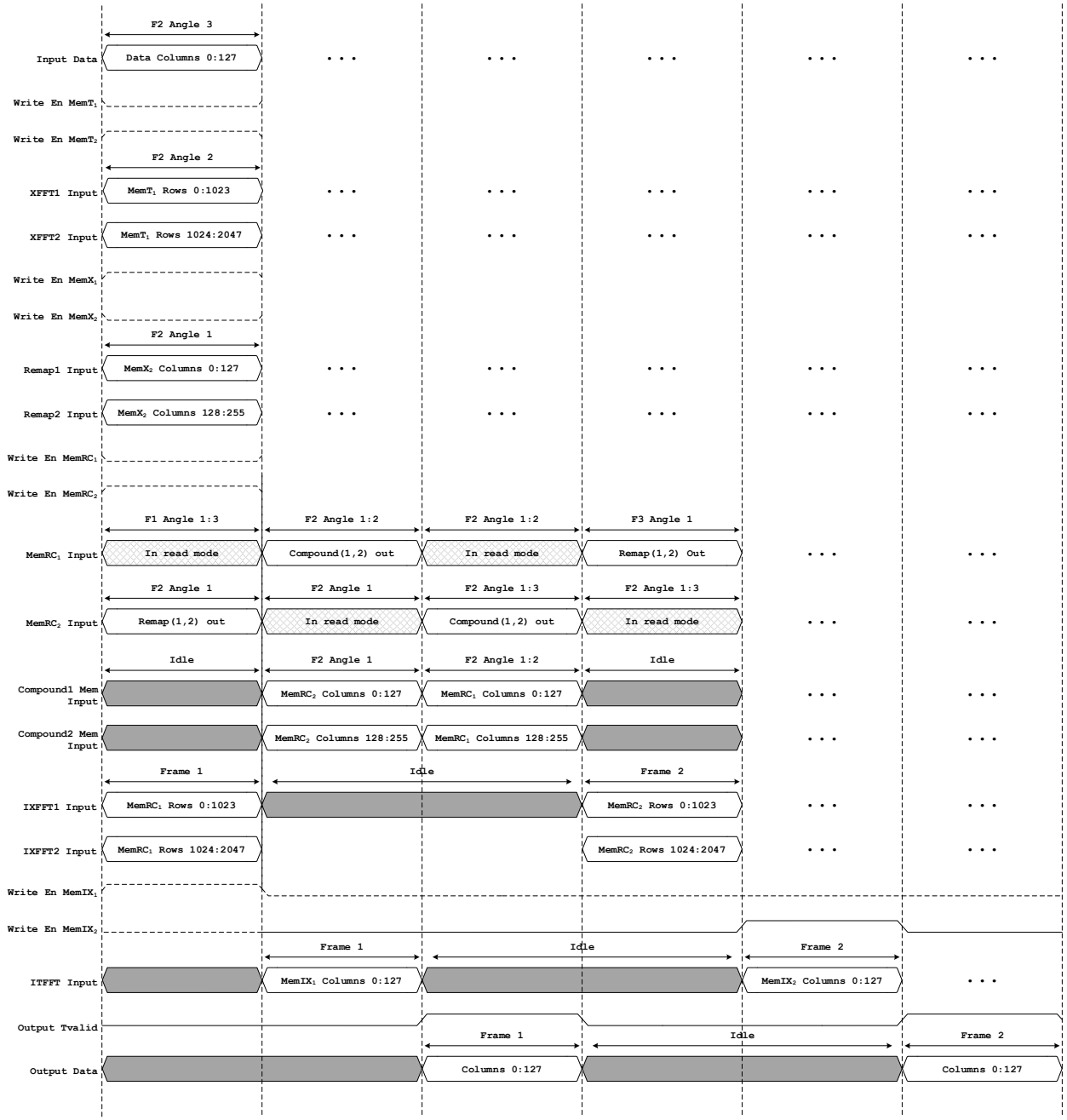


Figure 4.5: High-level timing diagram for three-angle compounded migration (part two).

4.2.2 DSPS module

The **DSPS** module, as previously discussed, is responsible for two tasks: constructing the double spectrum and applying phase shift compensation. Figure 4.6 provides an overview of the module's overall structure.

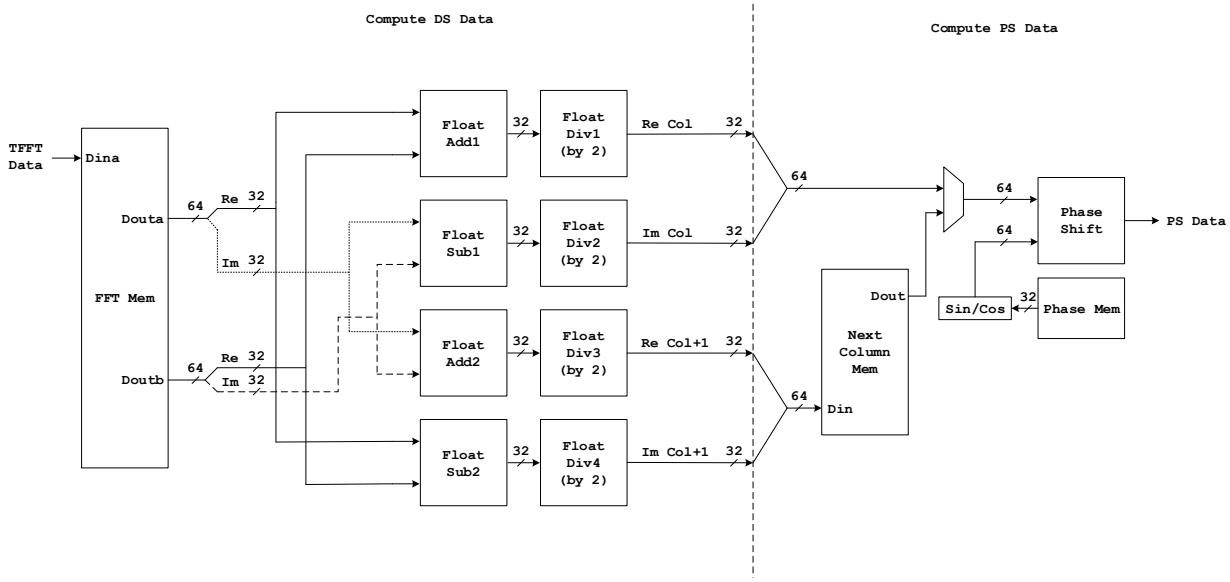


Figure 4.6: Architecture of DSPS module

The construction of the double spectrum relies on Equations (4.3) - (4.6), which necessitates simultaneous access to both the top and bottom halves of each column. To facilitate this requirement, the TFFT data is initially stored in a local buffer. This buffer (labeled **FFT Mem** in Figure 4.6) is a double read/write BlockRAM™ memory provided by the Xilinx® FPGA and has 4096 locations, perfectly suited to accommodate a single column of **TFFT** output. This arrangement enables double reading, providing access to two distinct column indices simultaneously.

Additions and subtractions required by Equations (4.5) and (4.6) are performed using floating-point adders and subtractors, which are based on the Float IP. Further details on this IP are provided in Chapter 6. Additionally, the **Float Div** module is employed to divide the output of the add/subtract float modules by 2. The implementation of this module involves a simple 8-bit subtractor, which decrements the exponent part of the single-precision float input. Since the data for the current column and the next column are generated simultaneously, the next column is stored in another local buffer (labeled **Next Column Mem** in Figure 4.6). It's a single read/write BlockRAM™ memory with 2048 locations for later output.

The second task of applying phase shifts to the *f* - *x* domain data is accomplished in accordance with a simple procedure outlined in Figure 4.7:

Algorithm 1 Phase shifting procedure

```
1   PS = 0
2   for i = 0 to 2047 do
3       PS = PS + Shift Angle
4       if PS > 2π then
5           PS = PS – 2 π
6       end if
7       if PS < -2π then
8           PS = PS + 2 π
9       end if
10      Re_rotate (i)= Re(i) * cos(PS) – Im(i) * sin(PS)
11      Im_rotate (i)= Re(i) * cos(PS) + Im(i) * sin(PS)
12  end for
```

Figure 4.7: Pseudo code for phase shifting.

The **`Sin/Cos`** module receives the initial Shift Angle from **`Phase Mem`** and iterates over all elements of a given column (i.e., over all *f* for a given *x*). When *x* is fixed, the phase shift is linearly dependent on *f*, which translates into simple increments of precomputed Shift Angle. In each iteration, it updates the phase shift value PS, calculates its sine and cosine, and then sends the results to the **`Phase Shift`** module (see Figure 4.9) that implements steps 10 and 11 of the algorithm.

The architecture of the **`Sin/Cos`** core is depicted in Figure 4.8. To calculate sine and cosine values, the **`Direct Digital Synthesizer (DDS) Compiler`** module, provided by Xilinx Vivado®, is employed. This IP utilizes an internal lookup table for and is known for its high performance. It operates with fixed-value inputs and outputs, and our design utilizes the highest precision supported by the core (25-bit input and 26-bit output) to maximize accuracy. Therefore, the `PS Angle` input is first converted to a 27-bit fixed-point representation and stored in a register (labeled **`Reg Angle`** in Figure 4.8) to retain this value for an entire column. The extra 2 bits are for enhanced precision during accumulation.

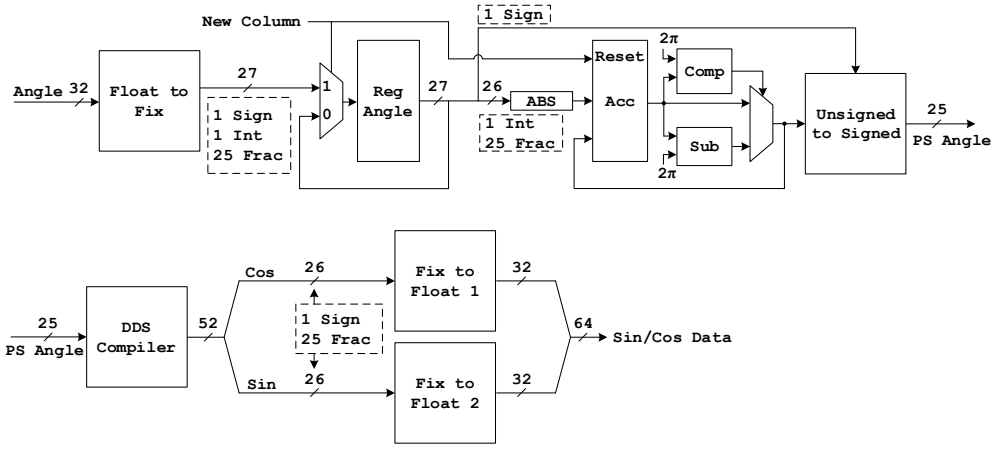


Figure 4.8: Architecture of Sin/Cos module.

The absolute value of the stored angle is computed and sent to the accumulator (**Acc**). This enables reuse of hardware resources for the implementation of lines 3-9 in Figure 4.7. The comparator (**Comp**) and subtractor (**Sub**) components ensure that the accumulation result always remains below 2π. Subsequently, based on the sign bit of the stored value, the accumulated PS is converted back to a signed angle. The **DDS Compiler** outputs 26-bit sin/cos values, which need to be converted back to single-precision floating-point numbers. The Floating-Point IP core provided by Xilinx® is employed to implement the **Float to Fix** and **Fix to Float** components.

The timing diagram of the **Sin/Cos** module is illustrated in Figure 4.11. This module has an initial latency of 16 clock cycles, and its throughput is one sin/cos value pair per cycle. The `PS Angle` output from Phase 0 to Phase 127 within one frame is pipelined. This means that there is no gap between the sin/cos outputs when switching to a new column and the arrival of the next Shift Angle. However, between each frame, there is a gap equal to the module's latency. The seamless output switch between columns is achieved by controlling the `New Column` signal. This signal, which originates from the higher-level controller, routes a new Shift Angle to the register and resets the accumulator.

The **Sin/Cos** and **Double Spectrum** modules work in parallel to prepare input data for the **Phase Shift** module. As depicted in Figure 4.9 the **Phase Shift** module utilizes the `Double Spectrum (DS) data` and the `Sin/Cos` input to calculate the rotated `Re` and `Im` values (lines 10 and 11 in Figure 4.7) that combined into the 64-bit output `PS Data`.

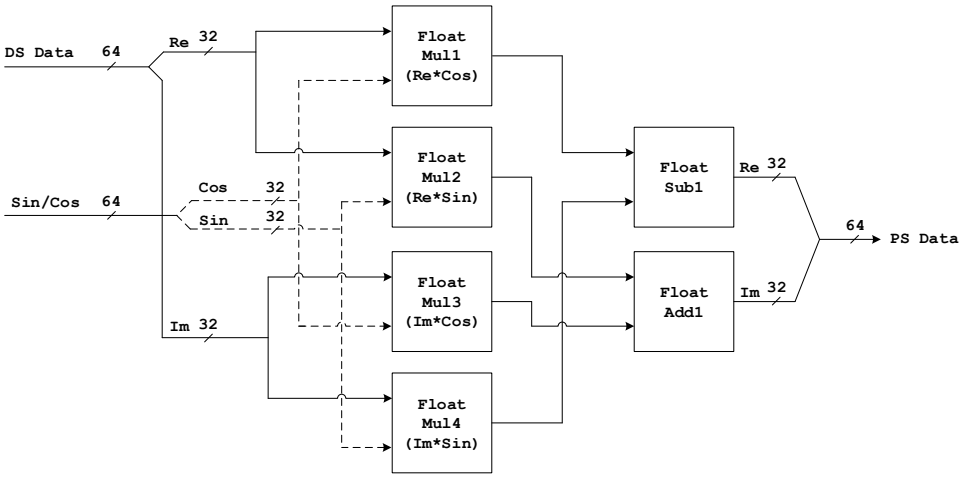
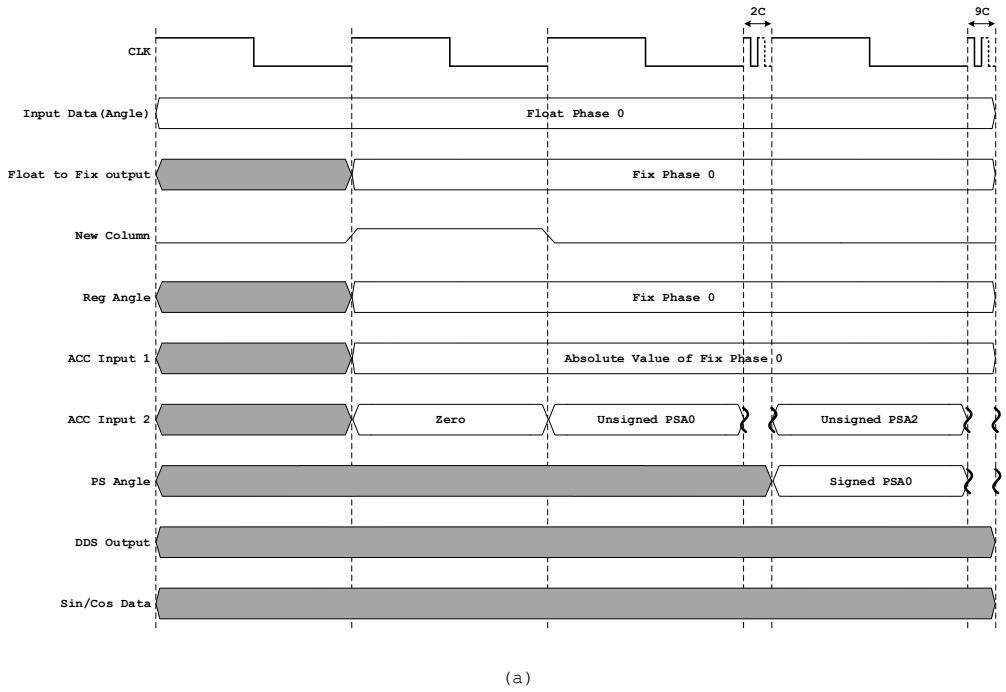


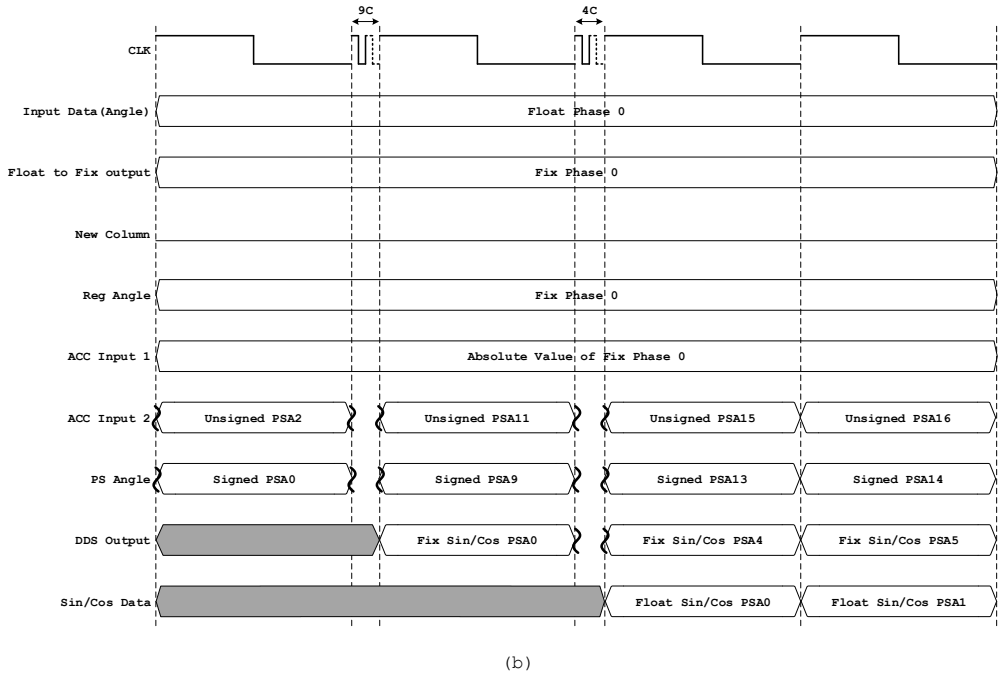
Figure 4.9: Architecture of Phase Shift module.

To illustrate the pipeline between the **TFFT** module and the **DSPS** module, Figure 4.11 show the timing diagram of **DSPS** module. The **TFFT** module performs a 4096-point FFT, resulting in each FFT output column containing 4096 data points. However, the **DS** sub-module of **DSPS** transforms these columns into 2048-sized columns, as each FFT column corresponds to two adjacent columns of real input data. Consequently, the **PS** sub-module of **DSPS** can process two 2048-sized simultaneously while **TFFT** handles one 4096-sized column. In practice, this allows **PS** to directly process the first adjacent column (even columns), while the second adjacent column (odd columns) is temporarily stored in **Next Column Mem**, to be processed by **PS** once the first part is complete.

For each column, **Phase Mem** sends the proper phase angle to the **Sin/Cos** module to calculate the corresponding sin/cos values. The initial latency of both the **DS** sub-module and **Sin/Cos** module is carefully synchronized to ensure that the **Phase Shift** module's inputs are ready at the same time.



(a)



(b)

Figure 4.10: Timing diagram of Sin/Cos module.

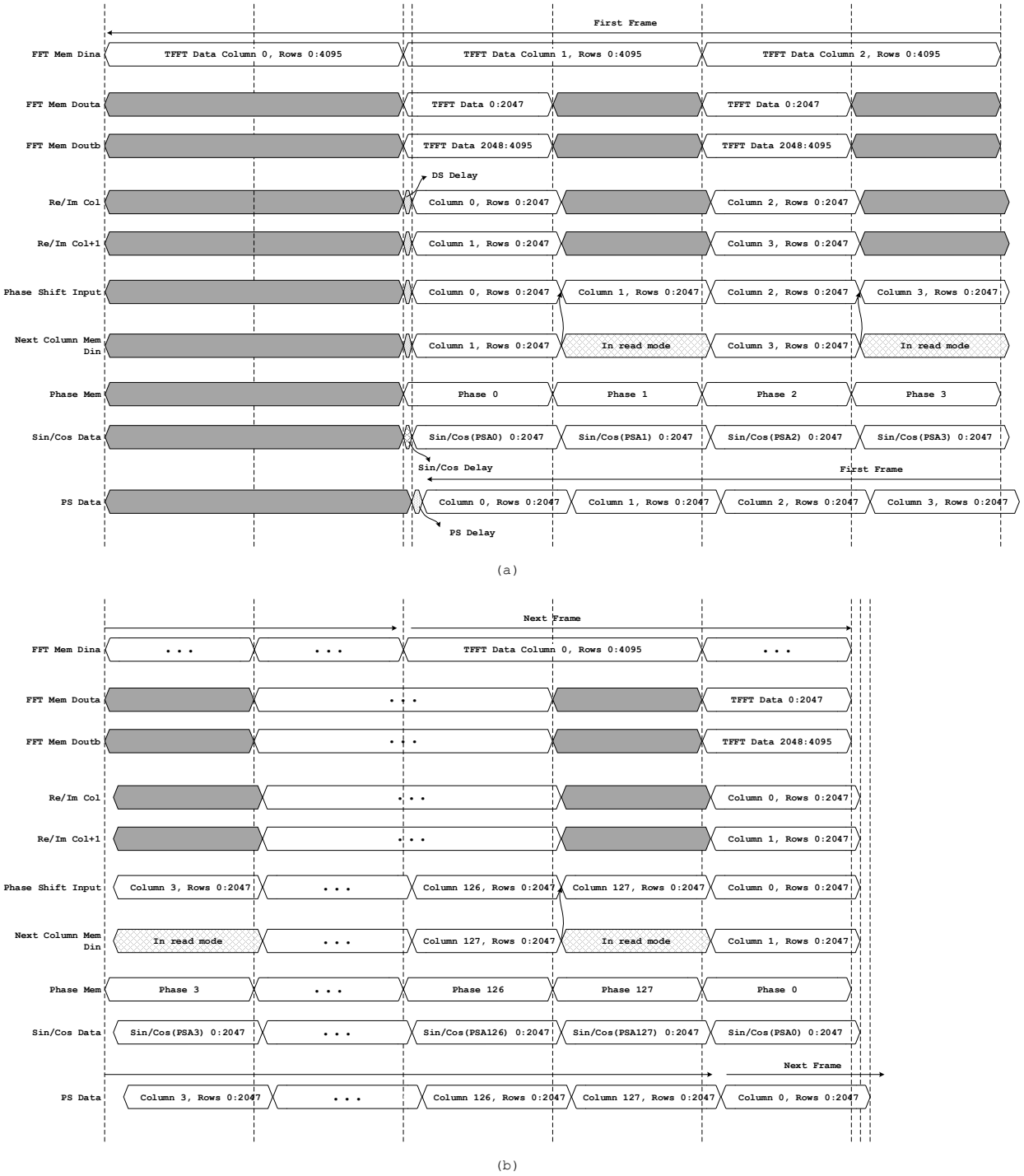


Figure 4.11: Timing diagram of DSPS module.

4.2.3 Remap

The **Remap** module is responsible for performing f-to-k_z frequency remapping, and our architecture has two of them (see Figure 4.2). This module retrieves its input from the internal **MemX** memories and external RLUT that contains interpolation points (iiF) and corresponding scaling factors (SFac). Figure 4.13 provides an overview of this component's architecture. Given that interpolation points are specific to a given k_x column, the first step involves writing an entire column of data to either **Data Mem1** or **Data Mem2**. Simultaneously, the RLUT data for that column, is written to **LUT Mem**. During this process, **Up counter** generates a sequence of write addresses used to load these memories. It's important to note that interpolation points are rational numbers, and the interpolation method employed here is linear interpolation. To perform the interpolation and scaling, the following algorithm is utilized:

Algorithm 2 Remapping procedure for a given column element

1 int = floor(iiF)
2 fracPart = iiF – int
3 $Re_{Remapped}$ = Re(int) + (Re(int +1) - Re(int)) * fracPart
4 Re_{Scaled} = $Re_{Remapped}$ * SFac
5 $Im_{Remapped}$ = Im(intPart) + (Im(intPart+1) - Im(intPart)) * fracPart
6 Im_{Scaled} = $Im_{Remapped}$ * SFac

Figure 4.12: Pseudo code for remapping (one point).

Once **LUT Mem** and one of **Data Mem** are filled, **LUT Mem** is read from address 0, while the addresses for **Data Mem** are given by int and int+1. To accomplish this, the interpolation point (iiF) is initially converted into a fixed number. The module used for this purpose is the Xilinx® Floating Point™ IP. The output conversion is done at the maximum precision supported by this IP, resulting in a 35-bit fixed-point number, comprising 1 bit for the sign, 11 bits for the integer part (**Data Mem** has 2048 locations), and 23 bits for the fraction. The fraction part (frac) is then converted back into the floating-point fraction part (Frac) for subsequent calculations. The int value is incremented by one to obtain int+1, and these numbers are fed to Addra and Addrb to retrieve $ImRe_{int}$ and $ImRe_{int+1}$, respectively. Then, eight Xilinx® Floating Point™ IP instances are utilized to compute lines 3-4 in Figure 4.12, as illustrated in Figure 4.13. The remapped Re and Im values are computed in parallel, and they are subsequently combined into a single Remap Data word.

It's important to note that **Data Mem1** and **Data Mem2** function as a ping-pong buffer. While data is read from **Data Mem1**, the next column's data is written to **Data Mem2**, and vice versa. On the other hand, **LUT Mem** is a single read/write memory that operates in a first-read mode. This means that while the RLUT data of the current column is being read for remapping, the RLUT data for the next column is overwriting the **LUT Mem**. Consequently, the output between columns is pipelined and gapless. All these memories are implemented as BlockRAM™ IP provided by Xilinx®.

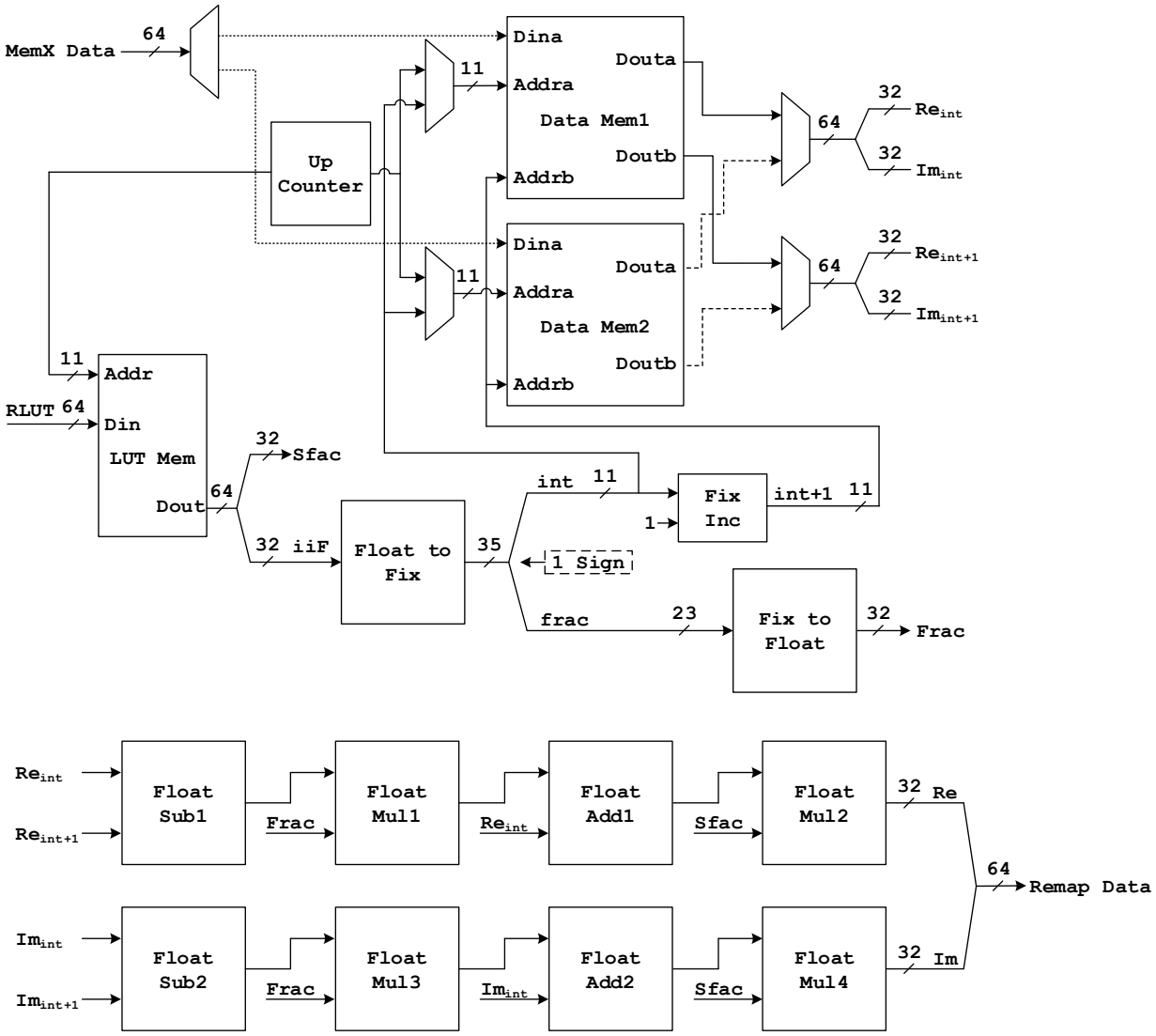


Figure 4.13: Architecture of Remap module.

Figure 4.14 illustrates the timing diagram of the **Remap** module. Given that all the memories in the Remap design are BlockRAMs™, read requests take three clock cycles to complete. One `iiF` read from memory requires 4 cycles for conversion from float to fixed-point, obtaining the `int` and `int+1`. The numbers in square brackets refer to the frequency bin within a column, also indicating where the `Sfac/iiF` values are located in the **LUT Mem**.

The **Remap** module is pipelined and has the initial latency of 41 clock cycles. As shown in Figure 4.14, to synchronize the `frac` and `SFrac` signals with the rest of the calculations, these signals will be delayed. The latencies of the **Float Sub** and **Float Mul** modules are 8 and 6 clock cycles, respectively. The `frac` signal is converted from fixed-point to float (`Frac`), which takes 4 clock cycles. Then, the `Frac` output will be delayed for 7 cycles, matching the 8-clock cycle latency of modules **Float Sub1** and **Float Sub2**. Additionally, the arrow pointing to `Sfac`

signifies a delay of 29 clock cycles, as it will be utilized in the final Float multiplier stage (**`Float Mul2`** and **`Float Mul4`**).

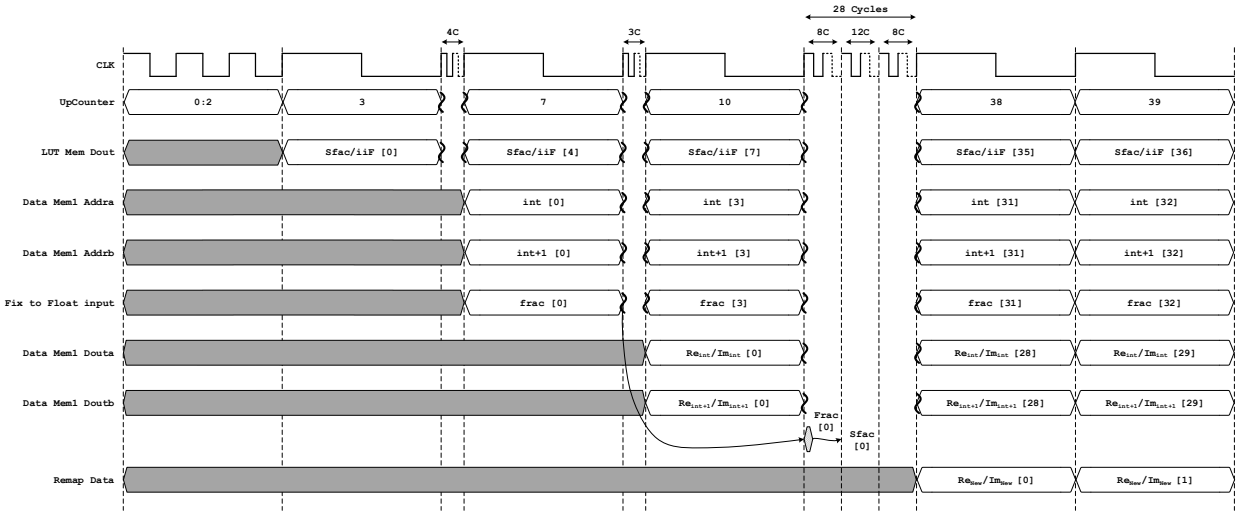


Figure 4.14: Timing diagram of Remap module.

4.3 Key Performance Features

This section focuses on highlighting key performance features of the proposed hardware design, which include *computation and communication overlap*, *inter-module pipeline*, *inter-angle pipeline*, and *high reconstruction frame rate*.

- **Computation and communication overlap:** Communication overlap refers to execution of different processes (TFFT, XFFT, Remap, etc.) simultaneously, while communication overlap refers to data transferring between component simultaneously without causing a conflict. These two features, as implemented in our architecture, enable faster data processing and higher frame throughput.

- **Inter-module pipeline:** Inter-module pipeline determines how data processing is parallelized within one frame. All components in the proposed design support inter-module pipeline, resulting in gapless output with a throughput of one output data unit per clock cycle for each of the processes within one frame.

- **Inter-angle pipeline:** Inter-angle pipeline is relevant to image reconstruction that involves CPWC using multi-angle data. Using more PW angles typically lead to better image quality but also increase processing time for each compounded frame. Our inter-angle pipeline reduces this processing overhead by enabling simultaneous execution of TFFT, XFFT, Remap, and Compound processes for different angles pertaining to the same compounded frame under reconstruction frame.

- **High reconstruction frame rate:** Frame rate refers to the number of frames per second (fps) the core can process. A higher data acquisition rate translates into higher temporal resolution when capturing fast movements in the imaged medium, which is essential for echocardiography and sonoelastography applications. Achieving a comparably high reconstruction frame rate (e.g., above 1000 fps for ultrafast imaging) requires efficient data handling, memory utilization, and coordination among internal components. Thanks to careful utilization of local memories, stage-by-stage data processing, and various levels of pipelining, our architecture achieves more than 1300 fps for single-angle beamformed data.

Chapter Five

Proposed Control Mechanisms

This chapter delves into the control mechanisms that govern our architecture. First, we examine Global control mechanisms and their roles in our hierarchical pipeline. Next, we explore stage-level control mechanisms, shedding light on the influence of local managers and their coordination with a Global controller. Then, we provide insights into local control mechanisms, covering the functions of local subordinates and their contributions to stage-level coordination. The chapter concludes with a summary of supported flexibility features of our architecture.

5.1 Global Control

The Global controller serves as the central control hub within the core. Its primary responsibilities include initiating operations of each module, managing AXI bus interfaces and external memory communication, and overseeing the ping-pong switch between pairs of memories. The modules under its control include **`TFFT`**, **`XFFT`**, **`Remap`**, **`Compound`**, **`IXFFT`**, and **`ITFFT`**. Additionally, the Global controller keeps track of angle counting and switches between the **`Remap`** and **`Compound`** modules. Given that the execution time of different modules may vary slightly due to the nature of their processes, the Global controller synchronizes the `start` signals of these modules to align with the one with the longest execution time. This synchronization enables seamless computation pipelining.

Pseudo code 1 Global controller

```
1   case halt:
2       if Core Enable = 1 and Core Reset = 1 then
3           Go to setup
4       end if
5   end case
6   case setup:
7       Reset all start signals
8       Reset all modules
9       Get number of angles
10      Reset angle counter to zero
11      Set initial memories switches to zero
12      Request data from External memory
13      while Core Start = 0 do wait
14      end while
15      Enable TFFT start register
16      Go to run
17  end case
18  case run:
19      while at least one [module] Process is running do
20          if TFFT Process = Done then
```

```
21                          Enable XFFT start register
22                          if Core Start = 1 then
23                                  Enable TFFT start register
24                          end if
25                  end if
26                  if XFFT Process = Done then
27                          if angle counter = 0 then
28                                  Enable Remap start register
29                          else
30                                  Enable Compound start register
31                          end if
32                  end if
33                  if angle counter = number of angles then
34                          if Remap Process = Done or Compound Process = Done then
35                                  Enable IXFFT start register
36                          end if
37                  end if
38                  if IXFFT Process = Done then
39                          Enable ITFFT start register
40                  end if
41                  if Sync = 1 then
42                          Start the processes that have enabled start registers
43                          Disable all start registers
44                  end if
45                  Switch between paired memories in stages that receive done signal
46          end while
47          Go to halt
48  end case
```

Figure 5.1: Pseudo code for Global control.

Figure 5.1 lists key actions performed by the Global controller. Each case in Figure 5.1 corresponds to a logical state: the controller remains in that state until a state-switching **Go to** statement is reached. After the core completes its initial setup, it waits for the core `start` signal to begin its operations. The core `start` signal is software-controlled and signifies that data are available in the external memories, allowing the core to commence calculations. The Global controller communicates with the stage-level controllers through `start` and `done` signals specific to each module. When the core is in the run state, the module processes are executed in parallel. The stage-level controllers receive the `start` signal as input and return the `done` signal as output, with both signals being high for one cycle.

The `start` signal indicates that one frame of data is ready for processing, while the `done` signal signifies that the entire frame has been processed and stored in the related memory. As a result, the start of each module, except **`TFFT`**, depends on the previous module's `done` signal. To synchronize the execution times of the modules, rather than directly feeding the `done` signal to

the `start` signal of the next stage, these start registers are enabled instead. This allows the `Sync` signal to initiate all processes simultaneously when activated.

The `Sync` signal is connected to the `done` signal of the currently running module that has the longest execution time. Consequently, if that module has completed processing all frames, the `Sync` signal is derived from the next module with the longest execution time. It's important to note that the execution times are slightly different, with the order from the longest to shortest being **`TFFT`**, **`Compound`**, **`Remap`**, and the other three modules (**`XFFT`**, **`IXFFT`**, and **`ITFFT`**) that have the same (shortest) execution time.

The Global controller operates at the frame sequence level. In the single-angle scenario, it runs only the Remap process (lines 27-29, Figure 5.1). In the multi-angle scenario, it switches between the Remap and Compound processes based on the current frame angle (lines 27-31, Figure 5.1). After one frame is constructed, whether from one angle or multiple angles, it initiates the IXFFT process (lines 33-37, Figure 5.1).

The number of angles setting in Figure 5.1, indicates how many frames should be compounded to form one fully reconstructed frame. The angle counter is incremented upon completion of each Remap or Compound process. It resets to zero when it reaches the number of angles specified by the user. Currently, the angle counter limit is set to 16, but it can be increased in a straightforward way if needed for the future design revisions.

Additionally, the Global controller manages the read and write modes of memory pairs between stages. Initially, it sets memories with index one (**`MemT`**$_1$, **`MemX`**$_1$, **`MemRC`**$_1$, **`MemIX`**$_1$) to write mode and memories with index two (**`MemT`**$_2$, **`MemX`**$_2$, **`MemRC`**$_2$, **`MemIX`**$_2$) to read mode. After completing a stage process, it swaps the read and write mode for the corresponding memory pair of that process.

5.2 Stage-Level Control

The Stage-level controllers are responsible for managing the related module calculations for one frame. Each module has its own stage-level controller. As mentioned earlier, these controllers communicate with the Global controller using two main signals: `start` and `done`. Additionally, the stage-level controllers request data from the external memories through the Global controller. The basic steps of all stage-level controllers is illustrated in Figure 5.2.

Pseudo code 2 Stage-level controller

```
1   case halt:
2       Wait for the start signal
3       Go to setup
4   end case
5   case setup:
6       Clear registers, counters
7       Configure the module structure as required
8       Go to run
9   end case
10  case run:
```

```
11          for i = 0 to Iteration Number/2-1do
12              for j = 0 to NData -1do
13                  Load data into the module
14                  Set input Tvalid high for the module
15                  if j = NData -1 then
16                      Set input Tlast high for the module for one cycle
17                  end if
18              end for
19          end for
20          Go to done
21  end case
22  case done:
23          Send the done signal to the Global controller
24  end case
```

Figure 5.2: Pseudo code for stage-level control.

There are six different stage-level controllers, namely TFFT_Process, XFFT_Process, Remap_Process, Compound_Process, IXFFT_Process, and ITFFT_Process. The basic structure of all these stage controllers is illustrated in Figure 5.2. Information about the *Iteration Number* and *NData* of each process can be found in Table 4.2. `Tvalid` is used by the stage controller to notify the local controller that it is ready to provide data, and `Tlast` indicates the transfer of the last sample of the frame, which will be explained in detail in the next subsection.

The communication protocol between the stage controller and the local controller is handshake-based. The stage controller sends `Tvalid_s`, `Tdata_s`, and `Tlast_s`, and receives `Tvalid_m`, `Tdata_m`, and `Tlast_m` in return. The `Tvalid` signals indicate that the input data (`Tdata_s`) or output data (`Tdata_m`) of the module is valid. The `Tlast` signals are used to indicate the last data transfer: they go high for one cycle when the number of input or output data transfers reach *NData* of that process. Therefore, each stage controller is responsible for sending data from memory to the module and storing the module's output back in memory.

As mentioned earlier, the memories used for reading data and writing data are controlled by the Global controller. However, the reading and writing process itself is controlled by the stage-level controllers. In addition to sending and receiving data, the stage-level controller generates read and write addresses required by the module process based on the information provided in Table 4.1. The stage-level controller also switches from reading memory to sending zeros or vice versa for the **`XFFT`** and **`ITFFT`** modules that require zero inputs during some iterations.

In addition to the general structure described above, each module has its own specific control configuration. They are explained in the following subsections, with additional details about other signals driven by the respective stage-level controllers.

5.2.1 TFFT/ITFFT and XFFT/IXFFT

The **TFFT** and **ITFFT** modules are based on the Xilinx FFT™ IP core. During the setup step of **TFFT** and **ITFFT**, the register bit for the forward transform and inverse transform is set accordingly. The core then sends back an acknowledgment confirming that the configuration has been set. The stage-level controller waits for this acknowledgment and proceeds with the rest of its procedure.

It's important to note that **DSPS** block operates under the **TFFT** module. Since **DSPS** utilizes the output of the **TFFT** module directly, the `Tdata_m`, `Tvalid_m`, and `Tlast_m` signals of **TFFT** are connected to the `Tdata_s`, `Tvalid_s`, and `Tlast_s` signals of **DSPS**. Consequently, the **TFFT** module uses the `Tdata_m`, `Tvalid_m`, and `Tlast_m` signals of **DSPS** for communication. Additionally, the **TFFT** module sends a data request to the Global controller for phase data required by **DSPS**.

The configuration settings for the **XFFT** and **IXFFT** stage-level controller are the same as those for the **TFFT** and **ITFFT** stage-level controller, respectively.

5.2.2 Remap/Compound

The **Remap** and **Compound** modules require two additional inputs, `RLUT1` and `RLUT2`, from external memories. These two stage-level controllers send requests to the Global controller to obtain this data. It's important to note that both the **TFFT** and **Remap/Compound** modules request `PLUT` and `RLUT1` data from the same external memory. However, these requests occur at different times and do not conflict with each other. The Global controller manages the switching between `PLUT` and `RLUT1` data accesses.

The structure of the Compound stage-level controller is similar to that of the Remap stage-level controller, with slight differences. Both stage-level controllers read data from the previous stage memory (namely **$MemX_1$** or **$MemX_2$**) and send data to the **Remap1** and **Remap2** blocks. However, in the case of Compound stage-level controller, there is an extra data request from the current read stage memory (namely **$MemRC_1$** or **$MemRC_2$**) to be compounded with the data produced by **Remap1** and **Remap2** blocks. Also, in the Remap stage-level controller, the control signals output from **Remap1** and **Remap2** blocks are directly connected to the Remap stage-level controller to write the data to the current write stage memory (**$MemRC_1$** or **$MemRC_2$**). In contrast, in the Compound stage-level controller, these control signals go to the **Compound1** and **Compound2** blocks, and the control output of these blocks is used for the Compound stage-level controller to the current write stage memory (**$MemRC_1$** or **$MemRC_2$**).

5.3 Local Control

The local controller refers to the control mechanism within each module and how stage-level controllers communicate with them. The basic control communication, as mentioned before, uses

a handshake protocol. In this section, for each IP core, a table of signals is provided to describe the related control mechanism.

5.3.1 FFT Core

The TFFT/ITFFT/XFFT/IXFFT modules use the FFT IP core with different configurations. The FFT core signals are shown in Table 5.1 [45].

Table 5.1: FFT core signal pinout.

Signal Name	Direction	Bit-width	Description
aclk	Input	1	Rising-edge clock.
aresetn	Input	1	Active-Low synchronous reset. Requires a minimum active pulse of two cycles.
s_axis_config_tdata	Input	16	Configuration data input channel (`Tdata_s`) carrying configuration information.
s_axis_config_tvalid	Input	1	Configuration data valid input signal (`Tvalid_s`) indicating data availability.
s_axis_config_tready	Output	1	Configuration data ready output signal (`Tready_s`) indicating the core's readiness to accept data.
s_axis_data_tdata	Input	64	Data input channel (`Tdata_s`) carrying unprocessed sample data, including real and imaginary components.
s_axis_data_tvalid	Input	1	Data valid input signal (`Tvalid_s`) indicating sample data availability.
s_axis_data_tready	Output	1	Data ready output signal (`Tready_s`) indicating the core's readiness to accept sample data.
s_axis_data_tlast	Input	1	Last sample input signal (`Tlast_s`) indicating the last sample of the frame.
m_axis_data_tdata	Output	64	Data output channel (`Tdata_m`) carrying processed sample data, including real and imaginary components.
m_axis_data_tvalid	Output	1	Data valid output signal (`Tvalid_m`) indicating processed sample data availability.
m_axis_data_tlast	Output	1	Last sample output signal (`Tlast_m`) indicating the last sample of the frame produced by the core.

In Figure 5.2, the stage controller initializes the module's configuration. For the FFT core, this configuration is received by the core via `s_axis_config_tdata`. The configuration data includes parameters such as cyclic prefix, Forward/Inverse (FWD/INV), transform point size (NFFT), and scaling schedule. In the proposed design, only the FWD/INV bit is used, and the other configuration signal bits are set to zero. According to the handshake protocol, when configuring

or sending data to the FFT core, the stage controller awaits until the associated `Tready_s` signal transitions to a high state. Once this condition is met, the stage controller proceeds to provide the configuration or data and then sets the corresponding `Tvalid_s` signal to a high state.

As the proposed architecture is designed for high-frame-rate real-time applications, all FFT cores are implemented in real-time mode. In this mode, the `Tready_s` and `Tvalid_s` signals on the input data line are used only for the first data transfer. Afterward, the FFT core disregards `Tvalid_s` on the data input, and the stage controller ignores the `Tready_s` signal as the remaining data is transferred to the core.

Furthermore, the `Tready_s` signal on the Data output channel, asserted by the external subordinate to indicate readiness to accept data, is unused in real-time mode. Consequently, once the output `Tvalid_m` signal goes high, the output data is written to memory. The output `Tvalid_m` signal serves as the write enable for the memory, except for TFFT, where the `Tvalid_m` is connected to `Tvalid_s` of **`DSPS`**.

5.3.2 DSPS

The DSPS core signals are shown in Table 5.2.

Table 5.2: DSPS core signal pinout.

Signal Name	Direction	Bit-width	Description
aclk	Input	1	Rising-edge clock.
aresetn	Input	1	Active-Low synchronous reset. Requires a minimum active pulse of two cycles.
en	Input	1	Active-High enable signal.
s_axis_data_tdata_phase	Input	64	Phase input channel (`Tdata_s`) carrying shift angles of two columns (one odd-indexed and the other even-indexed).
s_axis_data_tready_phase	Output	1	Phase data ready output signal (`Tready_s`) indicating the core's readiness to accept data.
s_axis_data_tlast_phase	Input	1	Last Phase input signal (`Tlast_s`) indicating the last sample of the frame.
s_axis_data_new_col	Input	1	New column signal for the Phase Input channel, indicated by the external manager when TFFT outputs the next column data, signaling the core to reset incremental angle calculation and switch to the next shift angle.
s_axis_data_tdata	Input	64	Data input channel (`Tdata_s`) carrying unprocessed sample data, including real and imaginary components.
s_axis_data_tvalid	Input	1	Data valid input signal (`Tvalid_s`) indicating sample data availability.

s_axis_data_tlast	Input	1	Last sample input signal (`Tlast_s`) indicating the last sample of the frame.
m_axis_data_tdata	Output	64	Data output channel (`Tdata_m`) carrying processed sample data, including real and imaginary components.
m_axis_data_tvalid	Output	1	Data valid output signal (`Tvalid_m`) indicating processed sample data availability.
m_axis_data_tlast	Output	1	Last sample output signal (`Tlast_m`) indicating the last sample of the frame produced by the core.

The input and output data channel signals (`Tdata`, `Tvalid`, and `Tlast`) operate similarly to the FFT core. It's important to note that in real-time mode, there is no `Tready_s` signal on the data channel. Additionally, as previously mentioned, the output of the TFFT core is directly connected to the input of the **DSPS** module.

The behavior of the `s_axis_data_tdata_phase`, `s_axis_data_tready_phase`, and `s_axis_data_tlast_phase` signals is slightly different compared to data channel signals. The **DSPS** module sets `s_axis_data_tready_phase` high to indicate its readiness to accept data. The external manager then begins sending phase shift angles, and **DSPS** stores these angles in its local memory. On the final angle data transfer, the external manager asserts `s_axis_data_tlast_phase` high for one cycle, signaling to **DSPS** that it has completed sending angles. Consequently, **DSPS** sets `s_axis_data_tready_phase` back to low.

As **DSPS** performs its calculations, it retrieves the initial (column-specific) phase shift angles stored in **Phase Mem**. Each time the `NewCol` signal is asserted, the core progresses to the next angle in **Phase Mem**. Once all the stored angles have been utilized, signifying the completion of the calculation for a whole frame, **DSPS** sets `s_axis_data_tready_phase` back to high to indicate its readiness to receive a new set of angles.

5.3.3 Remap

The Remap core signals are shown in Table 5.3.

Table 5.3: Remap core signal pinout.

Signal Name	Direction	Bit-width	Description
aclk	Input	1	Rising-edge clock.
aresetn	Input	1	Active-Low synchronous reset. Requires a minimum active pulse of two cycles.
en	Input	1	Active-High enable signal.

s_axis_data_tdata	Input	64	Data input channel (`Tdata_s`) carrying unprocessed sample data, including real and imaginary components.
s_axis_data_tdata_LUT	Input	64	Data input channel (`Tdata_s`) carrying the LUT data: interpolation point and scaling factor.
s_axis_data_tvalid	Input	1	Data and LUT valid input signal (`Tvalid_s`) indicating sample and LUT data availability.
s_axis_data_tlast	Input	1	Last sample input signal (`Tlast_s`) indicating the last sample and LUT of the frame.
m_axis_data_tdata	Output	64	Data output channel (`Tdata_m`) carrying processed sample data, including real and imaginary components.
m_axis_data_tvalid	Output	1	Data valid output signal (`Tvalid_m`) indicating processed sample data availability.
m_axis_data_tlast	Output	1	Last sample output signal (`Tlast_m`) indicating the last sample of the frame produced by the core.

The input and output data channel signals, including `Tdata`, `Tvalid`, and `Tlast`, behave in a similar manner as in the previously described modules. Additionally, `Tdata` for the LUT also relies on the same `Tvalid` and `Tlast` signals as the input data channel because both the LUT and data vectors have a size of 2048. Therefore, the external manager should provide the LUT and data vectors at the same time.

The `Tvalid_m` and `Tdata_m` outputs serves as the write enable and data input for the memory in the Remap stage-level controller.

5.3.4 Compound

The Compound core signals are shown in Table 5.4.

Table 5.4: Compound core signal pinout.

Signal Name	Direction	Bit-width	Description
aclk	Input	1	Rising-edge clock.
aresetn	Input	1	Active-Low synchronous reset. Requires a minimum active pulse of two cycles.
en	Input	1	Active-High enable signal.
s_axis_data_tdata_Remap	Input	64	`Tdata_s` for the Remap Input channel. Carries the output data from Remap.

s_axis_data_tdata_Mem	Input	64	`Tdata_s` for the Memory Input channel. Carries the last compounded data from memory.
s_axis_data_tvalid_Remap	Input	1	Data valid input signal (`Tvalid_s`) indicating sample and memory data availability.
s_axis_data_tlast	Input	1	Last sample input signal (`Tlast_s`) indicating the last sample of the frame.
m_axis_data_tdata	Output	64	`Tdata_m` for the Data Output channel. Carries the compounded data.
m_axis_data_tvalid	Output	1	Data valid output signal (`Tvalid_m`) indicating compounded data availability.
m_axis_data_tlast	Output	1	Last sample output signal (`Tlast_m`) indicating the last compounded data of the frame produced by the core.

The input `Tvalid_s` and `Tlast_s` signals of the **`Compound`** core are used for both the Remap and Memory Input channels. The `s_axis_data_tdata_Remap` signal is the direct output of the **`Remap`** module, while `s_axis_data_tdata_Mem` refers to the previous data stored in the memory, which could be either a single-angle Remap data or multi-angle Remap data that has already been compounded.

The Compound stage-controller synchronizes the memory read data with the output of the Remap blocks and inputs them both simultaneously into the **`Compound`** module. The `Tvalid_m` and `Tdata_m` outputs of the **`Compound`** module are used as the write enable and data input for the memory in the Compound stage-level controller.

5.4 Flexibility Features

The flexibility of the proposed hardware architecture is a pivotal attribute that significantly influences its performance and adaptability to varying applications and scenarios. This section explores the key flexibility features, showcasing its seamless accommodations of diverse configurations and use cases.

Our design choice to use externally precomputed PLUT and RLUT information is motivated by flexibility considerations. If a user requires different phase shifts, or more importantly, different remapping and scaling (e.g., filtering) patterns, `PLUT` and `RLUT` can simply be updated accordingly.

The core offers stage-level extensibility, enabling the straightforward addition of new modules to the core. These modules can be added either after or before the existing modules without altering the architecture's overall structure. This flexibility facilitates the application of computational processes for a wide range of scenarios.

To add a new module after an already existing one, the stage-level controller's output signals (`Tvalid_m`, `Tdata_m`, and `Tlast_m`) from the existing module should be connected to the newly introduced module's input signals (`Tvalid_s`, `Tdata_s`, and `Tlast_s`). The new module's output signals (`Tvalid_m`, `Tdata_m`, and `Tlast_m`) are then used by the stage-level controller instead of those from the already existing module.

The same principle applies when adding a new module before an already existing one. The connections should follow a cascading order, ensuring that the `<Signals>_s` of the first module and the `<Signals>_m` of the last module in a stage are connected to the corresponding stage-level controller. Additionally, intermediate `<Signals>_s` and `<Signals>_m` should cascade to each other in the order of calculations.

It's important to note that new modules should have the same data ordering (either row-wise or column-wise) as that of the stage it's added to, ensuring proper data flow and compatibility.

Chapter Six

FPGA Implementation Approach

In this chapter, we will provide descriptions of all the Xilinx® IP components used in the design.

6.1 FFT

The FFT core provided by Xilinx® LogiCORE™ IP implements the Cooley-Tukey FFT algorithm, which is widely used for computing the Discrete Fourier Transform. This core is capable of performing both forward and inverse Discrete Fourier Transform calculations for *N*-point transforms. The value of *N* can be chosen as 2^m, where *m* can vary from 3 to 16. In scenarios involving single-precision floating-point inputs, the core handles data input in the form of a vector containing *N* complex values. These values are represented as a Re/Im pair of 32-bit floating-point numbers, while the phase factors are encoded as 24- or 25-bit fixed-point numbers. It's important to note that the input data is structured in a natural order, and the resulting output data can be configured to be in either a natural order or a bit/digit-reversed order, providing flexibility to adapt to various processing requirements [45].

Table 6.1 illustrates the design customizations that were chosen to generate the FFT core, tailored to meet the specific computational demands of our application. Two distinct cores have been generated for the axial and lateral Fourier transforms. The selected parameters for these two cores are identical, except for the transform length. The architecture choice and throttle scheme have been configured for real-time streaming. Additionally, the optimization parameter has been set to resource optimization, so that the FFT cores do not constitute the primary bottleneck in the overall design.

Table 6.1: FFT IP core design customizations.

Design modifications	Selected Parameter
Transform length	4096 (axial), 256 (lateral)
Architecture choice	Pipelined, Streaming I/O
Data format	Floating point
Phase factor width	24
Output order	Natural order
Throttle scheme	Real time
Optimize option (Complex multipliers)	3-multipliers structure (resource optimization)
Optimize option (Butterfly arithmetic)	CLB logic (resource optimization)

6.2 DDS Compiler Core

The Xilinx® LogiCORE™ IP Direct Digital Synthesizer (DDS) Compiler core is designed for high-performance Phase Generation and Phase to Sinusoid Conversion. The DDS core comprises two elements: a Phase Generator and a SIN/COS Lookup Table, which enable the conversion of

phase information into sinusoidal waveforms. The IP provide the flexibility to utilize these components either independently or in conjunction [46].

The Table 6.2 displays the chosen design customizations for generating the DDS Compiler Core. To enhance core accuracy, we've selected the maximum phase and output width. We've also configured the noise shaping as Taylor Series Corrected, utilizing discarded fractional bits from phase truncation to compute corrections, thereby achieving higher precision. Furthermore, to enable the provision of new sin/cos values in each clock cycle, the optimization goal and latency settings were carefully chosen to meet this requirement.

Table 6.2: (DDS) Compiler IP core design customizations.

Design modifications	Selected Parameter
Configuration option	Sin cos LUT only
Noise shaping	Taylor Series corrected
Phase width	25 (max)
Output width	26 (max)
Output selection	Sine and Cosine
Optimization goal	Speed
Latency	9

6.3 Floating-Point Operator Core

The Xilinx® Floating-Point™ Operator IP core enables FPGA-based floating-point arithmetic operations. This core offers customization options for operation, word length, latency, and interface. There are different floating-point operations that are supported by this IP core [47]. Table 6.3 displays various cores relevant to our design, generated using this IP. The latency for all cores is configured to ensure a throughput of one, meaning the core delivers valid data in every clock cycle after the initial latency.

Table 6.3: Floating-Point Operator IP core design customizations.

Core	Design modifications	Selected Parameter
1	Operation selection	Float add
	Precision of inputs	Single
	DSP Slice Usage	No usage (Logic only)
2	Operation selection	Float subtract
	Precision of inputs	Single
	DSP Slice Usage	No usage (Logic only)
3	Operation selection	Float multiply
	Precision of inputs	Single
	DSP Slice Usage	Max usage (3x DSP48E2)
4 (in DSPS)	Operation selection	Fix to float
	Precision of input	1-bit integer, 25-bit fraction
	Precision of output	Single

5 (in Remap)	Operation selection	Fix to float
	Precision of input	1-bit integer, 23-bit fraction
	Precision of output	Single
6 (in DSPS)	Operation selection	Float to fix
	Precision of input	Single
	Precision of output	2-bit integer, 25-bit fraction
7 (in Remap)	Operation selection	Float to fix
	Precision of input	Single
	Precision of output	12-bit integer, 23-bit fraction

6.4 Embedded Building Blocks

6.4.1 BlockRAMs™

The Xilinx® LogiCORE™ IP Block Memory Generator core serves as a memory constructor, harnessing the embedded block RAM resources found in Xilinx® FPGAs to generate memories. This adaptable Block Memory Generator core supports both Native and AXI4 interfaces, with AXI4 configurations originating from the settings established in the Native interface [48].

The Table 6.4 displays various cores that have been created using this IP core. In our design, all Block Memory Generator instances employ the Native interface to adjust the bus and data transfer to match the unique requirements of each module. Both output registers are enabled to synchronize the BMGs with the desired clock signal. Additionally, the operating mode determines how the RAM behaves during simultaneous read and write operations.

Table 6.4: BlockRAMs IP core design customizations.

IP Core Instances	Design modifications	Selected Parameter
Next column Mem (DSPS)	Memory type	Single port ram
	Write/read depth	2048
	Operating Mode	READ_FIRST
Phase Mem (DSPS)	Memory type	Single port ram
	Write/read depth	128
	Operating Mode	NO_CHANGE
Data Mem1/Data Mem2 (Remap)	Memory type	True dual port ram
	Write/read depth	2048
	Operating Mode	NO_CHANGE
LUT Mem (Remap)	Memory type	Single port ram
	Write/read depth	2048
	Operating Mode	READ_FIRST
Same for all instantiations above	Interface type	Native
	Algorithm option	Minimum area
	Write/read width	64
	Output Registers	Enabled Primitives output register. Enabled Core output register.

Total read latency	3 cycles

While the BlockRAMs offer efficient memory operations, they come with limitations in terms of their write/read depth. However, the UltraRAM™ memories provide larger memory size at the cost of some speed. It's noteworthy that all memory pairs within the stage utilize UltraRAM™ to accommodate larger storage needs.

6.4.2 UltraRAM™

Inside the FPGA device, UltraRAM™ blocks operate as single-clock, synchronized units, organized in columns that can be grouped together. UltraRAM™ is known for its adaptability and memory density. Each UltraRAM™ block is structured as a 4Kb x 72 memory, capable of storing up to 288K bits of data. Notably, UltraRAM™ offers an eightfold increase in capacity compared to standard BlockRAM™. UltraRAM™ is also beneficial in terms of its dual-port design, allowing both ports to access the entire 4Kb x 72-bit memory space [49].

Each port can independently perform one read or write operation per clock cycle. It's important to note that each port can execute only one operation in a given cycle, either a write or a read. When a write operation is performed, the read outputs remain unchanged and retain their previous values.

6.4.3 DSP48E2™

The Xilinx® UltraScale™ FPGA family leverages the power of embedded DSP48E2™ blocks, serving as a fundamental building block for DSP applications. They provide dedicated hardware support for binary multipliers and accumulators crucial in DSP tasks, enabling optimal performance. The utility of DSP resources extends beyond traditional digital signal processing tasks. These resources also enhance the speed and efficiency of various other applications, such as wide dynamic bus shifters, memory address generators, wide bus multiplexers, and memory mapped I/O registers [50].

6.5 I/O Interface

As previously mentioned, the proposed design assumes the AXI4 interface for communication with external memories. The AXI4 interface, short for Advanced eXtensible Interface 4, represents the fourth generation of the AMBA™ interface specification developed by ARM®, serving as a vital on-chip communication bus protocol [51].

However, when it comes to harnessing the capabilities of DDR4 RAMs as external memory resources, an additional layer of communication becomes necessary. This layer facilitates the control of these RAMs through FPGA I/O resources while also providing an internal AXI interface within the FPGA itself. Bridging this gap is accomplished through the utilization of the Memory Interface Generator Xilinx IP core.

This Xilinx® UltraScale™ architecture-based FPGA Memory IP core is a comprehensive solution encompassing both a pre-engineered controller and a physical layer (PHY). Its primary purpose is

to establish connections between user-designed FPGA configurations and a diverse range of memory devices, including DDR3 and DDR4 SDRAM, LPDDR3 SDRAM, QDR II+ SRAM, QDR-IV SRAM, and RLDRAM 3 devices [52].

Chapter Seven

Evaluation Results

In this chapter, we present the evaluation results of our FPGA-based architecture from various aspects. We provide a comprehensive analysis of FPGA implementation characteristics, including resource utilization, performance metrics, and power consumption. The core functionality is verified by simulating the core in Xilinx Vivado®, generating images based on the real-world ultrasound dataset, and comparing them to reference images generated a software beamformer. Our image-to-image comparisons are based on well-known metrics such as SSIM (Structural Similarity Index), PSNR (Peak Signal-to-Noise Ratio), and MSE (Mean Square Error).

We developed the entire design, along with the testbenches, using the VHDL language. The verification, synthesis, and implementation stages are conducted within the Xilinx Vivado® environment. For testing and development, we opt for the Xilinx® Ultrascale+ family, as it offers UltraRAM™ blocks, catering to applications that involve continuous streaming of large data volumes. Moreover, the Ultrascale+ family is fabricated using the TSMC 16-nm process, aligning it closely with current ASIC technologies and rendering it a prime candidate for prototyping.

7.1 Test Image Data

Our hardware simulations make use of the Plane Wave Imaging Challenge in Medical Ultrasound (PICMUS) framework [53]. It provides a standardized tool for the objective evaluation of various beamforming methods. It has been specifically designed for evaluating Coherent Plane Wave Compounding (CPWC) techniques.

The PICMUS experimental datasets were acquired using the Verasonics Vantage 256 research scanner equipped with a linear L11-4v probe [53]. The probe specifications are shown in Table 7.1.

Table 7.1: L11-4v probe specifications [53].

Characteristic	Value
Pitch	0.30 mm
Element width	0.27 mm
Element height	5 mm
Elevation focus	20 mm
Number of elements	128
Aperture	38.4 mm

Table 7.2 provides the parameters characterizing the transmitted PW pulses used to insonify the imaged medium during data acquisition.

Table 7.2: Transmitted PW pulse parameters [53].

Parameter	Value
Transmit frequency	5.208 MHz
Sampling frequency	20.832 MHz
Pulse bandwidth	67%
Transmit voltage	30 V
Excitation	2.5 cycles

The PICMUS acquisition sequence consisted of 75 steered PW transmissions, whose angles were regularly spaced between −16° and 16°, which corresponds to an angle increment of 0.43°. As our test case, we chose a three-angle RF channel dataset acquired when imaging the longitudinal view of the carotid artery section. Specifically, the three PW angles under consideration were −16°, 0°, and 16°.

7.2 FPGA Implementation Characteristics

7.2.1 Device and Setting

The target Xilinx® FPGA in use was the **XCVU13P-FHGA2104-3-E** device. The specifications for this device are provided in Table 7.3. Our design was synthesized and implemented in the Xilinx Vivado® v2019.1 development environment with the default strategy for both synthesis and implementation.

Table 7.3: XCVU13P specifications [54].

Device Resources	Limit
System Logic Cells (K)	3,780
CLB Flip-Flops (K)	3,456
CLB LUTs (K)	1,728
Max Distributed RAM (Mb)	48.3
Total Block RAM (Mb)	94.5
UltraRAM (Mb)	360.0
DSP Slices	12,288
Peak INT8 DSP (TOP/s)	38.3
PCIe® Gen3 x16	4
150G Interlaken	8
100G Ethernet w/ KR4 RS-FEC	12
Max. Single-Ended HP I/Os	832
Max. Single-Ended HD I/Os	0
GTY 32.75Gb/s Transceivers	128

7.2.2 Timing Summary

The specified target clock frequency, set at 384 MHz (with a cycle time of 2.6 ns) in the constraints, has been successfully met, as depicted in the Figure 7.1.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.027 ns	Worst Hold Slack (WHS):	0.011 ns	Worst Pulse Width Slack (WPWS):	0.650 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	498731	Total Number of Endpoints:	498731	Total Number of Endpoints:	125348

All user specified timing constraints are met.

Figure 7.1: Design Timing Report.

At this clock frequency of 384 MHz, the most time-consuming process, TFFT, takes 725us to complete, as showcased in Figure 7.2 and Figure 7.3. This results in an image reconstruction rate of 1379 frames per second (fps). Figure 7.2 and Figure 7.3 further indicate that the initial latency of our pipelined architecture is less than 4.3ms.

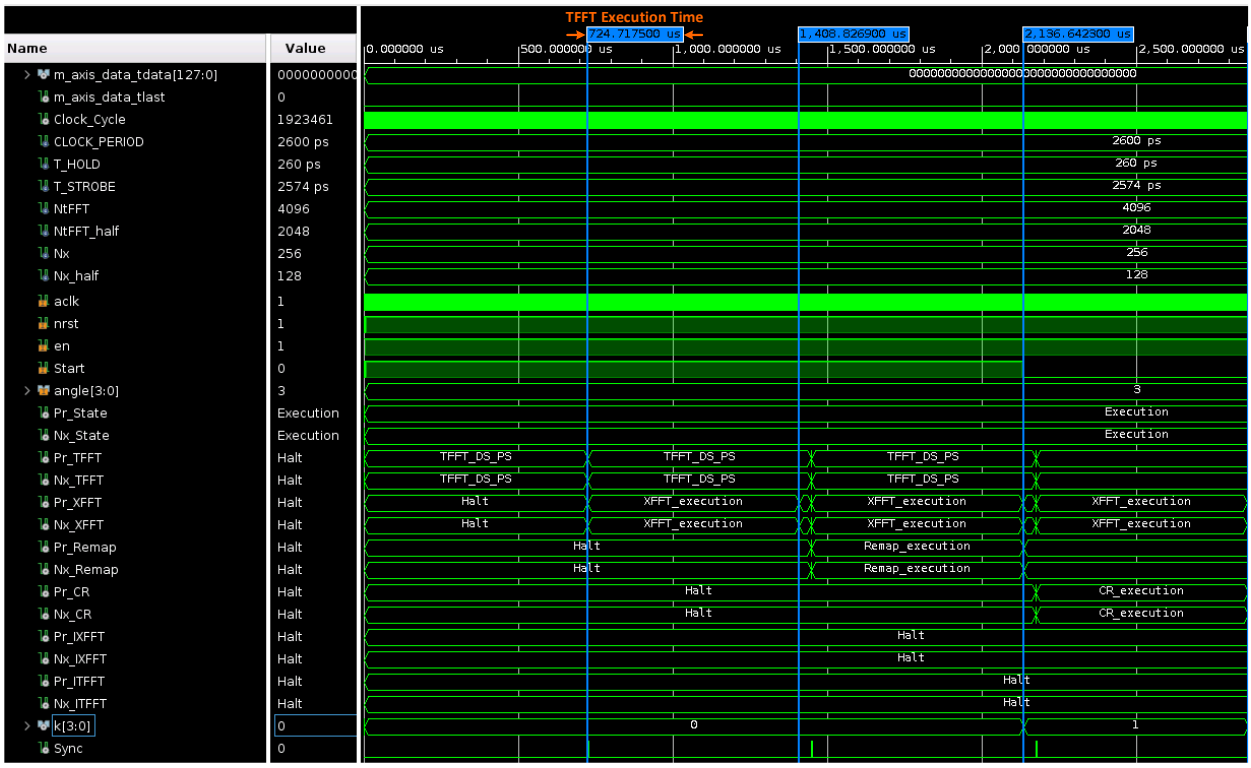


Figure 7.2: Example of waveforms during 3-angle image reconstruction (part one).

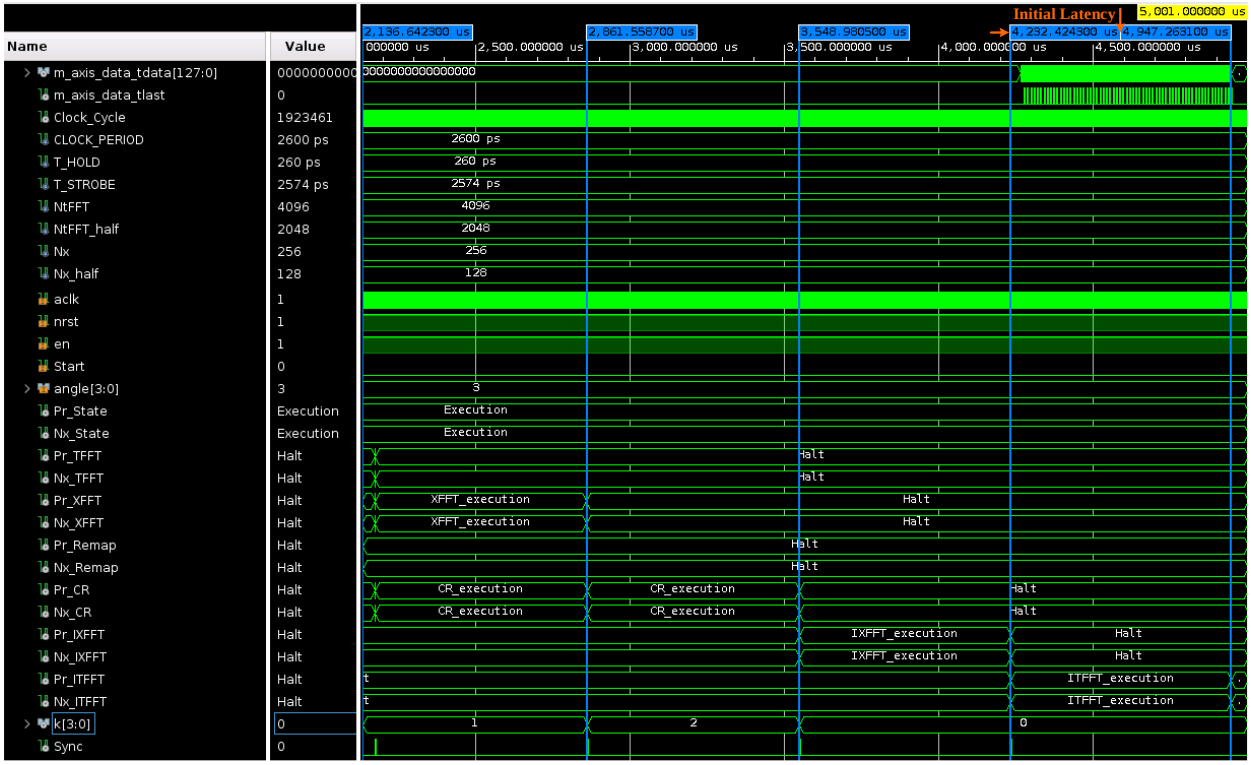


Figure 7.3: Example of waveforms during 3-angle image reconstruction (part two).

7.2.3 Resource Utilization

The post-implementation design resource utilization is summarized in Figure 7.4. Currently, the design utilizes 70% UltraRAM™ (URAM) and 40% I/O pins. Less than 5% of other resources, primarily dedicated to computation, are employed.

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	54714	1728000	3.17
LUTRAM	12125	791040	1.53
FF	105202	3456000	3.04
BRAM	137.50	2688	5.12
URAM	896	1280	70.00
DSP	238	12288	1.94
IO	335	832	40.26
BUFG	1	1344	0.07

Figure 7.4: Design resource utilization.

Such resource usage indicates the potential for achieving even higher frame rates by replicating the computational components of the design while maintaining the same UltraRAM™ utilization. In this scenario, the UltraRAM™ size remains constant but is divided into smaller segments to simultaneously accommodate additional computational processes. The relationship between

increasing the number of the components and the frame rate improvement is expected to be almost linear. For instance, by doubling the computational components of the current design, the frame rate would also almost double (reaching 2600 fps).

The only potential bottleneck to consider is the data transfer rate from external memories. To address this, it may be necessary to employ data compression or to increase I/O pin utilization to facilitate a higher data transfer rate, ensuring it meets the architectural requirements.

7.2.4 Power Consumption

As illustrated in Figure 7.5, the current design consumes 20.002 watts of dynamic power and 3.750 watts of static power, with the total on-chip power consumption of 23.752 watts. Notably, the computational power, associated with the logic and DSP components, accounts for 20% of the dynamic power consumption. A substantial portion of the dynamic power, approximately 28%, is consumed by communication tasks involving signals and I/O operations, which is not surprising given that significant data traffic occurs at each stage. Furthermore, the extensive data storage demands contribute significantly, with 44% of dynamic power consumption is attributed to BlockRAMs™ (BRAM) and UltraRAMs™ (URAM), emerging as the primary contributors to the overall dynamic power consumption.

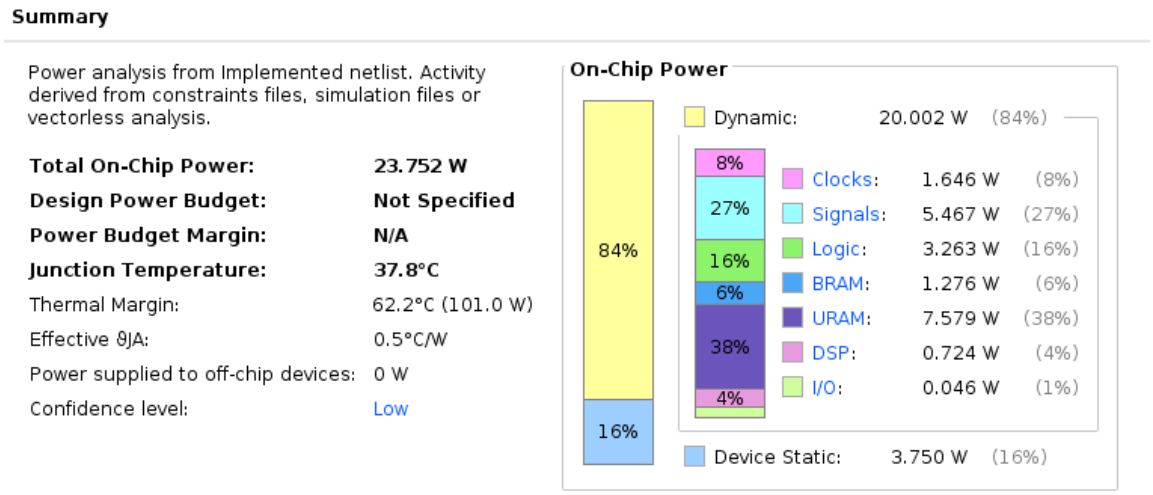


Figure 7.5: Design power consumption.

7.3 FPGA-based Beamforming Results

The correctness of our design was verified by a three-step process, which included behavioral simulation, post-synthesis simulation, and post-implementation simulation. The output data from the latter two simulations was cross-referenced with the output data from the behavioral simulation to confirm correct operation.

While the PICMUS dataset contains data from 75 different PW angles, the simulations primarily focused on a subset of three angles, namely -16, 0, and +16 degrees. This choice is primarily

attributed to the considerable time and resource demands associated with hardware simulations, especially at the post-implementation stage. It's noteworthy that the core's functionality can be effectively validated with these three angles and easily extended to cover a broader range, as the main difference lies in the number of iterations of the Compound stage-level controller. Furthermore, it is noteworthy that many studies in the field of sonoelastography have successfully employed plane waves with only three angles to track shear wave propagation, achieving clinically acceptable image quality [55].

In this evaluation, the images were generated using experimental data for the longitudinal view of the carotid artery section. The raw RF channel data frames for each PW angle were 1536-by-128 samples in size. The corresponding compounded beamformed data underwent envelope detection, max-normalization, and log compression to form compounded B-mode images with the same resolution of 1536-by-128 pixels.

The software-based reference beamformer (also based on the Temme-Mueller migration algorithm) was implemented in MATLAB® version R2021b using single-precision floating-point arithmetic. In Figure 7.6, the resulting B-mode images are displayed using the dynamic range of 60 dB. The image on the right is from the FPGA-based beamformer, while the image on the left is from the software-based reference beamformer. Upon visual examination, the two images are practically indistinguishable.

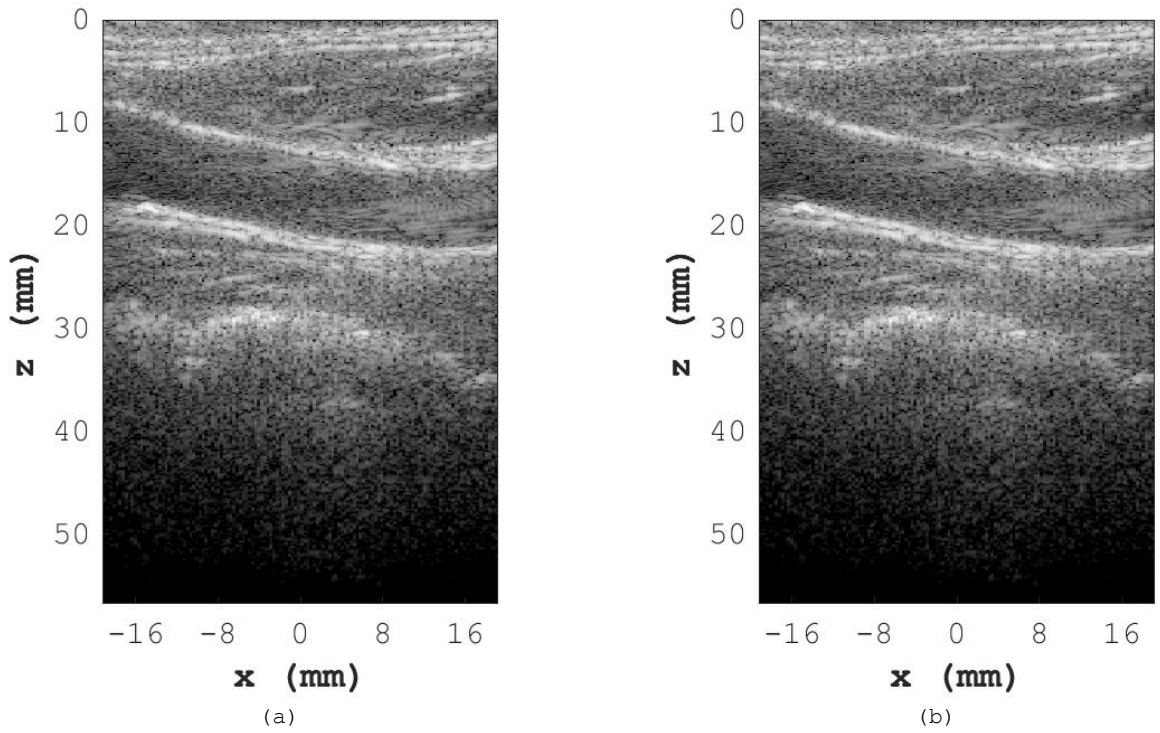


Figure 7.6: Compounded image of a carotid artery (longitudinal section): (a) Software-based (b) FPGA-based.

To compare the hardware-generated output against the reference software-generated output, we employed three key similarity metrics: mean square error (MSE), peak signal-to-noise ratio (PSNR), and structural similarity index (SSIM). These metrics were computed on four representations of the output data, and the numerical results are listed in Table 7.4. Specifically, we compared normalized Re (real) data components, normalized Im (imaginary) data components, and normalized envelopes (i.e., magnitudes of the output IQ data). In all three cases, the normalization factor was the maximum envelope value. We also compared the phases of the output IQ data.

As shown in Table 7.4, the PSNR values are high (exceeding 109 dB for amplitude and 26 dB for phase), while the MSE values are low (in comparison to the relevant data range). Furthermore, the SSIM values are either 1.000 (maximum possible) or 0.999. These numbers suggest a close match between the hardware-generated data and the software-generated data. This encouraging observation indicates that our design's limited use of fixed-point data processing in the **`DSPS`** and **`Remap`** modules did not affect the image reconstruction accuracy in any significant way.

Table 7.4: Similarity measurement results.

Output Data	Range [min, max]	SSIM	PSNR (dB)	MSE
Re (real)	[–1, 1]	1.0000	109.70	1.071E–11
Im (imaginary)	[–1, 1]	1.0000	109.69	1.073E–11
Envelope (abs.)	[0, 1]	1.0000	109.78	1.052E–11
Phase	[–π, π]	0.9986	26.56	0.0022

As it was mentioned in Chapter 1, TM migration is very similar to Lu's method. They have the same equation for remapping the f - k_x domain data to the k_z - k_x domain data except for the scaling factor (stored in an external LUT in our case). This means that the proposed architecture can be used (with minor modifications) to execute Lu's method as well.

In comparison to [42], this architecture does less work algorithmically. In [42], lateral inverse Fourier transforms are needed for every angle, whereas in Temme-Muller migration, it's only done once. This means reduced computational load and increased efficiency. The reported latency in [42] for a small test case was 270,610 cycles at a clock frequency of 80MHz (12.5ns) for 11 angles, resulting in a single-angle image reconstruction rate of approximately 3,200 fps for 1/16 frame sizes (small frames of size 256-by-16), which would translate to approximately 200 fps for full-sized frames (assuming ideal linear scaling). In other words, the architecture presented in this thesis has much higher performance. As for the FPGA resources, Table 7.5 provides a comparative summary for the case where the design from [42] was automatically synthesized using floating-point number representation. It's important to note that the target FPGA in [42] was different from ours, so one should be cautious when comparing the reported numbers directly.

Table 7.5: Summary of different FPGA implementations.

FPGA characteristics	Proposed	[42]
Clock Frequency	384 MHz	80 MHz
LUT	54714	29005
LUTRAM	12125	32
FF	105202	23787
BRAM	137.5	1063.5
URAM	896	-
DSP	238	168
IO	335	981

Chapter Eight

Conclusion

8.1 Summary

Over recent decades, high-frame-rate ultrasound imaging has gained significant attention, particularly in medical applications. Researchers have worked extensively on developing hardware accelerators, with a predominant focus on time-domain reconstruction. However, Fourier-domain reconstruction, known for its lower computational demands, remains underexplored. Existing literature shows that hardware acceleration of the Fourier-domain computations primarily relies on GPUs, effective at processing smaller frame sizes but struggling with the complexities of handling extensive data transfers associated with in larger frames. These challenges impede the real-time processing of large-sized data volumes, limiting the full potential of Fourier-domain beamforming.

This research embarked on the development of a scalable and modular single-precision floating-point FPGA accelerator for a Fourier-domain beamformer based on the Temme-Mueller migration algorithm. The primary objective of this work was to create a system capable of real-time ultrasound image reconstruction using CPWC techniques.

The proposed architecture proved to be efficient and highly accurate when compared to the software-based reference implementation. Evaluation of our FPGA design's characteristics indicated the potential for further enhancements in frame rates. This can be achieved by duplicating the main computational modules, which currently consume only 5% of available resources in a single FPGA device and achieve a frame rate exceeding 1300 frames per second.

8.2 Future Work

This research contributes to the field of ultrasound imaging by presenting an innovative FPGA-based approach that highlights the potential of Fourier-domain beamforming. The outcomes of this work hold promise for future developments, facilitating real-time applications and advancements in ultrasound image reconstruction. The completion of this research provides a foundation for several potential areas of future exploration listed below.

Enhancements to Hardware Design and Computation Duplication: Future work could focus on optimizing the hardware architecture further. This may involve exploring more advanced FPGA devices to fully harness their capabilities, fine-tuning data transfer rates, and further enhancing parallel processing capabilities. Given the substantial underutilization of FPGA logic resources (see Chapter 7), there exists substantial opportunity for increasing the reconstruction frame rate, potentially doubling it to approximately 2,600 fps, by introducing duplicated computational components. Pushing beyond duplication poses a bottleneck due to I/O utilization reaching its

limit. However, it may be possible to mitigate this constraint by moving LUT-related precomputations into the FPGA, leveraging available resources for the task.

Algorithmic Refinements and Exploring Additional Filters: Ongoing research can focus on refining the Temme- Mueller algorithm and investigating other approaches to ultrasound image reconstruction in the frequency domain. To facilitate further algorithmic improvements targeting higher-quality imaging results, Chapter 4 alluded to the concept of stage-level extensibility, allowing users to add extra filters to the architecture. This capability opens the door to enhancing the quality of the final image by exploring and integrating additional frequency-domain filtering techniques.

On-Chip Computation: In Chapter 4, it was observed that interpolation and phase shift data are calculated outside the FPGA. To reduce data transfer and enhance efficiency, such data could be generated within the FPGA depending on the application requirements.

References

[1] M. Tanter and F. Mathias, "Ultrafast Imaging in Biomedical Ultrasound," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control,* vol. 1, no. 61, pp. 102-119, 2014.

[2] T. L. Szabo, Diagnostic Ultrasound Imaging: Inside Out, New York: Academic Press, 2004.

[3] J. M. Sanches, A. F. Laine and J. S. Suri., Ultrasound Imaging, Berlin/Heidelberg: Springer, 2012.

[4] V. Shamdasani, R. Managuli, S. Sikdar and Y. Kim, "Ultrasound Color-Flow Imaging on a Programmable System," *IEEE Transactions on Information Technology in Biomedicine,* vol. 8, no. 2, pp. 191 - 199, 2004.

[5] C. Griffith and R. Nowakowski, "Powering Medical Ultrasound Imaging (Texas Instruments)," 2023. [Online]. Available: https://www.mouser.ca/applications/medical-ultrasound-power-supply/.

[6] C. M. Otto, "Principles of Echocardiographic Image Acquisition and Doppler Analysis," in *Textbook of Clinical Echocardiography, 5th Edition*, Elsevier Health Sciences, 2013.

[7] R. S. Cobbold, Foundations of Biomedical Ultrasound, Oxford University Press, 2006.

[8] A. Hadzic, Hadzic's Peripheral Nerve Blocks and Anatomy for Ultrasound-Guided Regional Anesthesia, 3rd Edition, McGraw Hill Professional, 2012.

[9] University of Minnesota, "Echocardiographic Modes," 2021. [Online]. Available: http://www.vhlab.umn.edu/atlas/echocardiography-tutorial/echocardiographic-modes.shtml. [Accessed 2023].

[10] J. Zhang, Q. He, Y. Xiao, H. Zheng, C. Wang and J. Luo, "Ultrasound Image Reconstruction from Plane Wave Radio-frequency Data by Self-Supervised Deep Neural Network," *Medical Image Analysis,* vol. 70, pp. 102018,, 2021.

[11] G. Montaldo, M. Tanter, J. Bercoff, N. Benech and M. Fink, "Coherent Plane-wave Compounding for Very High Frame Rate Ultrasonography and Transient Elastography," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency,* vol. 56, no. 3, p. 489–506, 2009.

[12] C. Zheng, H. Wang, X. Xu, H. Peng and Q. Chen, "An Adaptive Imaging Method for Ultrasound Coherent Plane-Wave Compounding Based on the Subarray Zero-Cross Factor," *Ultrasonics,* vol. 100, 2020.

[13] W. Yuanguo, C. Zheng, H. Peng and Y. Wang, "High-Quality Coherent Plane-Wave Compounding Using Enhanced Covariance-Matrix-Based Statistical Beamforming," *Applied Sciences,* vol. 12, no. 21, 2022.

[14] C. Mengjia and L. Zhenkun, "An Adaptive Imaging Method for Ultrasound Coherent Plane-Wave Compounding Based on the Polar Coherence Factor," in *6th International Conference on Intelligent Computing and Signal Processing (ICSP),* Xi'an, China, 2021.

[15] J. Quistgaard, "Signal Acquisition and Processing in Medical Diagnostic Ultrasound," *IEEE Signal Processing Magazine,* vol. 14, no. 1, pp. 67-74, 1997.

[16] M. Mozaffarzadeh, A. Mahloojifar, M. Orooji, S. Adabi and M. Nasiriavanak, "Double-Stage Delay Multiply and Sum Beamforming Algorithm: Application to Linear-Array Photoacoustic Imaging," *IEEE Transactions on Biomedical Engineering,* vol. 65, no. 1, pp. 31-42, 2017.

[17] B. Mohammadzadeh Asl and A. Mahloojifar, "Minimum Variance Beamforming Combined with Adaptive Coherence Weighting Applied to Medical Ultrasound Imaging," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control,* vol. 56, no. 9, pp. 1923 - 1931, 2009.

[18] J.-f. Synnevag, A. Austeng and S. Holm, "Benefits of Minimum-Variance Beamforming in Medical Ultrasound Imaging," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control,* vol. 56, no. 9, pp. 1868 - 1879, 2009.

[19] A. Mohades Deylami and B. Mohammadzadeh Asl, "A Fast and Robust Beamspace Adaptive Beamformer for Medical Ultrasound Imaging," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control,* vol. 64, no. 6, pp. 947-958, 2017 .

[20] T. N. Tran, W. Cowley and A. Pollok, "Automatic Adaptive Speech Separation Using Beamformer-Output-Ratio for Voice Activity Classification," *Signal Processing,* no. 113, pp. 259-272, 2015.

[21] L. Zhang, L. Huang, B. Li, M. Huang, J. Yin and W. Bao, "Fast-Moving Jamming Suppression for UAV Navigation: A Minimum Dispersion Distortionless Response Beamforming Approach," *IEEE Transactions on Vehicular Technology,* vol. 68, no. 8, pp. 7815-7827, 2019.

[22] W. Gao, X. Li and Z. Liu, "Two-Dimensional Adaptive Beamforming for Large Planar Array Antennas Based on Weight Matrix Reconstruction," *IET Radar, Sonar & Navigation,* vol. 15, no. 12, pp. 1622-1627, 2021.

[23] S. Fredrik Synnevag, A. Austeng and S. Holm, "Adaptive Beamforming Applied to Medical Ultrasound Imaging," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control,* vol. 54, no. 8, pp. 1606 - 1613, 2007 .

[24] S. M. Sakhaei, "A Decimated Minimum Variance Beamformer Applied to Ultrasound Imaging," *Ultrasonics,* vol. 59, pp. 119-127, 2015.

[25] K. Nagai, "A New Synthetic-Aperture Focusing Method for Ultrasonic," *IEEE Transactions on Sonics and Ultrasonics,* vol. 32, no. 4, pp. 531-536, 1985.

[26] H. Guo, H.-W. Xie, G.-Q. Zhou and N. Q. Nguyen, "Fourier-Domain Beamforming and Sub-Nyquist Sampling for Coherent Pixel-Based Ultrasound Imaging," in *IEEE International Ultrasonics Symposium (IUS)*, Venice, 2022.

[27] R. Cohen, Y. Sde-Chen, T. Chernyakova, C. Fraschini, J. Bercoff and Y. C. Eldar, "Fourier Domain Beamforming for Coherent Plane-wave Compounding," in *IEEE International Ultrasonics Symposium (IUS)*, Taipei, 2015.

[28] M. Albulayli and D. Rakhmatov, "Fourier Domain Depth Migration for Plane-Wave Ultrasound Imaging," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control,* vol. 65, no. 8, pp. 1321 - 1333, 2018 .

[29] Y. Zhang, Y. Guo and W.-N. Lee, "Ultrafast Ultrasound Imaging Using Combined Transmissions With Cross-Coherence-Based Reconstruction," *IEEE Transactions on Medical Imaging,* vol. 37, no. 2, pp. 337-348, 2017 .

[30] S. Chandramoorthi and A. K. Thittai, "ω-k Algorithm for Sparse-Transmit Sparse-Receive Diverging Beam Synthetic Aperture Transmit Scheme," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control,* vol. 67, no. 10, pp. 2046 - 2056, 2020.

[31] D. Garcia, L. Le Tarnec, S. Muth, E. Montagnon, J. Porée and G. Cloutier, "Stolt's f-k Migration for Plane Wave Ultrasound Imaging," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control,* vol. 60, no. 9, pp. 1853 - 1867, 2013.

[32] P. Temme and G. Muller, "Fast Plane-Wave and Single-Shot Migration by Fourier Transform," *Journal of Geophysics.,* vol. 60, pp. 19-27, 1986.

[33] R. H. Stolt, "Migration by Fourier transform," *Geophysics ,* vol. 43, no. 1, pp. 23-48, 1978.

[34] J.-y. Lu, J. Cheng and J. Wang, "High Frame Rate Imaging System for Limited Diffraction Array Beam Imaging with Square-Wave Aperture Weightings," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control,* vol. 53, no. 10, pp. 1796 - 1812, 2006.

[35] Shravanthi V. Musti, "Plane-Wave Fourier-Domain Beamforming with CNN-Assisted Resolution Enhancement," MEng Report, University of Victoria, 2022. [Online]. Available: http://hdl.handle.net/1828/13851.

[36] J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of Computation,* vol. 19, no. 90, pp. 297-301, 1965.

[37] H. Hasegawa, "Advances in Ultrasonography: Image Formation and Quality Assessment," *Journal of Medical Ultrasonics,* vol. 48, no. 4, pp. 377-389, 2021.

[38] T. Y. Phuong and J. Lee, "Design Space Exploration of SW Beamformer on GPU," *Concurrency and Computation: Practice and Experience,* vol. 27, no. 7, pp. 1718-1733, 2015.

[39] M. Walczak, M. Lewandowski and N. Żołek, "Optimization of Real-Time Ultrasound PCIe Data Streaming and OpenCL Processing for SAFT Imaging," in *IEEE International Ultrasonics Symposium (IUS)*, Prague, 2013.

[40] B. Y. Yiu, I. K. Tsang and C. H. Alfred, "GPU-Based Beamformer: Fast Realization of Plane Wave Compounding and Synthetic Aperture Imaging," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control,* vol. 58, no. 8, pp. 1698 - 1705, 2011.

[41] AMD Corporation , "Synthetic Aperture and Plane Wave Ultrasound Imaging with Versal ACAP WP520 (v1.0.1)," 3 May 2023. [Online]. Available: https://docs.xilinx.com/v/u/en-US/wp520-sa-pw-imaging.

[42] J. Shi and D. Rakhmatov, "Fixed-Point CPWC Ultrasound Image Reconstruction," in *IEEE International Ultrasonics Symposium (IUS)*, Glasgow, 2019.

[43] H. V. Sorensen and C. Burrus, "Fast DFT and Convolution Algorithms," in *Handbook for Digital Signal Processing*, S. Mitra and J. Kaiser, Eds. NY: Wiley, 1993, pp. 491-610, ch. 8.

[44] L. Marple, "Computing the Discrete-Time "Analytic" Signal via FFT," *IEEE Transactions on Signal Processing*, vol. 47, no. 9, pp. 2600-2603, 1999.

[45] AMD Corporation, "Fast Fourier Transform v9.1 LogiCORE IP Product Guide (PG109)," 4 May 2022. [Online]. Available: https://docs.xilinx.com/r/en-US/pg109-xfft/Fast-Fourier-Transform-v9.1-LogiCORE-IP-Product-Guide.

[46] AMD Corporation, "DDS Compiler v6.0 LogiCORE IP Product Guide (PG141)," 21 January 2021. [Online]. Available: https://docs.xilinx.com/v/u/en-US/pg141-dds-compiler.

[47] AMD Corporation, "Floating-Point Operator v7.1 LogiCORE IP Product Guide (PG060)," 16 December 2020. [Online]. Available: https://docs.xilinx.com/v/u/en-US/pg060-floating-point.

[48] AMD Corporation, "Block Memory Generator v8.4 LogiCORE IP Product Guide (PG058)," 6 August 2021. [Online]. Available: https://docs.xilinx.com/v/u/en-US/pg058-blk-mem-gen.

[49] AMD Corporation, "Versal ACAP Memory Resources Architecture Manual (AM007)," 24 November 2020. [Online]. Available: https://docs.xilinx.com/r/en-US/am007-versal-memory/Block-RAM-Summary.

[50] AMD Corporation, "UltraScale Architecture DSP Slice User Guide (UG579)," 30 August 2021. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug579-ultrascale-dsp.

[51] AMD Corporation, "Vivado Design Suite: AXI Reference Guide (UG1037)," 15 July 2017. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug1037-vivado-axi-reference-guide.

[52] AMD Corporation, "UltraScale Architecture-Based FPGAs Memory IP Product Guide (PG150)," 20 April 2022. [Online]. Available: https://docs.xilinx.com/v/u/en-US/pg150-ultrascale-memory-ip.

[53] H. Liebgott, A. Rodriguez-Molares, F. Cervenansky, J. A. Jensen and O. Bernard, "Plane-Wave Imaging Challenge in Medical Ultrasound," *IEEE International Ultrasonics Symposium (IUS)*, pp. 1-4, 2016.

[54] AMD Corporation, "UltraScale+ FPGAs Product Selection Guide (XMP103)," 20 June 2023. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ultrascale-plus-fpga-product-selection-guide.

[55] B. Du, X. Wu, H. Zheng, S. Fang, M. Lu and R. Mao, "Coherence Plane-Wave Compounding with Angle Coherence Factor for Ultrafast Ultrasound Imaging," in *40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, Honolulu, 2018.