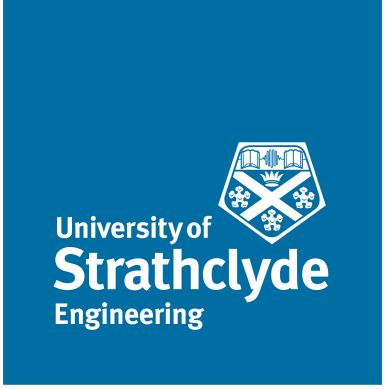

Run-Time Reconfigurable Systems and Algorithms for RFSoC-Based Software Defined Radio Applications

PhD Thesis

Joshua Goldsmith

Department of Electronic and Electrical Engineering

University of Strathclyde, Glasgow

February 20, 2026

This thesis is the result of the author's original research. It has been composed by the author and has not been previously submitted for examination which has led to the award of a degree.

Signed: **Joshua Goldsmith**

Date: **February 20, 2026**

Abstract

The growing demand for wireless connectivity has increased the shortage of usable Radio Frequency (RF) spectrum, resulting in congested unlicensed bands and underutilised licensed spectrum. Dynamic Spectrum Access (DSA), enabled by Software Defined Radio (SDR) and Cognitive Radio (CR), shows promise in improving spectrum utilisation by allowing opportunistic and adaptable usage of available bands. However, traditional SDR platforms often suffer from complexity, limited reconfigurability, and non-deterministic behaviour due to disparate hardware technologies.

This thesis addresses these challenges by developing practical methodologies and efficient algorithms for RF-sampling FPGA-based SDR devices, particularly AMD's Zynq UltraScale+ RF System-on-Chip (RFSoC).

First, a novel design methodology for SDR applications on the RFSoC is presented and used to develop a fully functional transceiver demonstrator, incorporating real-time data inspection and visualisation. The demonstrator provides user-driven software reconfiguration using Model Composer-generated HDL IP cores and integrated within the PYNQ framework. Experimental results demonstrate effective real-time visualisation at rates up to 20 FPS, with notably low FPGA resource utilisation. This work is shown to have had a positive impact on the community, and beyond, serving as a foundation for subsequent research.

Second, this research introduces SPECTRE, an open-source, Python-based frequency planning tool designed to dynamically identify spur-free configurations for RFSoCs. Frequency planning is defined as a search space problem and, through analysis, key interference spurs were identified, and various search algorithms evaluated. Results

highlight that a hybrid search strategy combining coarse grid and exhaustive random searches could achieve real-time performance on the RFSoC.

Finally, a novel FPGA-based fixed-point filter design method is developed for run-time reconfigurable FIR filters. Results show excellent filter performance (up to 88 dB stopband attenuation) and ultra-low deterministic latency (2.52 µs), suitable for real-time SDR applications.

All tools developed in this thesis were publicly released as open-source, significantly contributing to practical and accessible SDR and CR applications.

Acknowledgements

Firstly, I would like to thank my supervisors, Dr. Louise Crockett and Prof. Robert Stewart, for their continuous guidance and patience. Their steadfast support provided me with countless opportunities that shaped my career in ways I could never have imagined.

I am also deeply grateful to my fellow StrathSDR members, who have been an incredible source of inspiration, motivation, and much-needed comic relief throughout. A special mention goes to Craig Ramsay—without whom I would never have survived Colorado—as well as Douglas Allan and David Northcote, whose technical insights were invaluable.

I would also like to acknowledge my family and friends whose encouragement, patience, and belief in me were instrumental in bringing this thesis to completion.

Additional thanks go to the AMD University Program and the PYNQ team—of which I am now proud to be a member—for providing the resources and tools that made this research possible. In particular, I would like to thank Cathal McCabe for RFSoC support and Graham Schelle for his management of the PYNQ project.

Finally, I would like to give a special nod to the two brave RFSoCs that sacrificed themselves in the pursuit of knowledge. Your incendiary departures were not in vain.

Contents

List of Figures

List of Tables

List of Acronyms

ADC Analogue to Digital Converter

AMD Advanced Micro Devices

API Application Programming Interface

AXI Advanced eXtensible Interface

BOM Bill of Materials

BRAM Block Random Access Memory

CFFI C Foreign Function Interface

CIC Cascaded Integrator-Comb

CMA Contiguous Memory Array

CP Control Point

CPU Central Processing Unit

CR Cognitive Radio

DAC Digital to Analogue Converter

DC Direct Current

DDC Digital Down Converter

DDS Direct Digital Synthesis

DFT Discrete Fourier Transform

DIM Data Inspection Module

DMA Direct Memory Access

DRAM Dynamic Random Access Memory

DSA Dynamic Spectrum Access

DSP Digital Signal Processing

DUC Digital Up Converter

FFT Fast Fourier Transform

FIFO First In, First Out

FIR Finite Impulse Response

FPGA Field Programmable Gate Array

FPS Frames Per Second

FSM Finite State Machine

GPU Graphics Processing Unit

GUI Graphical User Interface

HDL Hardware Description Language

HLS High-Level Synthesis

HP High Performance

I2C Inter-Integrated Circuit

IDE Integrated Development Environment

IDFT/Inverse DFT Inverse Discrete Fourier Transform

IF Intermediate Frequency

IFFT Inverse Fast Fourier Transform

IIR Infinite Impulse Response

IP Intellectual Property

IPI IP Integrator

ISM Industrial, Scientific, and Medical

JSON JavaScript Object Notation

LFSR Linear Feedback Shift Register

LO Local Oscillator

LUT Lookup Table

MAC Medium Access Control

MMIO Memory-Mapped Input/Output

MNO Mobile Network Operator

NCO Numerically Controlled Oscillator

OFDM Orthogonal Frequency-Division Multiplexing

OP Observation Point

OS Operating System

OSI Open Systems Interconnection

PL Programmable Logic

PLL Phase-Locked Loop

PS Processing System

PU Primary User

QoS Quality of Service

QPSK Quadrature Phase-Shift Keying

RF Radio Frequency

RF-DC RF Data Converter

RFIR Reconfigurable FIR

RFSoC Radio Frequency System-on-Chip

RRC Root-Raised-Cosine

RTL Register Transfer Level

SD-FEC Soft-Decision Forward Error Correction

SDR Software Defined Radio

SFDR Spurious-Free Dynamic Range

SNR Signal-to-Noise Ratio

SoC System-on-Chip

SPECTRE Spur Planning, Evaluation, and Configuration Tool for RF-sampling dEvices

SU Secondary User

TDMA Time Division Multiple Access

UK United Kingdom

UML Unified Modelling Language

USRP Universal Software Radio Peripheral

VHDL Very High Speed Integrated Circuit Hardware Description Language

Chapter 1

Introduction

1.1 Spectrum Scarcity and Under-Utilisation

A commonly cited issue within the wireless communications field is the apparent lack of available Radio Frequency (RF) spectrum in which to broadcast. This is commonly described as the "spectrum drought" [8], or, perhaps more dramatically, the "spectrum crisis" [9].

RF spectrum in many countries is tightly regulated by governmental institutions, such as the Office of Communications (Ofcom) in the UK. These institutions license large portions of the spectrum for commercial and governmental use-cases, while providing smaller unlicensed bands for more general use; such as WiFi, Bluetooth, and other short-range wireless communication applications. As a consequence, the unlicensed bands have become congested, while many parts of the licensed spectrum are under-utilised by the licensees, particularly in low-population areas [10, 11].

This suggests that the challenge facing modern wireless systems is not the scarcity of spectrum itself, but rather the inefficient use of existing spectrum resources.

1.2 Dynamic Spectrum Access, Cognitive Radio, and Software Defined Radio

One such model to combat spectrum under-utilisation is Dynamic Spectrum Access (DSA) in which licensed, Primary Users (PUs) retain preferential access to a specific band, while non-licensed, Secondary Users (SUs) are given opportunistic access when it is not in use and would not cause interference with the license holder. This method of spectrum sharing provides a means of utilising the available licensed spectrum more efficiently [12].

Adaptability is a key aspect of DSA, as the availability of a licensed frequency band may change over time, depending on PU activity. This requires SUs to actively observe the spectral environment and adapt parameters accordingly [13, 14]. As such, DSA requires both the ability to reconfigure radio hardware and to make decisions based on the observed spectral environment. These abilities are provided by Software Defined Radio (SDR) and Cognitive Radio (CR), respectively [15].

The SDR concept aims to transform much of the traditionally static physical (PHY) layer of the radio with programmable signal-processing components that can be reconfigured in software. By implementing common radio features—such as modulation, filtering, and mixing—in reprogrammable logic or software-controlled hardware, a single SDR platform can support multiple frequency bands, standards, and waveforms on the same physical device [16, 17].

While SDR provides the underlying reconfigurable hardware capability, CR brings intelligence to the wider radio network. CR provides awareness and decision-making that enables the SDR to observe the spectral environment, learn from past behaviour, and adapt its transmission parameters to avoid interference with other users—this process of observation, analysis, decision, and action is known as the *cognitive cycle* [15, 18].

It is important to note that current implementations of DSA and CR are typically limited to specific bands and regulatory frameworks, such as TV white space [19], and do not assume unrestricted access across the RF spectrum. However, there is a clear trend towards more flexible and dynamic spectrum regulation, driven by persistent

under-utilisation and increasing demand for wireless services [20]. As regulatory bodies continue to relax constraints, future DSA networks will likely operate over a much wider portion of the spectrum, requiring SDRs to handle wider bandwidths and a variety of frequency bands. Additionally, as the number of PUs and SUs increases, the overall spectrum will likely become more dynamic. This will require the cognitive cycle to be performed faster and more frequently, placing requirements on the SDR to reconfigure its parameters in a timely and predictable manner. This motivates the need to consider real-time operation and deterministic behaviour at both a hardware and software level [21, 22].

1.3 Real-Time Operation and Determinism

1.3.1 Defining Real-time and Determinism

In general terms, a real-time system is one that responds to external events or stimuli within a defined time constraint [23] and, therefore, is highly application dependent. In the context of CR, spectrum sensing and subsequent reconfiguration must occur quickly enough to prevent interference with PUs, while maintaining Quality of Service (QoS) [18]. Therefore, within this thesis, real-time operation implies that the system—comprising signal-processing algorithms, control logic, and reconfiguration mechanisms—can respond to input stimuli within a latency budget that maintains uninterrupted operation of the system. Furthermore, as this total latency is the sum of the latencies of the individual processing and control components, each component must be designed to operate with minimal latency in order for the overall system to meet real-time performance [24].

Determinism refers to the property of a system in which its behaviour is uniquely defined given a set of inputs and initial conditions. A deterministic system guarantees that identical inputs lead to the same outputs and that the timing of these outputs is predictable within defined limits [25, 26]. In the context of SDR, determinism includes both functional and temporal aspects. Functional determinism requires that identical input signals produce the same output signals, whereas temporal determinism

requires that the timing of processes are constrained, ensuring predictable latency across multiple executions [26, 27]. This is particularly important in DSA systems, where non-deterministic behaviour, such as variable latency, can result in missed transmission opportunities, increased interference, and degraded QoS [14].

1.3.2 Defining the Reconfiguration Time of a Cognitive Radio

Having defined both real-time and determinism in the context of CR, it is also important to define the specific time constraints that apply to the cognitive cycle. In particular, it is the *act* phase which is the focus of this thesis, where the parameters of the SDR are reconfigured based on the observations and decisions of the previous phases [16, 28].

However, what remains notably absent in both the standards and literature is any comparable set of latency targets for this reconfiguration process, stating only that the *act* phase must be performed in real-time [14, 18, 29].

Where numeric values are provided, they relate primarily to the *sensing* part of the cognitive cycle. For example, in IEEE 802.22, spectrum sensing is specified across three distinct levels, each corresponding to a different abstraction layer and timescale. At the regulatory level, the standard defines a channel detection time of less than 2 s, which represents an upper bound on the total time allowed to reliably detect the appearance of a PU. At the MAC level, sensing is enabled through scheduled quiet periods, during which transmissions are suspended for durations ranging from a single Orthogonal Frequency-Division Multiplexing (OFDM) symbol of 0.33 ms, to one superframe of 160 ms. At the PHY level, sensing is divided into fast sensing of around 1 ms, and fine sensing up to 25 ms. This reflects the processing time required for coarse and detailed signal detection, respectively [19, 30–32].

Other standards that incorporate sensing, such as LTE Licensed Assisted Access and 5G NR-U, specify only microsecond-scale *listen-before-talk* intervals, without defining explicit sensing windows [33, 34].

Given these observations and the fact that the reconfiguration of the radio will primarily be performed on the radio's PHY layer, it is reasonable to infer that the *act* phase of the cognitive cycle should occur within a time-frame similar to the fast

and fine sensing of the IEEE 802.22 standard. Therefore, this thesis defines real-time reconfiguration as the ability to reconfigure radio parameters within 1 to 25 ms, depending on the application.

1.4 Limitations of Traditional SDR Platforms

A wide range of SDR platforms have been developed over the years to support CR research. In general, these platforms are composed of heterogeneous components, including Intermediate Frequency (IF)-sampling data converters, RF front-ends, Field Programmable Gate Arrays (FPGAs), Graphics Processing Units (GPUs), and embedded or general-purpose processors—often distributed across multiple devices [1, 5]. This modular approach enables significant flexibility, but also results in systems that are fragmented across both hardware and software domains.

While these SDR platforms and CR testbeds have shown promising results, they are often limited in scope, supporting only a small number of frequency bands, bandwidths, and protocols [1, 35, 36]. As a result, they are ill-suited for DSA networks that aim to exploit more of the available RF spectrum. Additionally, limited support for concurrent channels, antennas, and receiver chains, constrain requirements, such as spectrum sensing, reducing their effectiveness within the cognitive cycle [3].

The modular nature of these platforms often introduces multiple sources of latency and non-determinism, due to data and control signals passing through multiple interconnects—connecting RF front-ends, FPGAs, and processors. Furthermore, while FPGAs are well-suited to implementing latency-sensitive algorithms, limited resources in many of these traditional SDR platforms restrict the extent to which these algorithms can be offloaded [37]. Consequently, these platforms can struggle to provide the deterministic and low-latency performance required by CR [37–39].

Beyond performance considerations, these SDR platforms also present practical challenges for implementation of CR systems. Experiments and testbeds are often complex to configure, highly platform-specific, and difficult to reproduce across testbeds [39].

While these traditional SDR platforms have been important in advancing CR research, the fragmented architectures, limited scalability, variable latency, and design complexity,

have made it challenging to satisfy the requirements of DSA. These shortcomings motivate the investigation of more integrated radio architectures, such as direct-RF sampling System-on-Chips (SoCs), which offer the potential to address many of the challenges highlighted above.

1.5 RF-Sampling SoCs as an Enabling Technology

Recent advances in semiconductor technology have led to the development of RF-sampling SoC devices, which provide an alternative approach to SDRs. Unlike traditional SDR platforms, where RF front-ends, data converters, programmable logic, and control processors are implemented as separate components across multiple devices, RF-sampling SoCs integrate these elements on a single chip [40–42]. By reducing the need for external connections between major parts of the radio system, this approach mitigates several of the architectural limitations of traditional SDR platforms in CR/DSA systems [42, 43].

A key feature of RF-sampling SoCs is their support for direct-RF sampling, in which received signals are digitised as close to the antenna as possible. This allows many traditional analogue components, such as mixers, Local Oscillators (LOs), and IF stages, to be reduced or removed entirely [42, 44, 45]. This greatly simplifies the receiver architecture and allows more of the radio functionality to be reconfigured through software. Additionally, these devices also support wider bandwidths, higher channel counts, and more flexible multi-band operation, improving scalability [45, 46]. This makes these RF-sampling SoCs well suited to CR and DSA systems that operate across a wide range of frequencies and bandwidths [40, 45].

The integration of RF data converters, Programmable Logic (PL), and processors within RF-sampling SoCs also has important implications for both latency and deterministic behaviour. Because signal processing and control functions are implemented on the same device, data and control signals do not need to cross external links, such as USB, PCIe, or JESD204 [41, 46]. This reduces the delays introduced by these interfaces and the variability associated with host-based control [41].

These single-device RF-sampling SoCs offer the potential for faster and more deterministic reconfiguration than traditional SDR platforms, with the AMD Zynq Ul-

traScale+ (hereafter Radio Frequency System-on-Chip (RFSoC)) representing one of the most mature and widely available of these devices [46]. However, this increased level of hardware integration moves much of the remaining complexity into the software domain. In particular, issues related to design complexity, run-time frequency planning, and spur suppression remain, and motivate the contributions presented in this thesis.

1.6 Motivation and Scope

1.6.1 RFSoC Development Complexity and the Need for a Design Methodology

Despite its capabilities, RFSoC development remains complex and time-consuming, requiring expertise across FPGA design, embedded software, and RF systems [47, 48]. The tight coupling between these three domains means that even small design changes can require modifications across the entire stack. Additionally, typical development flows primarily target static designs and are not well suited to rapid prototyping and iterative experimentation, limiting research productivity [48–50].

Therefore, while RFSoC platforms provide the performance and flexibility required by CR systems, the traditional development flows can limit accessibility for users—motivating the need for a more productivity-focused design methodology.

1.6.2 Frequency Planning as a Requirement for Cognitive Radio

Direct-RF sampling devices, such as the RFSoC, introduce deterministic spurious signals (spurs) as a consequence of the data conversion process. If these spurs fall within the signal band of interest they can significantly degrade receiver performance [48, 51]. Frequency planning is commonly used to avoid this issue by selecting sample rates, carrier frequencies, and clock frequencies, such that these spurs are shifted outside the band of interest and can be suppressed by filtering [43, 48, 51].

In traditional architectures, radio parameters are typically fixed, and frequency planning is therefore performed once during the design stage. In CR systems, these parameters are instead adjusted dynamically in response to spectrum availability and PU

activity. Changes in centre frequency, bandwidth, or sampling rate alter the locations of spurs, meaning that a frequency plan computed in advance may no longer be valid once the radio reconfigures [43, 48].

Therefore, frequency planning in CR systems must be performed at run-time with low latency, rather than offline during the design stage. However, existing frequency-planning tools are primarily intended for static system configuration and are therefore unsuitable for CR systems that require frequent and time-constrained reconfiguration [52–54].

1.6.3 Reconfigurable FIR Filters for Spur Suppression

After frequency planning has shifted the spurs outside the band of interest, they must then be suppressed through filtering [43, 48]. These filters are designed to attenuate specific out-of-band spurs, whose frequency locations depend on the centre frequency, sampling rate, and clock configuration of the SDR.

In CR systems, these parameters may change frequently and unpredictably, causing the frequency locations of the spurs to change in a similar manner. This requires the associated filter parameters to be updated at run-time to ensure effective spur suppression [55]. Fixed filters—or small sets of pre-designed coefficients—are therefore unsuitable in this context as the number of possible operating configurations can be very large, due to the wide range of possible centre frequencies, bandwidths, and sampling rates.

Moreover, generating filter coefficients on the processing system, then loading them into FPGA-based filters at run-time, can introduce additional latency and non-deterministic behaviour, which conflict with the real-time requirements of CR systems [38, 39, 55]. Since the filters themselves reside on the FPGA, it is therefore worth considering computing the filter coefficients directly on the FPGA at run-time instead.

1.7 Research Aims and Contributions

This thesis addresses the challenges identified in Section 1.6 and makes the following original contributions to the design and implementation of CR systems based on RF-sampling SoCs.

First, this thesis proposes a design methodology for developing systems on the RFSoC platform. The methodology addresses the complexity arising from the tight integration of RF data converters, FPGA logic, and embedded processors on the RFSoC by providing a development approach that supports rapid prototyping and iterative experimentation. This contribution directly addresses the development complexity and productivity limitations identified in Section 1.6.1 and is presented in Chapter 3.

Second, a scriptable frequency planning tool is presented, enabling the real-time identification of spur-free operating configurations for RFSoC-based CRs. This work establishes frequency planning as a run-time requirement for RFSoC-based CR systems, addressing the limitations identified in Section 1.6.2, and is presented in Chapter 4.

Finally, this thesis introduces a natively fixed-point, run-time reconfigurable Finite Impulse Response (FIR) filter design method for FPGA hardware. The proposed method enables deterministic generation of filter coefficients on the FPGA at run-time, avoiding the latency and non-determinism associated with off-chip coefficient generation and filter reconfiguration. This contribution directly addresses the need to compute reconfigurable FIR filter coefficients on the FPGA identified in Section 1.6.3 and is presented in Chapter 5.

1.8 List of Publications

This section provides a list of publications resulting from the work presented in this thesis. Publications 1, 2, and 3 are specific to the contributions discussed in Chapters 3, 4, and 5, respectively. Publications 4, 5, and 6 are book contributions that make up much of the background information presented in Chapter 2. Publication 7 provides an additional work on wideband spectrum sensing for RFSoC.

1. J. Goldsmith, C. Ramsay, D. Northcote, K. W. Barlee, L. H. Crockett and R. W. Stewart, "Control and Visualisation of a Software Defined Radio System on the Xilinx RFSoC Platform Using the PYNQ Framework," *IEEE Access*, vol. 8, pp. 129012-129031, Jul. 2020, doi: 10.1109/ACCESS.2020.3008954.

2. J. Goldsmith, "RF Data Converters: Figures of Merit and Frequency Planning" in *Software Defined Radio with Zynq UltraScale+ RFSoC*, L. H. Crockett, D. Northcote, and R. Stewart, Eds. Strathclyde Academic Media; Jan. 2023, pp-383-409, ISBN: 9780992978792.

3. J. Goldsmith, L. H. Crockett and R. W. Stewart, "A Natively Fixed-Point Run-Time Reconfigurable FIR Filter Design Method for FPGA Hardware," in *IEEE Open Journal of Circuits and Systems*, vol. 3, pp. 25-37, Feb. 2022, doi: 10.1109/O-JCAS.2022.3152399.

4. J. Goldsmith, "RF Data Converters: Digital to Analogue" in *Software Defined Radio with Zynq UltraScale+ RFSoC*, L. H. Crockett, D. Northcote, and R. Stewart, Eds. Strathclyde Academic Media; Jan. 2023, pp-355-382, ISBN: 9780992978792.

5. J. Goldsmith, L. Brown, M. Šiaučiulis, and G. Fitzpatrick, "Design Tools and Workflows" in *Software Defined Radio with Zynq UltraScale+ RFSoC*, L. H. Crockett, D. Northcote, and R. Stewart, Eds. Strathclyde Academic Media; Jan. 2023, pp-415-455, ISBN: 9780992978792.

6. J. Goldsmith and C. Ramsay, "Software Stacks" in *Exploring Zynq MPSoC: With PYNQ and Machine Learning Applications*, L. H. Crockett, D. Northcote, C. Ramsay, F. Robinson, R. W. Stewart. Strathclyde Academic Media; Apr. 2019, pp-301-329, ISBN: 9780992978754

7. M. Šiaučiulis, D. Northcote, J. Goldsmith, L. H. Crockett, and Š, Kaladė, "100GBit/s RF sample offload for RFSoC using GNU Radio and PYNQ," *2023 21st IEEE Interregional New Circuit and Systems (NEWCAS)*, Edinburgh United Kingdom, 2023, pp. 1-5, doi: 10.1109/NEWCAS57931.2023.10198070

1.9 Thesis Structure

The remainder of this thesis is organised as follows:

- **Chapter 2** provides a background and literature review for the concepts discussed throughout this thesis. It explores the evolution of SDR as a foundation for CR, positioning the RFSoC as a key enabler. By reviewing current academic and industrial practices, the chapter identifies specific gaps in design productivity, real-time frequency planning, and deterministic hardware reconfiguration, establishing the motivation for the subsequent technical contributions.

- **Chapter 3** presents an SDR design methodology for the RFSoC, in which a reconfigurable radio transceiver demonstrator is developed, with built-in software signal inspection and visualisation. This is facilitated by the use of Model Composer for FPGA design and the extension of the PYNQ framework to support the RFSoC.

- **Chapter 4** looks at reconfigurability within the context of real-time frequency planning. A novel, open-source frequency planning tool is presented, capable of calculating device configurations that result in spur-free operation of the RFSoC. This tool is then used to evaluate real-time frequency planning, which can be used to reconfigure the RFSoC at run-time.

- **Chapter 5** continues the concept of reconfigurability within SDR applications by presenting a novel FPGA-based filter design method, operating entirely in fixed-point arithmetic, for use with reconfigurable FIR filters. A number of common FIR filter design techniques are evaluated to determine their suitability for use within FPGAs, providing the basis for a technique which operates entirely in fixed-point arithmetic.

- **Chapter 6** provides a résumé of the body of work presented within this thesis, along with a summary of key results, and future work that could be undertaken to extend the outcomes of this research.

Chapter 2

Background and Literature Review

2.1 SDR for Cognitive Radio

2.1.1 Cognitive Radio

The demand for radio spectrum is increasing, while at the same time much of the radio spectrum is owned by licensed users who underutilise it. This leaves unlicensed users with much smaller sections of available spectrum in the unlicensed bands, resulting in these bands becoming congested [56]. For example, it was found in [57] that the MNOs hold approximately 30% of all spectrum below 3.8 GHz in the United Kingdom (UK), whereas the unlicensed Industrial, Scientific, and Medical (ISM) bands make up just 5% of the spectrum within the same frequency range [58]. Due to the growth in devices requiring use of the spectrum, and with the apparent lack of use of large parts of the licensed bands, a number of solutions have been proposed to address these issues, with the most prominent being DSA.

As discussed in Chapter 1, one form of DSA divides users into two categories, the PU, which is the license holder; and the SU, which is the unlicensed user. In this DSA model, a PU has exclusive rights to the licensed band, and is permitted to use it uninterrupted, while the SU attempts to use the licensed band in an opportunistic manner when the PU is not using the spectrum [12, 20]. CR is a key enabler of this type

of spectrum management, providing intelligence to the radio network, with the ability to adapt and change radio parameters based on the current state of the spectrum. It is able to use spectrum sensing to detect unused gaps in a band, known as "spectrum holes", and allocate these to a SU.

One key factor in CR is spectrum sensing, where the ability to monitor the spectrum at a particular time and location is required to determine if any spectrum holes are available for a SU. Another key factor is the ability for a CR to use the spectrum intelligently. For example, if a SU only requires a low-throughput connection then a smaller bandwidth can be used, or if there is significant interference in the area then a more robust modulation scheme may be required. Because of these factors, CR presents a number of technical hurdles to overcome in order to make this technology feasible. These include the ability to detect when a PU is using, or wants to use, a specific band; communication between a transmitter and receiver, where the perceived spectrum holes may differ between them; and the requirement for radio parameters to be updated with low-latency, to avoid any collisions with PUs [12, 59].

The operational process of a CR is typically discussed in terms of the *cognitive cycle*, which consists of five stages: spectrum sensing, spectrum decision, spectrum mobility, and spectrum sharing; as shown in Figure 2.1 [20, 50].

In the cognitive cycle, the CR scans and captures the current state of the radio spectrum and identifies available spectrum holes. The CR then identifies the best spectrum hole to use for the SU, based on that user's requirements. If there is a change in the spectrum, for example if a PU is detected or there is a degradation of the channel, then the CR will move the SU to another available spectrum hole. Finally, the CR coordinates with other nodes on the network to avoid collisions.

To enable this cognitive cycle efficiently a CR must be able to operate in real time, have low and deterministic latency [37, 60, 61]. To maintain QoS of the network a SU must not interfere with a PU. Therefore, the spectrum sensing techniques used must be able to quickly detect PUs and update radio parameters accordingly. Moreover, a deterministic approach must be taken to ensure the CR is predictable and consistent.

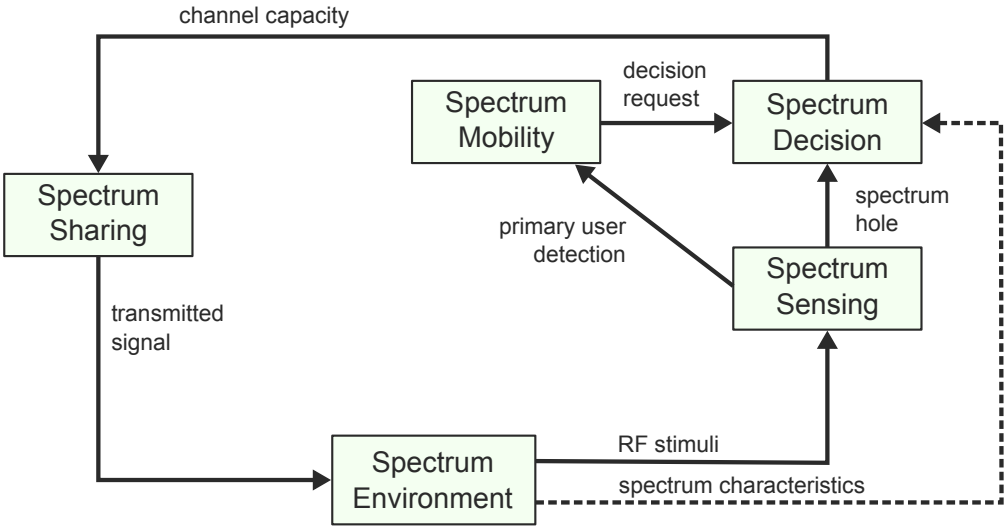

Figure 2.1: The cognitive cycle of a CR.

Factors such as latency in a system must be consistent to ensure that when a PU is detected, the SU can take action in a predictable time frame [12, 62].

2.1.2 Software Defined Radio

A Software Defined Radio can be simply described as a digital radio device in which some, or all, of its physical layer components are defined in software [63, 64]. This may include the frequency band it operates in, modulation scheme, wireless protocol, or standard. The physical (PHY) layer mentioned here is in reference to the lowest layer of the Open Systems Interconnection (OSI) networking model, in which the PHY is responsible for the transmission and reception of the raw bits from the Medium Access Control (MAC) layer above. Within the context of a digital radio architecture, the PHY layer is responsible for the baseband and RF processing [65].

The concept of SDR was conceived in the early 1990s within the military industry, where the original goal was to enable interoperability between existing and legacy radios, and to enable any future communications protocols to be implemented on hardware that was already deployed in the field [66]. The SDR concept was well-received and instigated a steady flow of academic research and early commercial products, with proponents

heralding SDR as the future of radio communications. Even as late as 2011 SDR was predicted to become the standard technology for both military and commercial radios by 2020 [67]. While this lofty goal is still to be realised, SDR has continued to evolve, with most modern radio architectures implementing the SDR concept in some form.

Due to the original goal of SDR being interoperability, the primary technology it was targeted at was general-purpose processors, where a general-purpose programming language, such as C or C++, would be used to implement all components of the radio architecture. However, due to high operational power, and difficulty using low-level features and operations at the baremetal level, it was quickly found that these devices were unsuitable for the task [66]. Since then, SDRs have primarily been targeted at a combination of embedded processor and FPGA, with the RF facilitated by separate front-end components [1, 5]. This combination of components helped to reduce power consumption due to the relatively low power pull of both FPGAs and embedded processors [68]. A further advancement was made with the release of FPGA-based SoCs, such as the Zynq-7000 SoC from AMD. These SoCs combine the FPGA and embedded processor on a single device, removing the external communication link between them, helping to improve latency [69].

2.1.3 Limitations of Legacy SDR Platforms for Cognitive Radio

A wide variety of SDRs are commercially available, including those based on general-purpose processors, GPUs, FPGAs, SoCs or Digital Signal Processing (DSP) processors [1, 37, 70, 71]. However, most literature on CR implementation has been focused on the use of FPGA or SoC-based SDRs—primarily the Universal Software Radio Peripheral (USRP) series from Ettus Research [1, 71]. These platforms are typically connected to a host computer via USB or Ethernet, where the radio processing is performed on the host computer, and the RF is handled by the SDR front-end.

These platforms have been used to demonstrate various parts of the cognitive cycle, such as spectrum sensing [4], decision making [3], and reconfiguration [35]. However, while these platforms have shown some success in demonstrating parts of CR functionality, they have been found to be lacking in the performance and capabilities required for

a complete real-time CR implementation. For example in [4], three SDRs, including two USRPs, were compared for their suitability in spectrum sensing applications. While each platform was able to perform spectrum sensing, the instantaneous bandwidth was limited by the host interface, such as USB or Ethernet. This resulted in wider frequency bands having to be stitched together from multiple captures, causing the system to miss short-time transmission bursts and resulting in non-coherent data. In [3], a resource allocation algorithm was implemented on USRP devices. However, the SDRs were found to be limited to a single channel and Time Division Multiple Access (TDMA) operation, and observed considerable latency on the host interface. In [35], reconfiguration of the radio parameters was performed using a USRP. However, the authors found throughput and latency to be limited by the host interface. Additional examples exist in the literature, demonstrating legacy SDRs as insufficient for complete implementations of CR applications [2, 5, 37, 72, 73]. Here, legacy SDRs are defined as any SDR platform which is non-RF-sampling.

One of the major limitations of legacy SDRs seen in the literature is the physical separation between the radio hardware and the processing system. In these systems, raw data must pass between a typically non-deterministic interface, such as USB or Ethernet, to then be processed by the host computer. This limits bandwidth and introduces significant round-trip latency and jitter [74]. These factors reduce the time available for spectrum analysis and decision-making, which can lead to missed opportunities for spectrum access, or collisions with PUs. Moreover, the limited bandwidth of these platforms means that they have to capture the spectrum piecemeal, which can lead to non-coherent data [4]. For example, the 802.22 band is approximately 800 MHz wide, which is far beyond the capabilities of the legacy SDRs represented in the literature [32].

Table 2.1 provides a summary of the specifications of commercially available SDRs used in CR research, showing the limitations in bandwidth and connectivity that are common across these platforms. Of these devices, only the USRP E310 is standalone, meaning it does not require a host computer to operate. However, this device is limited by its bandwidth, only supporting up to 56 MHz.

Device Name	Frequency Range	Bandwidth	Connectivity	Standalone
HackRF One	1 MHz–6 GHz	20 MHz	USB 2.0	No
BladeRF 2.0 micro	47 MHz–6 GHz	61.44 MHz	USB 3.0	No
LimeSDR USB	100 kHz–3.8 GHz	61.44 MHz	USB 3.0	No
Pluto-SDR	325 MHz–3.8 GHz	20 MHz	USB 2.0	No
USRP B200/B210	70 MHz–6 GHz	56 MHz	USB 3.0	No
USRP N210	DC–6 GHz	25 MHz	Ethernet	No
USRP X310	DC–6 GHz	160 MHz	10 GbE/PCIe	No
USRP E310	70 MHz–6 GHz	56 MHz	Ethernet/USB	Yes

Table 2.1: Representative summary of commercially available legacy SDRs used in CR research [1–5].

Therefore, there is a need for a more suitable SDR platform that can meet the requirements of real-time CR applications, such as higher bandwidths, lower latency, and better connectivity between the radio and the processing system.

2.1.4 RF-Sampling SDRs

The degree to which a digital radio is software defined depends on how much of its functionality is performed in the digital domain and, therefore, the point in the signal path in which the data converters are situated. The more functionality that is digitised, the more parts of the radio that can be controlled using software, with the *ideal* SDR being one in which all functionality is performed in the digital domain, right up to the analogue front-end, as shown in Figure 2.2.

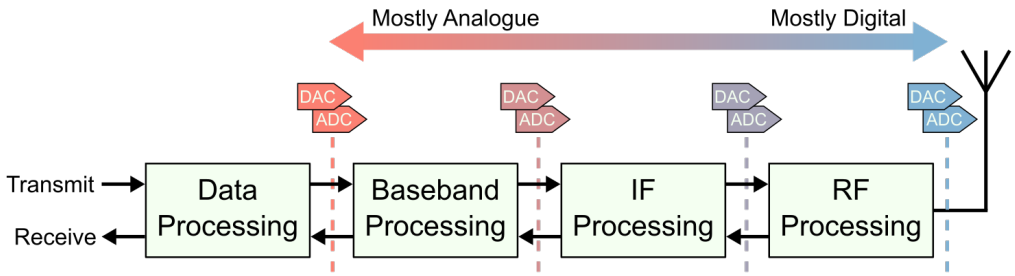

Figure 2.2: High level architecture of an SDR. The dashed lines depict the varying points in the signal path that could be digitised.

In previous technologies a major limiting factor was the resolution and speed at which data converters could operate, forcing the digitising process to happen at an

IF, rather than directly at the higher radio frequencies, causing systems to rely on less configurable, and more expensive, analogue components. In recent years, data converter technology has advanced to where data converters are capable of operating at RF, helping to move the digitisation process closer to the antenna.

The direct-RF, or RF-sampling, data converter is a device that is capable of operating at frequencies high enough to sample RF signals directly, without the need for an IF stage. These devices offer great potential for modern and future radio applications due to their flexibility and low power consumption compared to their analogue counterparts. Moreover, the typical issues found in analogue zero-IF architectures, such as offset and $1/f$ noise, do not apply to these digital zero-IF devices [75].

RF-sampling devices have been in the literature since at least the 1990s, with each iteration steadily increasing the maximum sample-rate of Digital to Analogue Converters (DACs) and Analogue to Digital Converters (ADCs), using a variety of innovative techniques [76–78]. However, these early devices did not have the double-digit resolutions required for modern communication protocols that demand high Signal-to-Noise Ratio (SNR), due to the limitations of the technology of the time. Following from this earlier research, semiconductor technology advanced to the point that results were reported in the literature showing data converters capable of much higher resolutions and bandwidths, at multi-Gigahertz sample rates [79, 80].

As of 2025, commercial RF-sampling devices from manufacturers such as Texas Instruments [81] and Analog Devices [82], provide multi-channel DACs and ADCs with up to 16-bit resolution, Gigahertz bandwidths, and include digital mixers and up- and down-conversion, all on the same silicon device. However, these devices contain no other digital signal processing capabilities, relying on separate embedded processors or FPGAs, connected via a high-speed JESD204B data connection [83]. An exception to this is the class of RF-sampling SoCs, such as the RFSoC and Versal RF from Advanced Micro Devices (AMD) [46, 50, 84], or Intel's Agilex 9 SoC FPGA Direct RF-series devices [85]

These RF-sampling SoCs integrate embedded processors, programmable logic, and RF-sampling data converters into a single device, enabling the development of more

flexible and efficient SDR systems, with lower power consumption and higher performance than discrete SDR architectures [85, 86].

Several factors make these devices ideal for CR applications. Integrated multi-channel, multi-Gigasample ADCs and DACs replace traditional analogue stages, reducing front-end complexity, cost, and power consumption. Unlike legacy SDR platforms, the tight coupling of embedded processors and programmable logic removes the need for host-side processing, providing the low-latency reconfiguration required for real-time radio operations [41, 46, 50].

The ability to replace much of the analogue front-end is another key enabler for CR. The multi-Gighertz instantaneous bandwidths of the RF-sampling SoCs enables them to capture and tune parameters entirely in the digital domain. In contrast to legacy platforms, RF-sampling SDRs can observe the entire band simultaneously, instead of scanning it piecemeal, increasing the likelihood of detecting spectrum holes and avoiding PUs.

Out of the various RF-sampling SoCs in the literature, the RFSoC is the most widely used in the literature and the only one commercially available at the time of writing. Therefore, the following section provides an overview of the RFSoC.

2.1.5 The Zynq UltraScale+ RFSoC

First produced in 2018, the RFSoC, at the time of writing, has four generations of devices, with the latest "DFE" device released in 2022. Each device contains up to 16 RF-sampling DACs and ADCs, with sample rates up to 10 Gsps and 5.9 Gsps respectively [87]. The RFSoC is based on AMD's Zynq UltraScale+ technology—combining FPGA fabric and embedded Arm processors—and includes "hardened" RF Data Converter (RF-DC) and Soft-Decision Forward Error Correction (SD-FEC) Intellectual Propertys (IPs) connected directly to the FPGA. A fifth-generation device, the Versal RF series has been announced that will support direct sampling up to 18 GHz at rates up to 32 Gsps [84]. However, this device is not expected to enter production until 2027.

The RFSoC is split into two regions: the Processing System (PS) and the PL. The PS contains multiple processor units, platform and power management, as well as

functionality to interface with off-chip peripherals. The PL contains the UltraScale+ FPGA fabric—capable of accelerating arbitrary DSP algorithms—as well as the RF-DCs and SD-FECs, as shown in Figure 2.3.

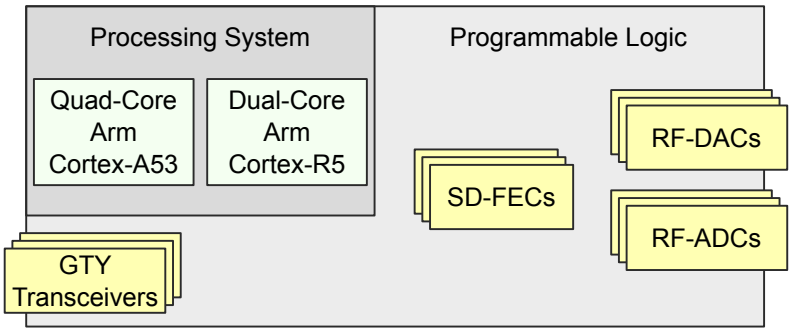

Figure 2.3: Block diagram showing the major constituent parts of the RFSoC.

A major feature of Zynq UltraScale+ devices is the ability to freely communicate data between the PS and PL portions of the device using dedicated hardware interfaces. These communication layers allow the PS components to be operated as a typical embedded processor, while computationally intensive arithmetic or complex algorithms can be accelerated on the FPGA. This allows data to move freely between them via the Advanced eXtensible Interface (AXI) interconnect, with throughput up to 85 Gbps and latency as low as the nanosecond range [50].

Software on the PS can be run either directly on the chip (known as baremetal), or on top of an operating system, such as Linux. Baremetal programs typically allow for greater control over the hardware and provide lower latency, but this comes at the cost of longer development times due to design complexity. In contrast, using an operating system, such as Linux, allows many of the low-level operations to be abstracted away, particularly memory management and task scheduling, making it easier to build upon pre-built software and drivers. The PS has been demonstrated to be capable of running a full Linux distribution, such as Ubuntu, making it ideal for on-device software development and prototyping [48, 50]. This removes the requirement for a separate communication link between the processor and the FPGA, such as USB or Ethernet used in legacy SDRs.

2.1.6 RFSoC Hardware

To facilitate evaluation, development, and prototyping of RFSoC-based systems, a variety of devices and hardware development platforms are available from AMD and third party vendors. Therefore, it is useful to select one of these platforms to use as a reference for the work presented in this thesis. The chosen development platform is the RFSoC4x2, which contains the XCZU48DR; a third generation RFSoC device.

The RFSoC4x2 is an academic-focused development platform exposing 4x 5 Gsps ADCs and 2x 9.85 Gsps DACs via SMA connectors with integrated baluns. The board includes 4 GB each of PS and PL DDR4 memory, as well as a QSFP28 interface supporting 100 GbE connectivity, promoting high-speed data offload [88, 89]. It is a more compact and cost-effective platform compared to other RFSoC development boards, making it ideal for academic research and teaching.

The on-chip resources for the XCZU48DR device are tabulated in Table 2.2, while a labelled photograph of the RFSoC4x2 development platform, highlighting the key components and features, can be found in Appendix A.

Table 2.2: XCZU48DR on-chip resources. Replicated from [6]

Category	**XCZU48DR**
Processing System	
Application Processing Unit	Quad-core Arm Cortex-A53 (up to 1.3 GHz)
Real-time Processing Unit	Dual-core Arm Cortex-R5F (up to 533 MHz)
RF Data Converter	
14-bit RF-ADC w/DDC	8 ADCs; up to 5.0 Gsps
14-bit RF-DAC w/DUC	8 DACs; up to 9.85 Gsps
RF input frequency max	6 GHz
SD-FEC	8
Programmable Logic	
System logic cells	930 K
CLB LUTs	425 K
Max distributed RAM	13.0 Mb
Total block RAM	38.0 Mb
UltraRAM	22.5 Mb
DSP slices	4,272

2.1.7 RFSoC for Cognitive Radio

The architectural features of the RFSoC directly address the technical hurdles of CR implementation compared to legacy SDR platforms. The integration of high-resolution, high-speed RF data converters enables the wide instantaneous bandwidths required for spectrum sensing and the detection of spectrum holes. Additionally, the tight coupling of the PS and PL provides the low-latency processing and reconfiguration required for the decision-making and reconfiguration stages of the cognitive cycle. This positions the RFSoC as an ideal platform for CR applications, and is the primary hardware device used throughout the work presented in this thesis.

2.2 Limitations of RFSoC Design Methodologies

Users of legacy SDR platforms benefit from a mature ecosystem of tools, including MATLAB/Simulink [90–93], LabVIEW [94–96], GNU Radio [91, 97–99], and Python via libraries such as SoapySDR [100–103]. These tools provide high-level abstractions of the underlying radio hardware, promoting high productivity and rapid prototyping [71]. This significantly lowers the barrier to entry for SDR development, allowing researchers to focus on algorithmic innovation [104].

However, these tools generally lack specific support for RFSoC beyond a few examples. For instance, the Ettus USRP X410 SDR contains a first generation RFSoC and supports many of the tools mentioned above. However, its use is limited to a simple "black box" approach, where the RFSoC is treated as a simple data converter, with limited support for custom IP development or access to the embedded processor [105]. Furthermore, it is significantly more expensive than other research-focused RFSoC development boards [88]. Additionally, [89] showed that the RFSoC could be used alongside GNU Radio, but this was a custom implementation that does not scale to the general case.

2.2.1 Conventional SoC Design Flow

To take advantage of the full capabilities of the RFSoC, developers are instead faced with the conventional design flow for SoCs: a Hardware Description Language (HDL) is

used in Vivado [106] to develop the various IP blocks required for the FPGA design, using testbenches to confirm correct operation of each component in an iterative fashion. The IP blocks are then connected together to form the full FPGA design. Software for the embedded processor is then developed in Vitis [107] using either C or C++, including linking to any drivers or libraries required, then compiled to a binary format using a cross-compiler. This software is either compiled to run directly on the hardware (i.e., baremetal), or as a software layer running on top of a custom Linux Operating System (OS), generated by using the Petalinux toolchain [108].

While this design flow is typically considered the best method to achieve optimal performance from the hardware, it demands a significant amount of domain-specific knowledge and can result in lengthy development times due to its complexity [50]. An engineer using this method must have intricate knowledge of all aspects of the underlying hardware; including the PS, PL, and RF data converters; as well as proficiency in both embedded and hardware description languages. Debugging can add further complexity to this design flow due to the requirement to test the system both piecemeal and as a whole. For example, the interaction between PS and PL, shared memory access, and RF signals transmitting or being received by the data converters. The ability for run-time signal inspection is also difficult without the use of external instrumentation, which can be costly for the wide bandwidths and multi-Gigabit data rates of the RFSoC.

Moreover, the traditional design flow is not conducive to rapid prototyping, as the iterative process of RTL synthesis, place-and-route, bitstream generation, software compilation, and kernel generation creates a "productivity bottleneck"—where productivity it typically measured in terms of development time and lines of code [109–111].

It should be noted that these issues are not unique to RFSoC development, but are common across all FPGA and SoC development, albeit with different tools and languages used for different platforms [109]. With that said, the RF data converters of the RFSoC do add an additional layer of complexity.

2.2.2 Productivity Focused Development Tools

To combat this productivity bottleneck, a number of alternative commercial and open-source design tools are available that aim to abstract much of the complexity of embedded and FPGA development away. This allows engineers to focus on algorithms, rather than be concerned about the underlying structure of the hardware. Moreover, these alternative tools tend to use programming languages and user interfaces that are much simpler than the tools mentioned above, providing users with "friendlier" development environments.

FPGA Design Tools

For FPGA development, there are a number of alternative HDLs that use extensions to higher-level languages, such as C++ [112] or Python [113], known as High-Level Synthesis (HLS). These extended languages use custom compilers to generate lower-level HDLs such as Very High Speed Integrated Circuit Hardware Description Language (VHDL) or Verilog; generally aimed at engineers more experienced in traditional general-purpose programming languages. These tools have been shown to significantly reduce development time. For example, [114] showed that in all cases surveyed, HLS reduced development time—in some cases by up to 50%. However, HLS often requires a coding style that deviates significantly from the standard syntax of the base language due to compiler constraints. As such, achieving even modest performance targets still requires a deep understanding of the underlying hardware [114]. Additionally, the performance of HLS generated code is often significantly worse than that of handwritten HDL, with the survey performed by [114] showing a mean performance degradation of approximately 33%.

MathWorks also provides their own tools for both embedded [115] and FPGA [116] design, using either MATLAB code or within their graphical programming environment, Simulink. These tools are able to generate lower-level HDL and C from an abstract description. Due to this abstraction from low-level constructs, the use of these tools can greatly improve productivity, reducing development time and increasing the ability to reuse code in other applications, although usually to the detriment of performance.

However, it has been shown that, in some conditions, the performance of the code generated by these abstracted tools can approach that of handwritten low-level code, while greatly reducing the amount of development time [117–119].

Model Composer (previously System Generator) [120] is a model-based design tool from AMD that integrates into MathWorks' Simulink environment, providing a graphical environment for FPGA algorithm design, simulation, and verification. It provides a blockset of DSP primitives and IP blocks that can be combined with Simulink's native signal processing tools—such as signal generators, oscilloscopes, and spectrum analysers—to develop and verify designs within a single environment. Once verified, Model Composer generates VHDL or Verilog IP-XACT IP blocks for import into Vivado. Model Composer has been shown to greatly improve productivity. For example, [121] showed a 10:1 improvement in development efficiency when implementing a satellite communication system, while in [122] a 2-3x reduction in design time was observed when implementing a space-time coded telemetry receiver using the tool, with 10% clock rate and 30% area penalties observed compared to a hand-coded design.

While each of these tools can be used to design IP blocks for the PL of the RFSoC, they do not provide support for the embedded software development on the PS, or for the interaction between the two. This means that users of these tools are still required to use the traditional design flow for embedded software development, and must have a detailed understanding of the underlying hardware to ensure that their software can interact with their hardware design.

Embedded Design and Runtime Environments

The software ecosystem for SoC and RFSoC embedded development is more mature than the FPGA equivalent, as the PS functions as a standard embedded processor with specialised peripherals. Users typically work in Vitis to build baremetal applications in C or C++, or build applications on top of Petalinux or Yocto environments. However, a few notable tools exist that provide an abstracted development environment and promote productivity.

PYNQ is an open-source project from AMD that provides a simplified framework for embedded development on AMD's Zynq and Zynq UltraScale+ range of SoC devices. PYNQ runs an Ubuntu-for-Arm operating system, using AMD's custom Linux kernel. It challenges the conventional embedded C development workflow by using a Jupyter web interface as the Integrated Development Environment (IDE), and Python as the embedded language [48, 50, 123, 124]. Along with this development environment, PYNQ also supplies a Python library that exposes many of the embedded features of the target SoC. This includes Memory-Mapped Input/Output (MMIO) for interacting with the physical address space, and drivers for IPs such as the AXI Direct Memory Access (DMA).

The use of Python as an embedded language is unconventional due to its perceived slow performance metrics, such as execution time, due to it being an interpreted language [48, 125]. However, because of its popularity, ease of use, and extensive ecosystem of libraries; development time tends to be significantly faster than the traditional C approach. Moreover, it is shown that, under certain circumstances, Python can perform well in low-level operations, such as reading and writing to memory [48]. While no exact measures of productivity have been given in the literature, PYNQ has been demonstrated to reduce lines of code [48] and reduce system complexity [126]. Additionally, in [127] the workload of undergraduate teaching staff was found to be reduced with the introduction of PYNQ to the curriculum.

Originally released in 2016, PYNQ is widely used in the research community, supporting a range of applications, including DSP [48], machine learning [128], and SDR [50].

Beyond PYNQ, [129] introduced FOS, a framework for abstracted embedded software development on SoC platforms. FOS provides a high-level, productivity-focused API for interacting with the hardware, and supports multiple programming languages, including Python and C++. In [130], Redsharc is introduced, a programming model and network-on-chip solution for SoC architectures, and supports both AMD and Intel platforms. Additionally, [131] introduced Cynq, a runtime library based on PYNQ, but written in C++ instead of Python. However, while these tools show improved productivity and

similar or better performance than PYNQ, they have not been widely adopted, have not been actively maintained, and do not support the RFSoC.

In summary, the RFSoC ecosystem lacks a full-stack development flow that supports rapid prototyping and productivity. Current abstracted design tools are fragmented, and experimental research tools lack the necessary support and maintenance for widespread use. This gap suggests that a new methodology should be built upon well-maintained tools rather than the development of another custom solution. Therefore, the challenge is to identify and extend a collection of existing tools into a cohesive full-stack design methodology that prioritises productivity.

2.3 Frequency Planning

High QoS is an essential requirement of CR as it ensures reliable communication and efficient use of the spectrum [18]. However, the presence of noise and spurious emissions in RF systems can degrade QoS, leading to misidentification of PU signals or conversely, masking weak signals of interest [14, 132].

Noise in RF systems comes from a variety of different sources including external components, such as amplifiers and clocks, and within the data converters themselves. For data converters, there are two main sources of noise: quantisation noise, which results from the rounding operation when converting a continuous analogue voltage to a discrete digital representation, and thermal noise, which is produced by quantum effects within the data converter circuitry and manufacturing imperfections [133, 134]. Additionally, limitations of the manufacturing process can cause non-linearities in the data converter transfer function, leading to spurs in the output.

Both noise and spurs can have adverse effects on the quality of an RF signal. There are a variety of metrics that are used to characterise the performance of the data converter—covered in detail in [50]. However, there are two principal performance metrics that are used throughout the work presented in thesis: SNR and Spurious-Free Dynamic Range (SFDR).

SNR is defined as the ratio between the power of the input signal and the total noise power of the system, excluding any spurious components, usually given in dB.

It quantifies how much the signal of interest stands out from the background noise, with higher values indicating better signal quality [135]. SFDR is a measurement of the usable dynamic range of a data converter with respect to any spurious components present in the Nyquist Zone of interest [50]. A Nyquist Zone is defined as a frequency range equal to that of the Nyquist rate, starting at 0 Hz. For example the first Nyquist Zone is from 0 Hz to $\frac{f_s}{2}$, the second Nyquist Zone is from $\frac{f_s}{2}$ to f_s, and so on. SFDR is defined as the ratio of the power of the input signal to the power of the most prominent spur, typically given in dBc (decibels relative to the carrier).

Both noise and spurs reduce SNR and SFDR, which in turn degrades system QoS by reducing throughput and increasing latency. For CR and SDR systems operating in DSA scenarios with stringent QoS requirements, maintaining high SNR and SFDR is critical to ensure reliable communication and to avoid interference with other users [136].

2.3.1 RF Data Converters and Interleaving ADCs

The RF data converter subsystem on the RFSoC contains multiple RF-speed ADCs and DACs, supporting sample rates up to several Gigahertz [87]. To achieve these high sample rates while mitigating clock jitter issues, the RF-ADCs use an interleaving technique.

Interleaved ADCs operate by using two (or more) identical ADCs, running from the same sample clock, that simultaneously sample a signal and combine these samples to create a higher sample rate than either individual ADC can operate at. This is achieved by creating a phase difference between the sample clock for each ADC, such that the signal is sampled at different points in time. For example, for a two-interleaved ADC, each with a sample rate of 100 MHz, the first ADC will have a phase of 0° and the second ADC will have a phase of 180° . When the two sampled signals are then interleaved, the resultant signal will have a sample rate of 200 MHz, as depicted in Figure 2.4.

The ADCs on the RFSoC are set up with either four (Quad) or two (Dual) ADCs per tile, with an interleaving factor of 4 or 8, respectively. The XCZU48DR has a Dual RF-ADC configuration with an interleaving factor of 8, making the phase difference

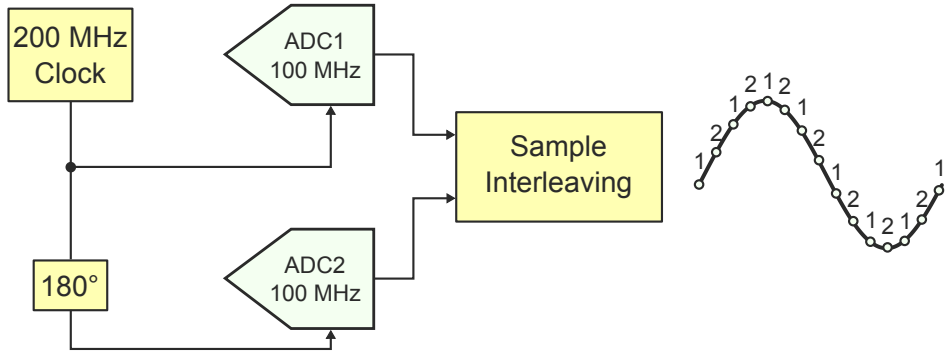

Figure 2.4: Simplified diagram of how two interleaved 100 MHz ADCs create a resultant signal with a sample rate of 200 MHz.

between each ADC 45° ($\frac{\pi}{4}$ radians). Each ADC has a maximum sample rate of 5 GHz, meaning that each sub-ADC has a maximum sample rate of 625 MHz.

The interleaving process has one major disadvantage in that every ADC is slightly different. Manufacturing tolerances, imperfections in the silicon die, and temperature, among other factors, cause each sub-ADC to behave differently, leading to spurious emissions (interleaving spurs) when the signals are combined.

2.3.2 Spurious Emissions of RF-Sampling Devices

The International Telecommunication Union (ITU) defines spurious emissions as any unwanted signal or emission outside the main signal's bandwidth that can be reduced without affecting the main signal [137]. Keysight expands this definition to include both in-band and out-of-band emissions that emanate from the device under test [138]. In this thesis, spurious emissions (or spurs) refer specifically to unwanted signals generated during the data conversion process of RF-sampling devices. These may appear in-band or out-of-band and have the potential to interfere with the signal of interest.

During operation, all data converters will create spurs which affect the signal passing through them, potentially causing interference and distortion, and reducing SNR. Spurs occur due to a combination of physics and inaccuracies in the manufacturing process but, consequently, it is relatively simple to calculate where on the spectrum they will

occur. In the following sections the most significant spurious emissions caused by data converters are explored.

Harmonic Distortion

Harmonic distortion is caused by non-linearities along the data converter signal path, which can manifest as spurs at equally spaced intervals across the spectrum. The frequencies of these spurs are directly related to the frequency of the input signal, as integer multiples, and can be calculated by

$$HD_n = nf_{in}, \tag{2.1}$$

where n is the order of the harmonic and f_{in} is the input frequency. The amplitude of the harmonics will decrease as n increases due to the physical properties of the data converter forming a low-pass filter [139]. Typically, the first three harmonics (HD_2, HD_3, HD_4) are the most prominent [140].

Figure 2.5 shows the first four harmonics of an input tone, showing this integer multiple relationship. For example, if the input frequency is 200 MHz, then HD_2 will be 400 MHz, HD_3 will be 600 MHz, and so on. The amplitude of the harmonics is dependent on the degree of non-linearity the data converter displays as well as the amplitude of the input signal. Typically, the manufacturer's datasheet will only contain data regarding the first two harmonics, sometimes with a variety of input signal amplitudes [141, 142].

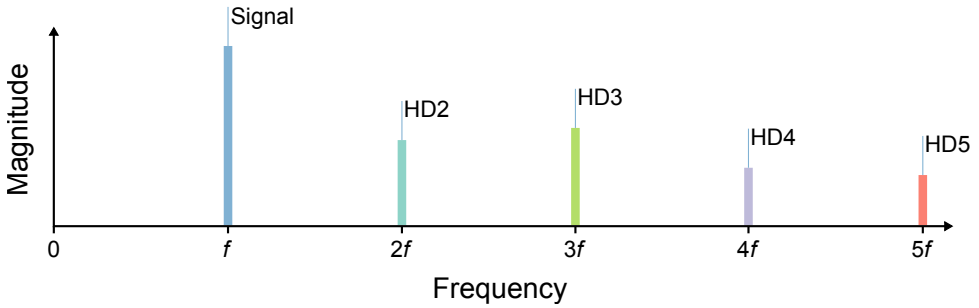

Figure 2.5: The first four harmonic components of an input signal.

A common measurement of these spurs is total harmonic distortion (THD), which is defined as the ratio between the power of the input signal and the power of the sum of all harmonic components—usually stopping at the sixth harmonic [140].

Interleaving Spurs

As discussed earlier, interleaving ADCs combine multiple phase-shifted, lower sample rate ADCs to achieve an overall higher sampling rate. However, mismatches between the sub-ADCs can cause spurs to occur in the signal path. The first of these is where the Direct Current (DC) offset between sub-ADCs is mismatched causing an oscillating signal to occur, related to the sample rate and the interleaving rate. The set of DC offset interleaving spurs can be calculated by

$$f_{DC_k} = \frac{k}{M} f_s, \quad \text{for } k = 0, 1, 2 \ldots, M-1, \tag{2.2}$$

where M is the interleaving rate and f_s is the sample rate. Figure 2.6 shows the DC offset effect for a 2-interleaved ADC.

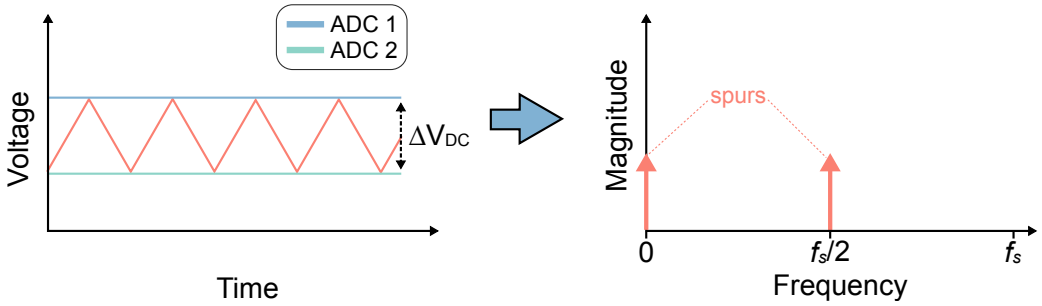

Figure 2.6: DC offset mismatch for a 2-interleaved ADC.

The remaining interleaving spurs arise from gain and timing (phase) mismatches. Unlike DC offset spurs, which only relate to the sample rate, gain and phase spurs are also a function of the input frequency. This set of spurs can be calculated by

$$f_{gt_k} = \frac{k}{M} f_s \pm f_{in}, \quad \text{for } k = 0, 1, 2 \ldots, M-1. \tag{2.3}$$

Figure 2.7 shows the effect of both gain and phase mismatches on a signal for a 2-interleaved ADC.

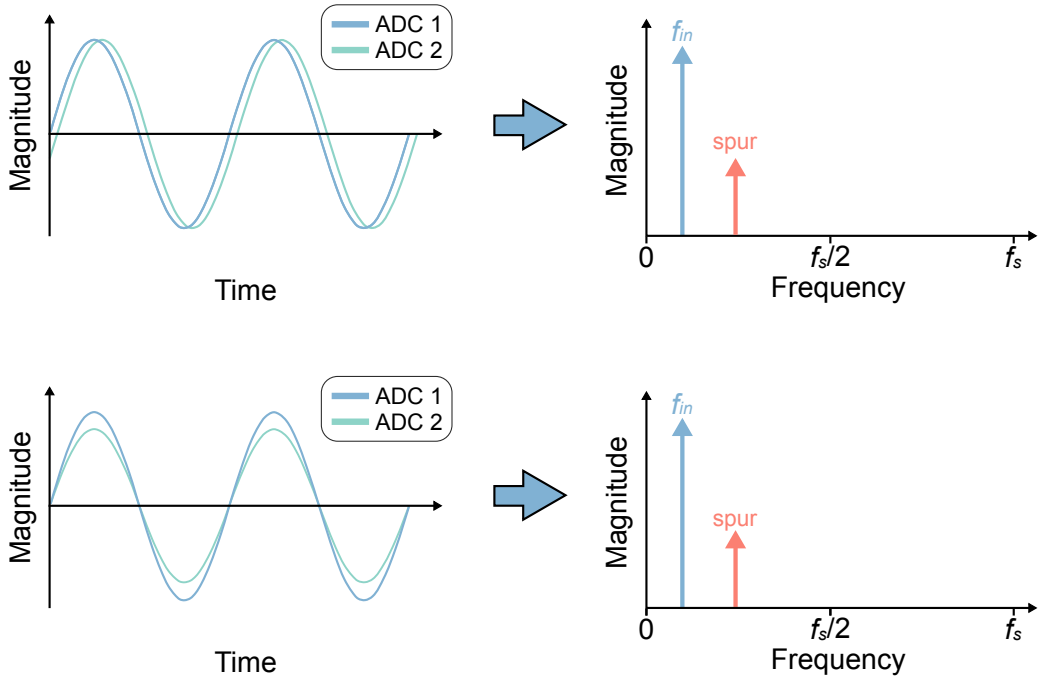

Figure 2.7: Phase (top) and gain (bottom) mismatch for a 2-interleaved ADC.

Since harmonic distortion can occur before the interleaving process, gain and phase interleaving spurs can also affect the harmonic components in a manner similar to the input signal. Although these spurs are typically much lower in amplitude, they can still introduce in-band interference if not taken into account when frequency planning. Their locations can be calculated by

$$f_{hmix_k} = \frac{k}{M} f_s \pm HD_n, \quad \text{for } k = 0, 1, 2 \ldots, M-1. \tag{2.4}$$

Clock Spurs

The sample clocks of the data converters can be another source of spurs along the signal path. The relationship between the input signal frequency, f_{in}, and the clock frequency, f_{clk} can have adverse effects on signal quality. For example, if f_{in}/f_{clk} is an integer,

quantisation noise tends to be concentrated around the harmonics, making it more difficult to remove them with filtering [135].

The clock spurs occur due to the higher frequency component effectively mixing with the lower component frequency, similar to a heterodyne mixer. These clock spurs can be calculated by

$$f_{pllmix} = f_{in} \pm f_{clk}. \tag{2.5}$$

Spurious Emissions of Bandlimited Signals

The spurs defined in the previous sections assume a single-tone input signal. However, most communications signals are bandlimited, such that they contain multiple frequencies over a finite bandwidth. Therefore, any spurs related to the input signal will also be bandlimited, increasing the likelihood a spur will overlap with the signal of interest, causing in-band interference. These include gain and phase interleaving spurs, clock spurs, and harmonics.

Harmonics are generally the most challenging to mitigate due to their nf_{in} relationship with the input signal, which results in much larger bandwidths. For example, the second harmonic has twice the bandwidth of the input signal, while the third harmonic has three times the bandwidth. In contrast, gain and phase interleaving spurs, as well as clock spurs, have bandwidths equal to that of the input signal, making them slightly easier to manage.

Figure 2.8 depicts an example of bandlimited harmonics with a 100 MHz bandwidth input signal at a centre frequency of of 450 MHz, and a sample rate of 4 Gsps.

2.3.3 Frequency Planning for RF-Sampling Devices

As was shown in the previous section, the spurious emissions caused by data converters are deterministic, and their frequency locations can be calculated from system parameters. This allows spurs to be calculated before a signal is transmitted or received, and to determine whether a specific configuration will result in a spur (or spurs) overlapping with the signal of interest. Frequency planning uses this insight to tune data converter

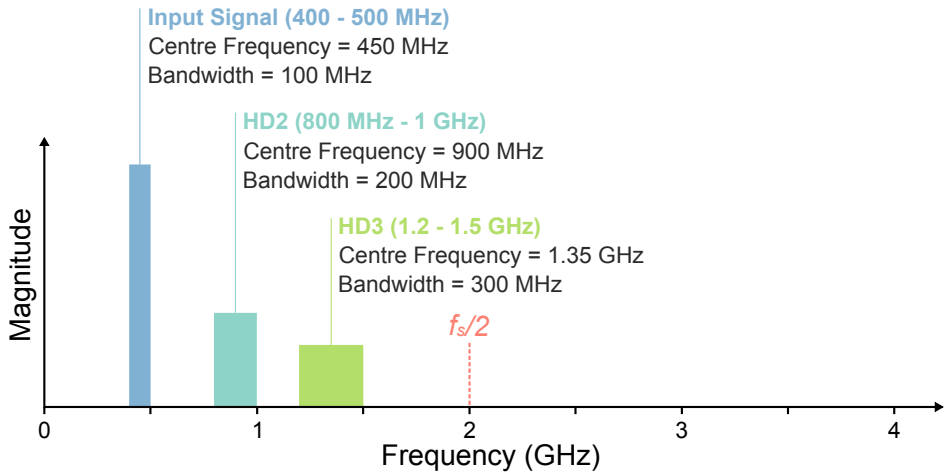

Figure 2.8: Harmonic bands resulting from a bandlimited signal.

parameters, such as sample rate and clock frequency, to avoid spurs causing in-band interference.

Frequency planning for RF-sampling devices must also account for additional effects. For example, in RF-DACs harmonics can appear in the spectrum very close to the signal of interest when transmitting in the upper Nyquist bands. In RF-ADCs spurs can alias into the first Nyquist Zone, resulting in a congested spectrum that makes it difficult to avoid in-band interference [50]. The rest of this section explores these factors within the context of frequency planning for the RFSoC.

Effects of Aliasing

When calculating the frequency positions of spurs using the equations discussed in Section 2.3.4, it is clear that for certain configurations, spurs will occur at frequencies within the upper Nyquist Zones. Any spurs above the Nyquist rate will be aliased into the first Nyquist Zone, potentially interfering with the signal of interest.

The resultant frequency of a spur after aliasing depends on which Nyquist Zone it occupies. For example, a spur in the second Nyquist Zone with a frequency of f_{spur} will be aliased into the first Nyquist Zone with frequency $f_s - f_{spur}$, where f_s is the sample rate. Similarly, a spur in the fourth Nyquist Zone will have an aliased frequency of $2f_s - f_{spur}$, and a spur in the sixth Nyquist Zone will have an aliased frequency

of $3f_s - f_{spur}$. This repeating pattern allows the frequency of any spur located in an upper Nyquist Zone to be identified using a simple algorithm: the frequency of the spur modulo the sample rate is taken, with the result then being subtracted from f_s if it falls within the second Nyquist Zone. A mathematical representation of this algorithm is shown in Equation 2.6.

$$f_{\text{aliased}} = \begin{cases} f_{\text{spur}} \bmod f_s, & \text{if } f_{\text{spur}} \bmod f_s \leq \frac{f_s}{2} \\ f_s - (f_{\text{spur}} \bmod f_s), & \text{if } f_{\text{spur}} \bmod f_s > \frac{f_s}{2} \end{cases} \tag{2.6}$$

Figure 2.9 shows a practical example of aliasing where two harmonics of a 100 MHz bandwidth input signal, with a centre frequency of 1.15 GHz, are located in the second Nyquist Zone of a 4 Gsps RF-ADC. As described in Section 2.3.4, HD_2 will have a bandwidth of 200 MHz centred at 2.3 GHz, while HD_3 will have a bandwidth of 300 MHz centred at 3.45 GHz. Using Equation 2.6, after aliasing HD_2 will have a centre frequency of $4 - 2.3 = 1.7\ GHz$, while HD_3 will have a centre frequency of $4 - 3.45 = 0.55\ GHz$.

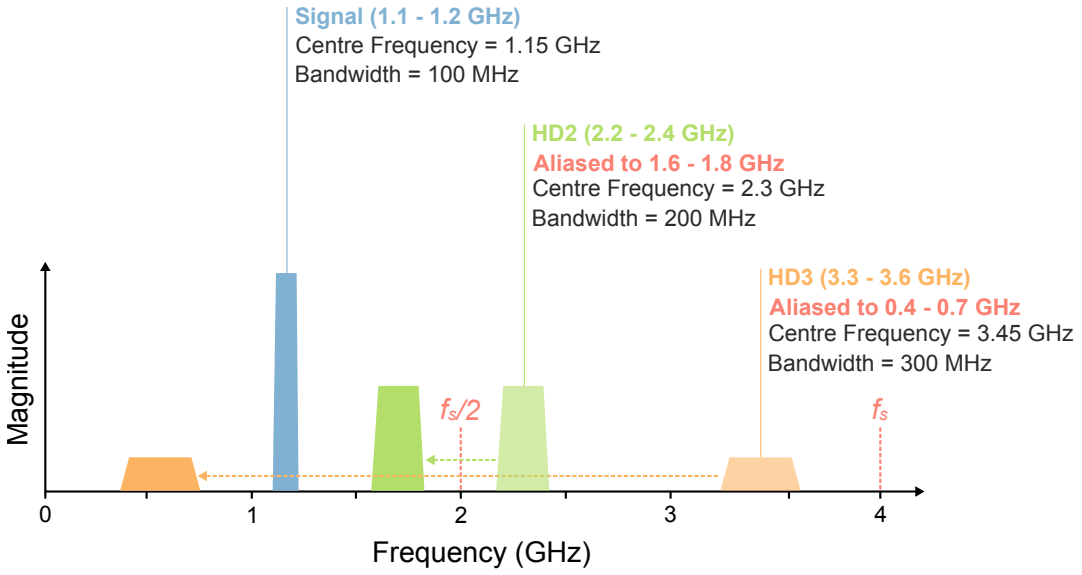

Figure 2.9: Example of aliased spurs.

As can be seen from this example, HD_2 and HD_3 are now positioned on either side of the signal of interest, requiring a band-pass filter to suppress the spurs. A gap of 400 MHz is left between the closest spur and the signal, amounting to 10% of the sample rate—more than sufficient for a relatively relaxed filter to be used. However, in this example only the first two harmonic spurs are taken into account. When considering additional harmonics, as well as interleaving spurs and clock spurs, the spectrum can become congested. This makes it more challenging to find a configuration that leaves sufficient space around the signal of interest, requiring a filter with a much tighter transition band.

Effects of Digital Down Converters

As discussed in Section 2.3.1, the data converters of the RFSoC also contain digital complex mixers, as well as Digital Up Converters (DUCs) and Digital Down Converters (DDCs) in the RF-DACs and RF-ADCs, respectively. If used, both the Numerically Controlled Oscillator (NCO) and the DDC of the RF-ADCs will have an effect on spur frequency along the signal path and, because the DDCs come after analogue to digital conversion, the positioning of the spurs affects the aliased spurs.

In the case of the digital complex mixer, as well as shifting the signal of interest down to (or near) baseband, all spurs present in the first Nyquist Zone will also be shifted by the same value. Figure 2.10 depicts an example of this frequency shifting with a 100 Msps sample rate ADC, a signal with a centre frequency of 40 MHz, and an NCO mixing frequency of 35 MHz. For simplicity, this example uses a purely real, single tone signal (i.e. it has no imaginary component and is not bandlimited). Additionally, to make the calculations easier to follow, only the first harmonic, HD_2, is considered.

First, HD_2 is aliased into the first Nyquist Zone, from 80 MHz to 20 MHz. Because the signal is purely real it will also have negative images at -40 MHz, and the aliased version of HD_2 will have a similar image located at -20 MHz. When the signal is mixed with the NCO of the digital mixer, the signal of interest will shift from 40 MHz to 5 MHz, while HD_2 will shift into the negative frequencies, from 20 MHz to -15 MHz. Similarly, the negative images of the signal of interest and HD_2 will shift and wrap

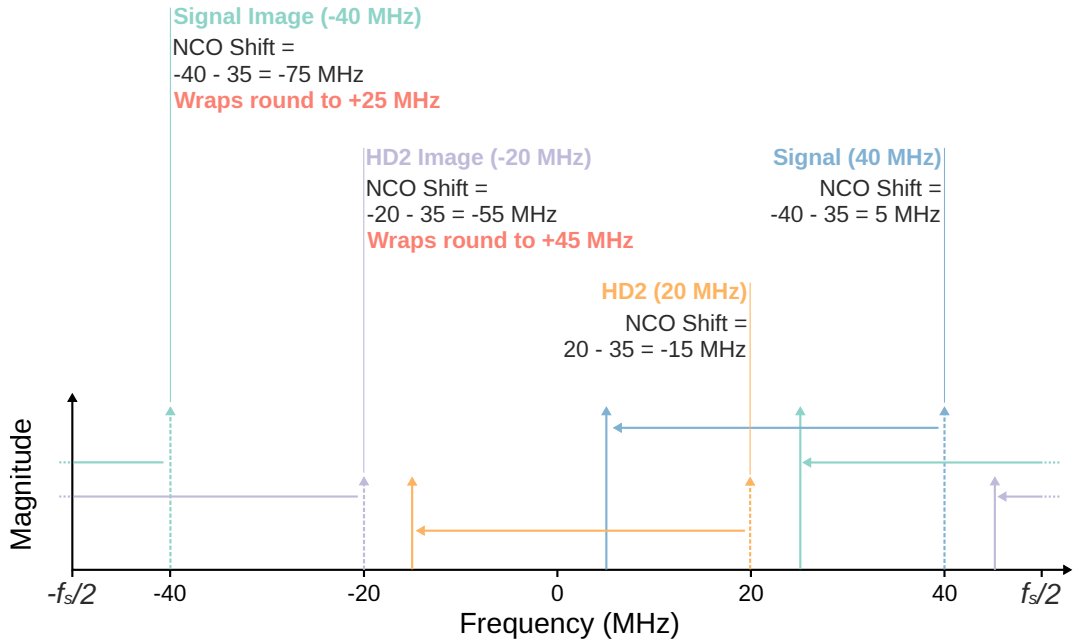

Figure 2.10: Example of how the NCO shifts frequency components and their images in the complex spectrum.

around to 25 MHz and 45 MHz respectively, placing them both in the positive Nyquist Zone. To summarise this phenomenon, the shifts are listed below.

- The input sine wave shifts from 40 MHz to 5 MHz.
- The negative image of the input sine wave shifts from -40 MHz and wraps around to 25 MHz.
- HD2 is shifted from 20 MHz to -15 MHz.
- The negative image of HD2 is shifted from -20 MHz and wraps around to 45 MHz.

As can be seen from Figure 2.10, the complex mixer has shifted all components of the signal, including the negative image components. Moreover, the first Nyquist Zone now contains three components; the signal of interest and its image, as well as the image of HD_2. As more spurs are considered, the spectrum can become congested after aliasing and mixing, making it difficult to find a favourable frequency plan. A simple way to avoid this is to just increase the sample rate, leaving more space for spurs to

sit away from the signal of interest, and avoid the amount of frequency wrapping that occurs.

Related to this frequency wrapping, decimation can also change the frequency of spurs within the spectrum. Decimation is the process of reducing the sample rate of a signal by first suppressing any unwanted signals above the new Nyquist rate, followed by removing $R_D - 1$ samples out of every R_D samples. Any signals that appear outside of the new Nyquist rate will then be subject to aliasing.

The use of an anti-aliasing filter will suppress any spur above the new Nyquist rate. However, as these filters are unable to achieve a true "brick-wall" response, any spurs located near the edges of the Nyquist Zone may not be sufficiently attenuated, and may be difficult to remove with additional filtering. Figure 2.11 depicts an example of the effects of decimating by continuing the example from the previous section, decimating the original signal by a factor of two. This process is listed below.

- The 2x decimation reduces the Nyquist rate from 50 MHz to 25 MHz.
- Having previously been demodulated, the input sine wave and HD2 are within the new Nyquist Zone 1 (between 0 Hz and half of the new sample rate, i.e. 25 MHz), and therefore remain unchanged.
- The image of the input signal now resides at exactly the new Nyquist rate of 25 MHz, and is not fully attenuated by the anti-alias filter.
- The HD2 image is now outside the new Nyquist Zone 1, and is wrapped from 45 MHz to -20 MHz. However, the anti-alias filter should suppress this spur sufficiently, and therefore it can be disregarded.

It is worth noting that in RF-DACs, where mixing and interpolation precede the digital to analogue conversion process, spurs are not affected in the same way. However, if any additional mixing takes place in the analogue domain, for example if the RFSoC is used as an IF processor and the signal is mixed up to millimetre wavelengths (mmWave), then spurs will be shifted by this mixing process.

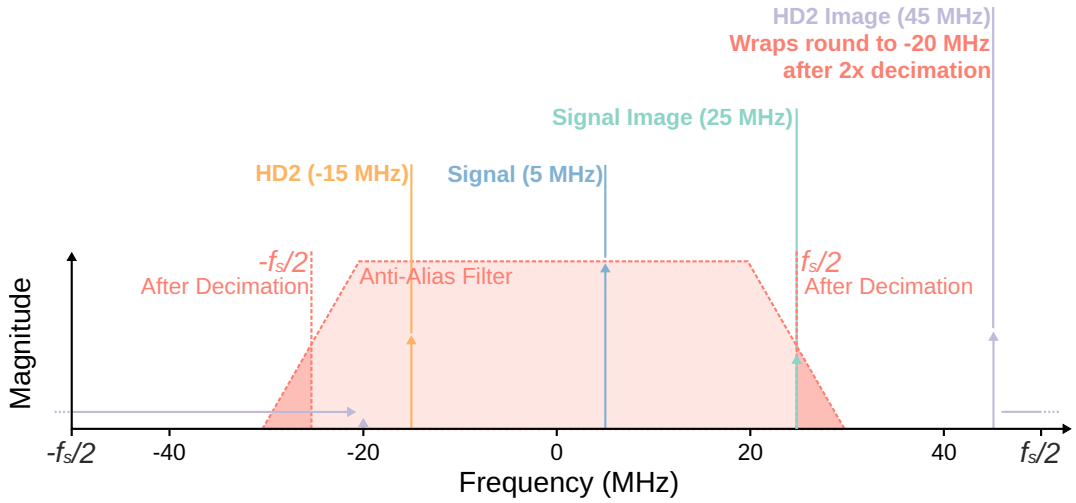

Figure 2.11: Example of how decimation can affect the frequency and amplitude of spurs.

Filtering to Eliminate Spurs

Spurs that are caused by the sampling process can be mitigated with good frequency planning. This protects the signal of interest from in-band interference, leaving the out-of-band spurs to be suppressed with filtering. The type of filter required depends on the state of the spectrum after frequency planning, and the type of data conversion process.

For ADCs, spurs occur during the data conversion process and are shifted through the signal path of the DDC, meaning that a digital filter is required to suppress any spurious components.

The frequency response of the required filter will be determined by the location of the spurs in relation to the signal of interest. For example, if spurs are located only at higher frequencies than the signal of interest, then a low-pass filter can be used, while if spurs are located on either side, then a band-pass filter will be required. However, as is shown in the following example, effective frequency planning can allow much of the filtering to be performed during decimation.

Figure 2.12 depicts the resultant spectrum after the example from the previous section is decimated by an additional factor of two. This process is listed below.

- The additional 2x decimation reduces the Nyquist rate from 25 MHz to 12.5 MHz.
- The input signal is located within the new Nyquist Zone 1 and remains unchanged.
- HD2 now wraps from -15 MHz to 10 MHz, placing it near the Nyquist Zone 1 boundary where it may not be sufficiently suppressed and additional filtering may be required.
- The signal image at 25 MHz folds to DC due to decimation, but is expected to be sufficiently attenuated by the anti-alias filter and can be ignored.

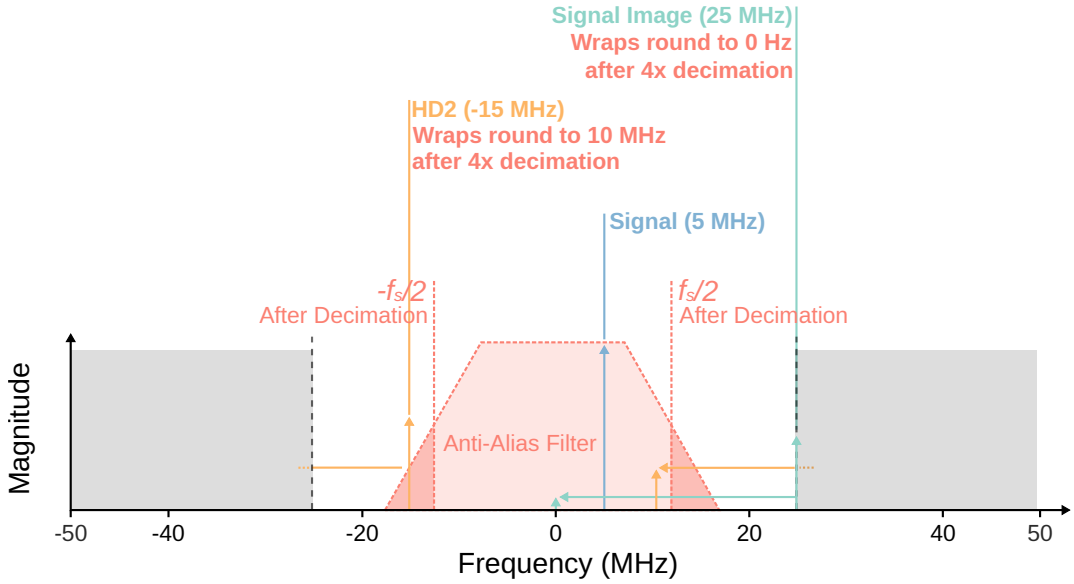

Figure 2.12: Example of decimation filtering to suppress spurs.

For DACs, spurs will be present only after the digital to analogue conversion, meaning that an analogue filter is required to sufficiently suppress the spurs. This is particularly important for transmitters as spurious components can leak into adjacent radio channels, causing interference with other users.

When considering run-time reconfigurable systems, such as CR, it is worth considering that the filtering process must be reconfigurable as well. As has been shown in this section, the frequency locations of spurs, while deterministic, is highly dependent on the configuration of the data converters. Therefore, a filter must be able to adapt to

the changing frequency location of the spurs when a device is reconfigured, in order to provide the best performance.

2.3.4 Frequency Planning for Cognitive Radio Systems

In traditional wideband digital radio applications, where the bandwidth of a signal is close to the maximum bandwidth of the data converter, in-band interference from spurs can be difficult, if not impossible, to avoid. In RF-sampling devices, where sample rates are many times that of the signal bandwidth, in-band interfering spurs can be avoided with good frequency planning and filtering. This suggests that an RF-sampling SDR implementing CR would have a frequency planner as a key component of the system, responsible for calculating spur-free configurations for the data converters to ensure high QoS in dynamic environments.

For fully autonomous systems, such as in DSA and CR, the radio must continuously sense the spectrum, decide on a free channel, and reconfigure its data converters on-the-fly, as shown in Figure 2.13. In this system, a dedicated RF-ADC first performs real-time spectrum sensing, feeding a CR/DSA decision engine, which identifies an available centre frequency and bandwidth to transmit or receive on. The frequency planner then computes a valid sample rate and Phase-Locked Loop (PLL) configuration that results in a spur-free region around the signal of interest, then instructs the RF control block to program the RF-DACs or RF-ADCs accordingly. Additionally, a reconfigurable filter is used to remove the out-of-band spurs.

Several frequency planning tools currently exist that allow users to input device parameters and visualise the resulting spectrum. However, these tools are limited in their capabilities, and are not suitable for real-time operation. For example, vendor-specific tools from manufacturers such as Analog Devices [52], Texas Instruments [143], Keysight [53], and AMD [144] remain Graphical User Interface (GUI)-driven and unscriptable, making them incompatible with autonomous systems.

The academic literature on the subject is also limited. In [54] a Java applet for single-tone intermodulation analysis allowed users to plot mixer products against LO, RF, and IF, but remained confined to a graphical interface and single-tone signals.

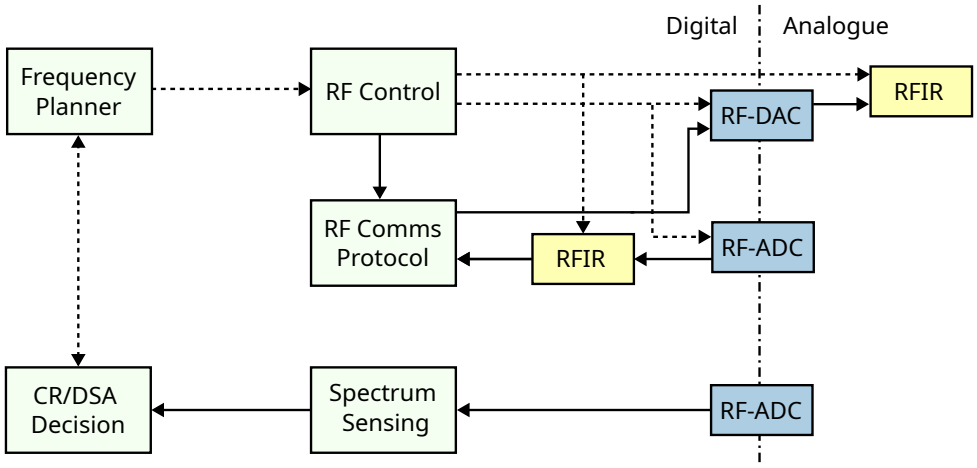

Figure 2.13: A fully autonomous system, where the frequency planner is used to reconfigure the RF-ADC and RF-DACs on-the-fly. Solid lines represent the data flow, while dashed lines represent control signals.

In [145] the authors demonstrated an interactive tool that swept LO, RF, and IF simultaneously, reducing calculation time, but still bound users to a graphical interface and offline operation. The Distances Chart of [146] introduced a novel separation-metric plot to simplify spur distance calculations, but was limited to only harmonic components and depended on pre-computed diagrams for each new configuration. Additionally, the authors of [139] introduced a frequency planning method based on an optimization process that calculated the IF and LO frequencies that provide the highest SFDR at a specific centre frequency. However, this method was limited to RF-DACs, only considered harmonic mixing products, and did not provide a complete solution for real-time frequency planning.

The closest solution for automated frequency planning is presented in [147]. Their graphic-analytic model predicts spur-free regions for arbitrary Direct Digital Synthesis (DDS) configurations by identifying intersections between harmonic spurs and guard bands to define “clean” sectors. In effect, this provides a formula for identifying spur-free regions for a given set of input parameters.

However, this method only addresses DDS, is limited to single-tone signals, and only takes into account two harmonic components. Extending this approach to RF-sampling transceivers would require modelling a significantly wider array of effects, including

interleaving and PLL spurs, additional aliasing effects introduced by decimation, frequency shifting due to NCO mixing, and multiple Nyquist Zones. Together, these effects greatly increase the number of spurs and intersections that must be evaluated to identify spur-free regions. For this reason, the approach used in [147] does not easily scale to the more general frequency planning problem addressed here.

From this review, it is clear that both the vendor tools and work in the literature are focused either on manual, offline analysis or simplified architectures, and does not establish whether real-time, automated frequency planning is practical for RF-sampling systems. In particular, it remains an open question whether spur-free configurations can be identified quickly enough to support real-time operation in dynamic environments, such as CR systems.

2.4 Reconfigurable FIR Filters

FIR filters play an important role in any SDR system; from low-pass filters during decimation or interpolation stages, to band-pass filters used in channelisers. FIR filters are typically preferred over Infinite Impulse Response (IIR) filters due to their stability, ease of implementation, and ability to achieve linear phase responses [132, 148].

Typical radio systems contain multiple FIR filters with fixed parameters. In SDR systems, where multiple wireless communications standards may be supported, the ability to reconfigure a filter's coefficients at run-time helps to use hardware resources more efficiently. These are known as Reconfigurable FIR filters, or RFIR filters.

These RFIR filters are well suited to FPGA architectures due to their parallel structure and inherent reconfigurability, which allows for fast reconfiguration times and operation speeds, suitable for CR applications. In recent years RFIRs have become an active research topic, specifically targeting FPGA-based hardware [149–152].

The design of FIR filters is usually approached by separating the process into two problems: *approximation* and *realisation*. *Approximation* deals with the computation of the filter coefficients, where the response is designed to correspond as closely as possible to an ideal response. The *realisation* problem deals with how best to implement the structure of the filter in hardware. For RFIR filters, the literature is primarily concerned

with the latter. For example, two RFIR filter architectures were developed in [149], based on distributed arithmetic and a Lookup Table (LUT) multiplication scheme. By using configurable LUT primitives they were able to achieve reconfiguration times of less than 100 ns. Or in [153] a *canonic signed digit* coefficient grouping method was used to implement an RFIR, achieving low resource and power usage, and an increase in operation speeds. Further examples from the literature are discussed in [55].

In the works described above and discussed in [55], the *approximation* part of filter design is assumed to have already been carried out, with the calculated filter coefficients typically stored in memory, either on-chip or off-chip. However, on-chip storage reduces the amount of available resources, while off-chip storage can be expensive in both Bill of Materials (BOM) costs and engineering time. For FPGAs, on-chip memory could be in the form of LUTs (distributed memory) or Block Random Access Memory (BRAM). However, this type of storage is not scalable, especially for systems that require long-length filters and arbitrary frequency responses, where storage requirements would become a limiting factor. This signifies that, for a truly dynamic RFIR, filter coefficients must be calculated on-the-fly and at run-time.

Two obvious choices for calculating filter coefficients on-the-fly are (*i*) off-chip embedded or DSP processors and (*ii*) SoCs, such as the PS portion of AMD Zynq devices. However, both approaches require a communications link between processor and FPGA in order to reprogram the RFIR filter, creating a bottleneck, which can increase reconfiguration and development times. Moreover, for devices such as the RFSoC, where the RF-DCs are tightly coupled to the FPGA fabric, it is worth considering the computation of filter coefficients directly on the FPGA.

Figure 2.14 shows a block diagram depicting these three possible methods of programming an RFIR filter, as described below:

- In diagram *a)* an embedded processor is used to compute the filter coefficients and transfer them to shared memory. A memory controller on the FPGA then retrieves the coefficients from memory. A separate IP converts the coefficients into a form valid for the filter architecture, such as adding control signals, and then reconfiguring the RFIR.

- In diagram *b)* an FPGA-based SoC device is depicted, where the filter coefficients are first calculated on the embedded processor, then transferred to the FPGA via an AXI interconnect to the coefficient converter. The converted coefficients are then used to program the RFIR.

- In diagram *c)* there is no requirement for external memory as the filter coefficients are calculated directly on the FPGA and passed to the RFIR without the need to store them in memory first.

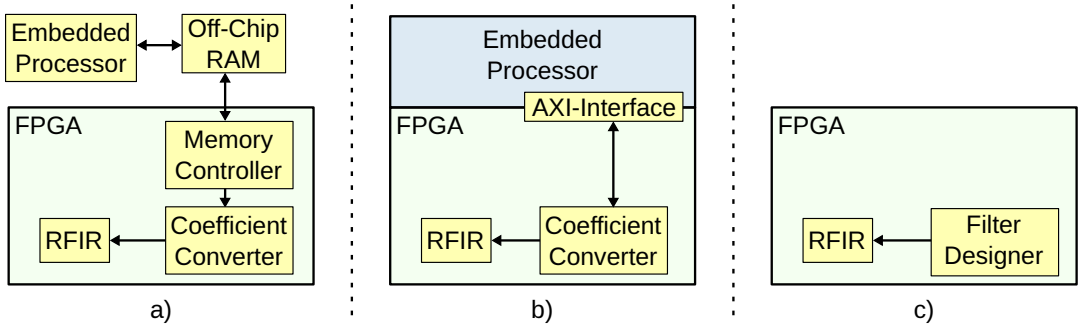

Figure 2.14: Block diagram depicting three methods of reconfiguring an FPGA-based RFIR: a) an embedded processor with off-chip memory, b) an SoC, and c) the FPGA-based filter design method described in Chapter 5.

When comparing these three methods, it is obvious that the architecture shown in diagram *c)* requires fewer components and avoids the need for memory access or communication between off-chip and on-chip components, reducing the overall reconfiguration time.

FPGA-based filter design has some obvious benefits for CR applications, where faster reconfiguration times would improve overall system performance. In devices such as the RFSoC, where the RF-DCs are connected directly to the FPGA fabric, reconfiguration times can be greatly reduced by computing coefficients on the FPGA.

2.4.1 Filter Design Techniques for FPGAs

FIR filters are a practical choice for SDR systems as they do not include a feedback element, making them inherently stable, and they are easily constrained to achieve

linear phase response. If a system has linear phase response it means that all frequency components of a signal shift in time by the same amount, preserving the original waveform shape and preventing distortion [154].

A FIR filter will be guaranteed to achieve linear phase response if its impulse response is even or odd symmetric, or even or odd antisymmetric, around its centre-point. This results in four types (or classes) of FIR filters—typically denoted I-IV—each with their own specific properties. Type II filters will always have 0 amplitude at $\omega = \pi$ (Nyquist) making them unsuitable for high-pass filter responses. Type III and IV filters have a constant phase shift of 90°, which can be used for Hilbert transforms or differentiators [155]. Type I filters are the most universal, allowing for most arbitrary frequency responses and have zero phase shift, and therefore are the type considered in this chapter.

Filter design typically starts by defining an ideal frequency response, specific to a given application. The frequency response, $H_d(\omega)$, of an ideal FIR filter can be written in the form

$$H_d(\omega) = \sum_{n=-\infty}^{\infty} h_d(n) e^{-j\omega n}, \tag{2.7}$$

where $h_d(n)$ are the filter coefficients, and $0 \leq \omega \leq \pi$.

Because Equation 2.7 is both infinite in length and non-causal (i.e. includes negative time samples), the FIR filter can only be realisable if it is truncated to a length N, and time-shifted such that $n \geq 0$; resulting in

$$H(\omega) = \sum_{n=0}^{N-1} h(n) e^{-j\omega n}, \tag{2.8}$$

where $H(\omega)$ and $h(n)$ are the realisable frequency response and filter coefficients, respectively—now only an approximation of the ideal response from Equation 2.7. This approximation results in undesirable effects in the frequency response of the filter; such as ripple in the passband, a decrease in stopband attenuation, and an increase in transition bandwidth; all of which contribute to a reduction of filter quality. Figure 2.15 depicts a typical frequency response with these effects annotated.

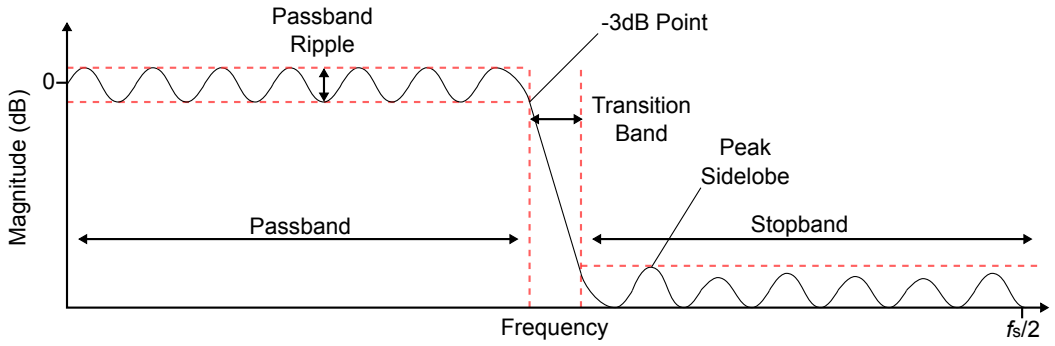

Figure 2.15: Frequency response of a typical low-pass filter with the effects of approximation annotated.

A naïve approach to filter design is to simply increase the filter length until the frequency response meets design specifications. However, this method is impractical due to the large amount of hardware resources required for even the simplest of applications. Various FIR filter design techniques have been developed over the years which attempt to mitigate the effects of approximation, while keeping the filter length small enough to be practical for hardware implementation.

These issues are compounded when considering a filter design technique suitable for FPGAs, where arithmetic is generally restricted to fixed-point, and restrictions on wordlength along the signal path may be necessary in order to avoid excessive bit growth. These restrictions can cause quantisation errors, causing $H(n)$ to move further away from $H_d(n)$, which will manifest in poor performance in the stopband. Moreover, because quantisation errors are accumulative, for applications that require long filter lengths it maybe necessary to increase the wordlength to mitigate these effects. Quantisation errors can also be mitigated by various forms of rounding [156–158]. However, this process can be resource intensive and may not be appropriate for applications where hardware resources are limited.

All these factors must be taken into account when considering filter design techniques for FPGA hardware. Therefore, it is worth considering the conditions such a technique may require. In doing so, three such conditions have been identified:

1. It can produce filters with arbitrary and deterministic frequency responses.

2. It is suitable for implementation on FPGA hardware.

3. The execution time is deterministic at compile-time (i.e. at IP generation).

For conditions 1 and 3, the word "deterministic" means a process will produce the same outcome every time it is executed under the same conditions, ensuring repeatability.

FIR filter design techniques can be separated into two categories: optimal and suboptimal. The rest of this section looks at algorithms in both of these categories and attempts to identify a suitable algorithm able to satisfy all three of the conditions enumerated above.

Optimal Algorithms

Optimal techniques, such as the Parks-McClellan [159] and Least Squares [160] algorithms approach filter design as an optimisation problem, where the error between $H_d(n)$ and $H(n)$ is minimised until an optimal fit is found. Typically, filters designed using optimisation techniques tend to be of high quality and allow for some fine-tuning over parameters; such as the width of the transition band. However, optimal design techniques are inherently unsuited to FPGAs and so will only be briefly discussed here.

The Parks-McClellan algorithm is one of the most well-known and widely used methods of filter design. It uses an iterative approach where it makes an initial "best guess" approximation then makes adjustments to the filter until a solution is found that meets the design specifications [159]. Because of this iterative approach, a solution may require many iterations to converge, or might not converge at all. This makes the process non-deterministic, causing it to fail Condition 3. Additionally, while it is possible to put constraints on the number of iterations to perform, this may result in non-deterministic and low quality filters; causing it to fail Condition 1. Moreover, these constraints only give an upper limit to finding a solution, therefore the exact execution time cannot not be known at compile-time; again failing Condition 3. Finally, the Parks-McClellan algorithm cannot be computed with wordlength restrictions [161], failing Condition 2.

The Least Squares algorithm is another popular filter design method that produces good quality filters, is relatively simple to implement on traditional hardware and, unlike the Parks-McClellan algorithm, can be implemented with fixed-point arithmetic. It avoids the iterative approach to minimising the error between $H_d(n)$ and $H(n)$ by solving a system of linear equations [155]. However, the algorithm is dependent on a pseudo-inverse operation which is inherently difficult to achieve on FPGA hardware due to the division operation (which is both slow and resource intensive); failing Condition 2. Moreover, the size of the matrices required for long-length filters can also be a limiting factor.

Furthermore, there are a number of optimal filter design techniques using algorithms such as integer and linear programming [161], simulated annealing [162], and genetic algorithms [163]. However, these all suffer from some or all of the issues mentioned above, making them unable to meet all three conditions.

Suboptimal Algorithms

The two most common methods of suboptimal filter design are the window and frequency sampling methods. They are suboptimal in the sense that they do not attempt to minimise an error, and do not require recursive or iterative algorithms—resulting in a single solution. They typically have fewer tunable parameters compared to optimal design methods and generally require longer length filters to achieve comparable quality (i.e. performance in terms of signal integrity and distortion) [155].

These methods do not suffer from the same problems inherent in the optimal techniques and, therefore, it is worth looking at them in greater detail to determine their suitability for implementation on FPGA hardware.

The Window Method The window method performs the filter design process entirely in the time domain. An ideal impulse response, $h_d(n)$, representing the desired frequency response, is multiplied with a window function, $w(n)$, of length N, to create a realisable filter, $h(n)$. This can be written as

$$h(n) = h_d(n).w(n), \tag{2.9}$$

where in the case of a low-pass filter, the impulse response is the sinc function, which can be written in the form

$$h_d(n) = \frac{sin(\omega_c n)}{\pi n}, \tag{2.10}$$

where ω_c is the cutoff frequency in radians/sample.

The simplest window to use is the rectangular function, which can be achieved by simply truncating $h_d(n)$ to a given length, N. However, this truncation introduces sharp discontinuities in the sinc function, resulting in poorer performance in the stopband and excessive ripple in the passband, known as the Gibbs Phenomenon [164]. Instead specialised window functions are used which gradually taper the amplitude towards zero, reducing the discontinuities caused by the truncation. This results in an overall improved filter performance, particularly in passband ripple.

The most common types of windows used are the generalised cosine windows such as the Hamming, Hanning, and Blackman. Each window has its own characteristics where there is usually a trade-off between transition bandwidth and stopband attenuation [7]. A comparison of filter performance for each of these windows is shown in Table 2.3.

Table 2.3: Comparison of various window functions, where ω_s is the sampling frequency in radians/sample and N is the filter length. Replicated from [7]

Window	Mainlobe (ω_s/N)	Transition BW (ω_s/N)	Peak Sidelobe (dB)
Rectangular	2	0.9	-13
Hanning	4	3.1	-31
Hamming	4	3.3	-41
Blackman	6	5.5	-57

If the filter length, N, is fixed at compile-time then the window function, $w(n)$, can be precomputed and stored in memory. Furthermore, because the effects of the window on the filter can be predicted, as shown in Table 2.3, this is able to satisfy Condition 1 by producing deterministic frequency responses. However, while the window method is simple in concept, it does pose some problems for implementation on FPGA hardware.

If the cutoff frequency, ω_c, is to be reconfigurable at run-time then this introduces the requirement of a division operation, as shown in Equation 2.10, which can be challenging to perform on FPGAs efficiently, causing it to fail Condition 2. Moreover, while the impulse response of a low-pass filter is related to a simple sinc function, other frequency responses have more complex equations in the time domain, indicating that this method, in fact, fails Condition 1.

The Frequency Sampling Method Similar to the optimal methods described in the previous section, the frequency sampling method starts the filter design process by describing the filter response in the frequency domain. The ideal response, $H_d(\omega)$, is sampled at equally spaced frequencies, $\omega_k = \frac{2\pi k}{N}$, where $k = 0, 1, \ldots, N-1$. This can be written in the form

$$H_d(\omega_k) = \sum_{n=0}^{N-1} h_d(n) e^{-j2\pi kn/N}, \tag{2.11}$$

and is equivalent to the Discrete Fourier Transform (DFT), allowing the filter coefficients (i.e. the impulse response) to be retrieved using the Inverse Discrete Fourier Transform (IDFT or Inverse DFT) [160].

As with the window method, the frequency sampling method also poses problems for implementation on FPGAs. As the response is only defined at the sample points, the gaps in between the samples are effectively interpolated. This can result in undesirable effects at the points where large changes in amplitude occur, typically around the transition band. This can be managed by introducing tapered transition band samples, but the algorithm to calculate these uses an optimisation technique, which is unsuitable for implementation on FPGAs [165]. Furthermore, for long-length filters the DFT is impractical to implement and therefore the Fast Fourier Transform (FFT) algorithm must be used. However, while the FFT is a well established algorithm able to be implemented on FPGAs, it is generally fixed to power-of-two values, reducing the ability to produce odd-symmetric Type I filters, failing Condition 1.

Table 2.4 summarises the comparison of the different filter design techniques against the three conditions defined above.

Table 2.4: Comparison of filter design techniques against the three conditions defined in Section 2.4.

Method	Arbitrary Response	FPGA-friendly	Deterministic
Parks-McClellan	Yes	No	No
Least Squares	Yes	No	Yes
Window	Yes	Partially	Yes
Frequency Sampling	No	Yes	Yes

In summary, this survey indicates that existing filter design techniques fail to meet the combined requirements of arbitrary frequency response, compile-time determinism, and FPGA suitability. As such, they remain unsuitable for the constraints of highly dynamic CR systems on RFSoC devices. This research gap motivates the development of a filter design technique which satisfies these requirements.

2.5 Chapter Summary

This chapter has provided a survey of the background and related work surrounding SDR and CR systems. First, the RFSoC was established as an ideal platform for CR, noting its ability to overcome the latency and bandwidth limitations of legacy SDR platforms. However, despite these capabilities, three key research gaps were identified.

The first gap is methodological. While established established tools, such as PYNQ and Model Composer offer productivity gains over the conventional toolflows, they remain fragmented and do not provide a unified design flow conducive to rapid prototyping. The second and third gaps are technical. Existing frequency planning tools are restricted to offline, interactive use, and current FIR filter design methods are unsuited for deterministic, run-time reconfiguration required by a highly dynamic spectral environment.

To address these gaps, the remainder of this thesis is structured around three primary research questions:

- How can existing design tools be extended to create a unified design methodology for the RFSoC that promotes productivity?

- How can a frequency planning tool be designed to operate in real-time within the latency constraints of a CR system?
- How can a filter design method be developed that allows for deterministic, low-latency reconfiguration of FIR filters on FPGA fabric?

These questions are explored in this thesis through the following core contributions:

- A full-stack design methodology for the RFSoC that integrates and extends established tools to enable rapid iteration and experimentation, is presented in Chapter 3.
- A scriptable frequency planning tool designed to operate within the cognitive cycle, capable of dynamically identifying and avoiding spurious components caused by the RF data conversion process, is presented in Chapter 4.
- A filter design technique for calculating and updating arbitrary filter coefficients directly on the FPGA fabric, providing deterministic execution times and low-latency reconfiguration, is presented in Chapter 5.

The following chapters detail these contributions, establishing a set of tools and methodologies that address the technical challenges of implementing real-time, reconfigurable CR systems on RFSoC platforms.

Chapter 3

SDR System Design on RFSoC Devices

3.1 Chapter Overview and Contributions

In this chapter a design methodology is presented for the development of SDR systems on the RFSoC, with a focus on productivity. Here, productivity is defined in terms of both lines of code required to interface with hardware, and development time.

To demonstrate this methodology, a fully functional transceiver design is developed for the RFSoC. This demonstrator provides real-time data inspection and visualisation, as well as user reconfiguration through software; all performed on a single device. Model Composer is used to develop the custom HDL IP cores for the PL, with PYNQ as the target embedded framework. This design methodology is then evaluated in terms of productivity, community impact, and system performance.

The work in this chapter is based on an original contribution published in the IEEE Access journal [48], which was one of the first non-AMD publications on RFSoC devices, and the first to evaluate the PYNQ framework on the platform.

The contributions of this chapter are as follows:

- A rapid-prototyping design methodology using Model Composer and PYNQ.

- The development of open-source RFSoC drivers for PYNQ to lower the barrier of entry.
- A Quadrature Phase-Shift Keying (QPSK) transceiver reference design to demonstrate the methodology.

As [48] contained the work of multiple authors, it is important to highlight this author's own contributions to the project, which are as follows:

- Adding AXI functionality to the transmit and receive IPs within Model Composer. In particular, AXI-Lite for user control and AXI-Stream for data streaming; discussed in Section 3.4.1.
- Converting the transmit and receive models to AXI IPs for use within Vivado, discussed in Section 3.4.1.
- Building the overall RFSoC PL design, within Vivado, around the IP cores developed in Model Composer, including clocking, data inspection, and additional filtering, discussed in Section 3.4.2.
- Development of basic testing software before the integration of PYNQ was completed.
- Retrieval of results.
- Contribution of writing and reviewing of the paper.

3.2 Methodology and Design Strategy

As established in Chapter 2, the conventional RFSoC design flow is not well-suited for productivity and rapid prototyping. While a survey of the literature revealed a number of academic attempts to address this, these tools suffer from lack of maintenance and support once the initial research had concluded. Therefore, rather than developing a new tool, this work instead uses and extends existing and established tools to ensure long-term support and reproducibility.

As such, Model Composer was selected for the design and development of IP cores for use on the FPGA. In particular, this tool was chosen for its ability to accelerate algorithm development and validation by using Simulink's built-in signal generators and spectrum analysers. This allowed signal processing pipelines to be quickly iterated and validated in simulation before committing to the time-consuming bitstream generation process.

For the PS runtime, PYNQ was selected due to its well-established effectiveness in abstraction and productivity. However, at the time of writing the original paper [48], the PYNQ framework did not support the RFSoC architecture. Therefore, a core contribution of this work is the extension of the PYNQ framework to support the device's RF data converters and clocking infrastructure. The proposed flow is as follows:

- **IP Design and Validation (Model Composer)**: Radio and signal processing functionality are designed and validated against Simulink sources, with explicit definitions for hardware control and data introspection.
- **System Integration (Vivado)**: The validated IP cores are exported to Vivado, where it is integrated with DMA cores to enable data streaming between the PL and PS.
- **Run-time Control (PYNQ)**: The bitstream is loaded onto the RFSoC, where Python drivers provide run-time control and visualisation of the system.

3.3 System Architecture Overview

To evaluate the methodology proposed in Section 3.2, a full radio transceiver design was developed. The architecture of this system is described in this section, with the design and implementation details of each component described in the following sections. The purpose of this demonstrator was to act as a proof-of-concept reference design, rather than to create a high-performance communications system. Therefore, a simple QPSK modulation scheme is used with a low data rate of 1 kbps. The low data rate allowed for real-time visualisation in the initial stages of development, providing additional debugging capabilities.

Figure 3.1 depicts a functional block diagram of the demonstrator system, split between the PS and PL portions of the device. On the PL, the FPGA is used to implement the majority of the DSP algorithms of the transceiver, while the RF-DCs are used for additional interpolation and decimation, as well as complex mixing via the integrated NCO. On the PS, the PYNQ framework is used for control of the transmit and receive architecture on the FPGA, the RF-DCs, as well as visualisation of captured data throughout the signal path.

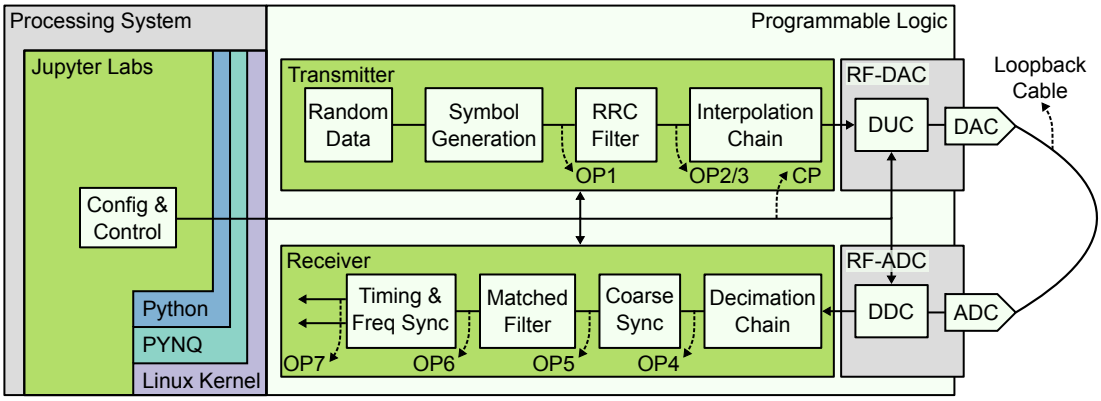

Figure 3.1: Functional block diagram of the SDR demonstrator.

3.3.1 PL System Overview

The FPGA fabric on the PL contains various custom IPs that make up the transmitter and receiver, shown on the left side of the PL block in Figure 3.1. In the transmitter, random data is generated, converted to I/Q symbols, pulse-shaped with a Root-Raised-Cosine (RRC) filter, then passed through a multi-stage interpolation chain before reaching the RF-DAC. The RF-DAC further interpolates the signal by a factor of 8 (increasing the sample rate to 1024 Msps), then it passes through the complex RF mixer to be modulated by a carrier. The modulated signal is then converted from the digital domain into the analogue domain.

The receiver architecture performs a reciprocal chain of operations where the RF-ADC converts the analogue signal into a digital representation, demodulates it with a carrier of equal frequency, and then decimates the signal by a factor of 8 before passing

it to the FPGA. The FPGA signal path performs additional decimation (decreasing the sample rate from 128 Msps to 4 ksps), followed by coarse synchronisation, a matched filter, and, finally, timing and frequency synchronisation that allows the QPSK symbols to be retrieved.

Throughout the signal path of the transmit and receive architectures, there are various points at which the signal can be visualised, or parameters controlled on the PS. These are named Observation Points (OPs) and Control Points (CPs), respectively. For example, after the pulse-shaping filter in the transmit path, there are two OPs that allow inspection of either time domain or frequency domain data, while the CP after the interpolation chain allows the gain of the signal to be configured before arriving at the RF-DAC.

3.3.2 PS System Overview

As shown on the left-hand side of Figure 3.1, the PS hosts the PYNQ framework. PYNQ runs software that allows the control and configuration of the PL radio architecture, as well as user interactivity and data visualisation. As discussed in Chapter 2, the PYNQ framework consists of an Ubuntu-based Linux OS, PYNQ software library, and a Jupyter web-server, which allows users to write and run code on the board via a web browser-based IDE running on a client computer.

The PYNQ framework is particularly useful for the initial development stages of the design process. For example, the browser-based IDE allows users to write and run embedded code directly on the device with no additional software required on the client computer, except for a web browser. Moreover, the use of the *Overlay* class allows PYNQ to automatically assign drivers to known IPs, as well as to generate generic drivers for custom IPs. The inclusion of Python also enables the use of established software libraries, such as *Plotly* for visualisation, and *ipywidgets* for user control.

It is important to note that the visualisation and user control are supplemental to the radio architecture, meaning that these components of the software can be removed after the development and prototyping stages if required. With that said, these introspection components can also be useful after deployment as they can be further used as remote

debugging tools for devices in the field. For example, after the deployment of a radio it may be necessary to check the quality of received signals periodically and update parameters depending on these observations. The use of the Jupyter web-server allows users to remotely access the device from anywhere in the world and reconfigure the device without requiring physical access.

3.4 Hardware Design

This section describes the design and development of the PL system. As the finer details of the PL design are described in [48], and the source code is publicly available [166], only the elements relevant to this thesis are replicated here. In particular, this section outlines the use of Model Composer for IP design and simulation, how it can be used effectively for reconfigurable SDR designs on the RFSoC; the use of Vivado's IP Integrator (IP Integrator (IPI)) to build a full PL system; and how the data capture and visualisation infrastructure was designed.

3.4.1 Development of Radio IP Cores

Two approaches were taken when designing the transmit and receive cores in order to investigate the advantages and disadvantages of monolithic and heterogeneous IP design, and how it affects software development, discussed later in Section 3.5. The transmitter architecture was designed as the former, while the receiver architecture was designed as the latter. Model Composer was used for the development of each architecture, allowing for the generation of HDL IP cores to be integrated into a complete FPGA design within the Vivado IDE.

The use of Model Composer enabled continuous testing during development to verify the correct operation of each IP. While this is possible in other methods of IP development, such as writing Register Transfer Level (RTL) code and testbenches within the Vivado IDE, the ability to use Simulink's built-in signal source and visualisation tools to provide stimulus for the IPs, and get immediate feedback with the use of oscilloscopes and spectrum analysers, makes this much simpler in comparison; helping to

reduce development time. Moreover, as Model Composer is built into the MATLAB and Simulink ecosystem, it allows for the initial IP design to be first developed and tested using built-in functions and floating point arithmetic, then incrementally converting each component of the IP to make it HDL-compatible; all within the same software package.

The transmitter was developed in a monolithic manner, containing all components of the transmit logic within a single IP core. The transmitter generates random data, performed with the use of a Linear Feedback Shift Register (LFSR) at a rate of 1 kbps. The symbols are then Gray encoded [167] and separated into I/Q samples at a rate of 500 symbols/s. The I and Q samples are then passed separately through an interpolation chain which consists of an RRC filter, interpolating by 4; and a series of half-bands and Cascaded Integrator-Comb (CIC) filters, interpolating by a combined factor of 12,800; bringing the sample rate up to 25.6 Msps. The I and Q samples are then concatenated at the output of the IP to conform with the input requirements of the RF-DC IP within Vivado [168]. Three OPs were used for the transmitter architecture, one after the generation of the I/Q symbols and two after the RRC pulse-shaping filter. One of the RRC OPs was connected to a hardware FFT IP, allowing frequency domain data to be passed to the PS. Additionally, a single CP was used that controlled the gain at the output of the interpolation chain. The CP was set up for use with AXI-Lite, enabling the control of the gain from the PS using MMIO.

The receiver was developed in a heterogeneous manner, separating the various stages of the receive logic as individual IPs. The receiver first decimates the signal from 25.6 Msps to 4 kbps through a reciprocal decimation chain. An FFT-based coarse synchronisation is then performed followed by frequency correction with the use of a matched filter, interpolating by a factor of 4. Finally, timing and frequency synchronisation is performed using the method similar to that described in [169]. Each of these stages were generated as separate HDL IPs, with an OP connected at each of their outputs. In contrast to the transmitter, no hardware FFT was used for any of the OPs on the receiver, opting to use software-based FFTs instead. This was done

to determine any differences in performance between hardware and software FFTs for visualisation purposes, as discussed in Section 3.6.

3.4.2 Vivado System Design

Vivado IPI was used to create the PL design around the IP cores developed in Model Composer. The transmitter and receiver IP cores, as well as any supporting IPs, were partitioned into separate hierarchical subsystems in order to make the software design within the PYNQ framework more efficient, as discussed in Section 3.5.

In the top-level design, the Zynq UltraScale+ IP was configured with two High Performance (High Performance (HP)) slave ports, one each for the transmit and receive logic, to allow for data streaming between the PS and PL with the use of the shared PS Dynamic Random Access Memory (DRAM). A single HP master port was used to serve the various AXI-Lite connections used within the design. The inputs and outputs of the receiver and transmitter hierarchies were connected to the RF-DC IP respectively, which was configured with a sample rate of 1024 Msps, 8x interpolation and decimation, and the NCO was set to fine-mixing mode. Three clock domains were used throughout the design: a single 100 MHz PL clock, which served all AXI-Lite connections, and two 409.6 MHz off-chip reference clocks for the RF-DACs and RF-ADCs. The reference clocks were further divided to 128 MHz and 64 MHz to serve the RF-DAC and RF-ADC respectively. Figure 3.2 shows a high-level block diagram of the top-level Vivado IPI design.

As the transmitter IP was monolithic, this simplified the design of the transmitter hierarchy. Each of the OP ports were connected to Data Inspection Module (DIM) IPs, discussed in Section 3.4.3, which in turn were connected to separate AXI DMA IPs. This allowed for the data to be streamed from the PL to the PS via shared memory. The CP was connected to an AXI-Lite interface for the control of the IP through software on the PS. The main output of the transmitter IP was connected to an additional 5x interpolation stage, facilitated by an AMD LogiCORE FIR Compiler [170], which increased the data rate from 25.6 Msps to 128 Msps. This was necessary due to limitations of the Model Composer software, which restricts the amount of rate change

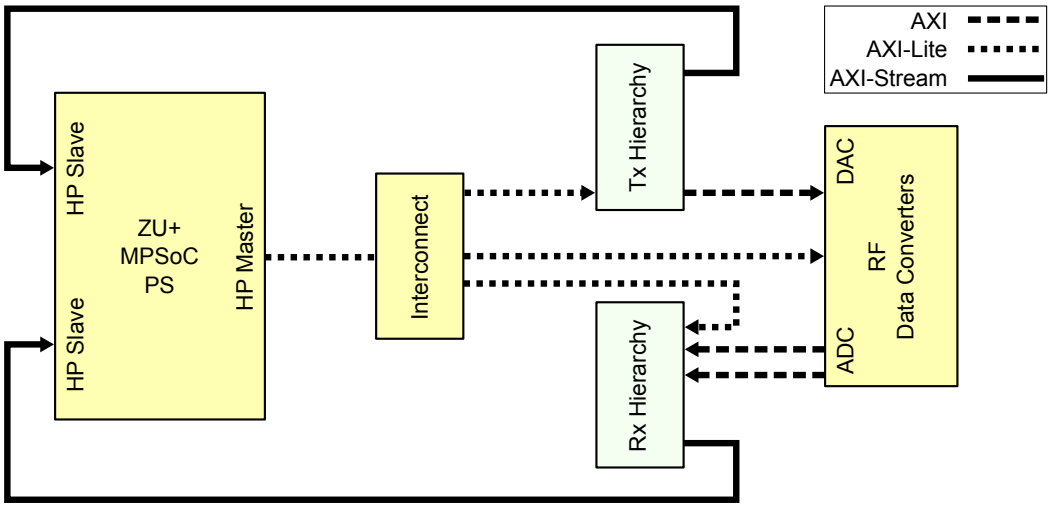

Figure 3.2: High-level block diagram of the Vivado IPI design, where *ZU+ MPSoC PS* is the Zynq UltraScale+ Processing System IP block.

allowed within a single design, due to its use of clock enables to change the sample rate internally. The use of the additional 8x interpolation within the DUC of the RF-DAC upsampled the signal to the appropriate value of 1024 Msps. A high-level block diagram of the transmitter hierarchy is shown in Figure 3.3.

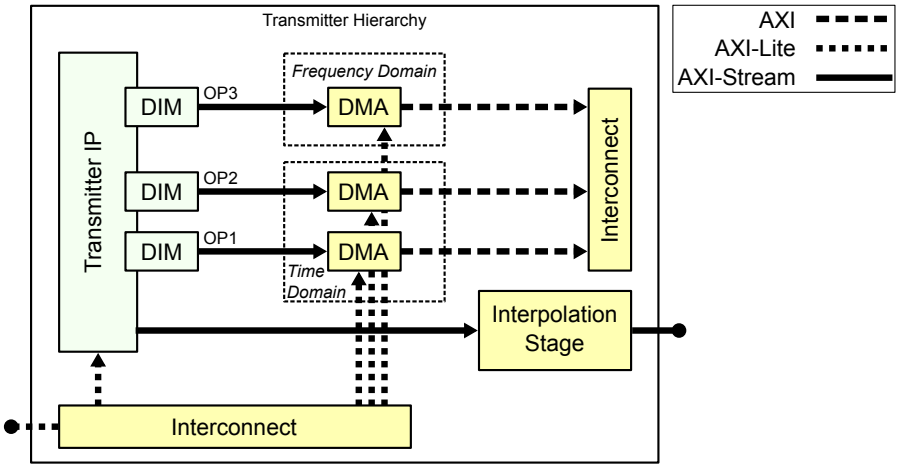

Figure 3.3: High-level block diagram of the transmitter Vivado IPI design.

The modular design approach for the receiver IP made the receiver hierarchy slightly more complex due to the additional connections required between the IPs. The signal is first passed through a reciprocal 5x decimation stage to convert the signal from the

RF-ADC from 128 Msps to the 25.6 Msps signal required by the first stage of the receiver IP core. As most of the IPs have different sample rate and clocking needs, it would typically be required to use more clocking resources to serve them. However, as discussed above, it was possible to exploit Model Composer's use of clock enables to change the sample rate internally, allowing each IP to be served by the same frequency clock. As with the transmitter hierarchy, the OPs of the separate receiver IPs were connected to AXI DMA IPs via separate DIMs. A high-level block diagram of the receiver hierarchy can be seen in Figure 3.4.

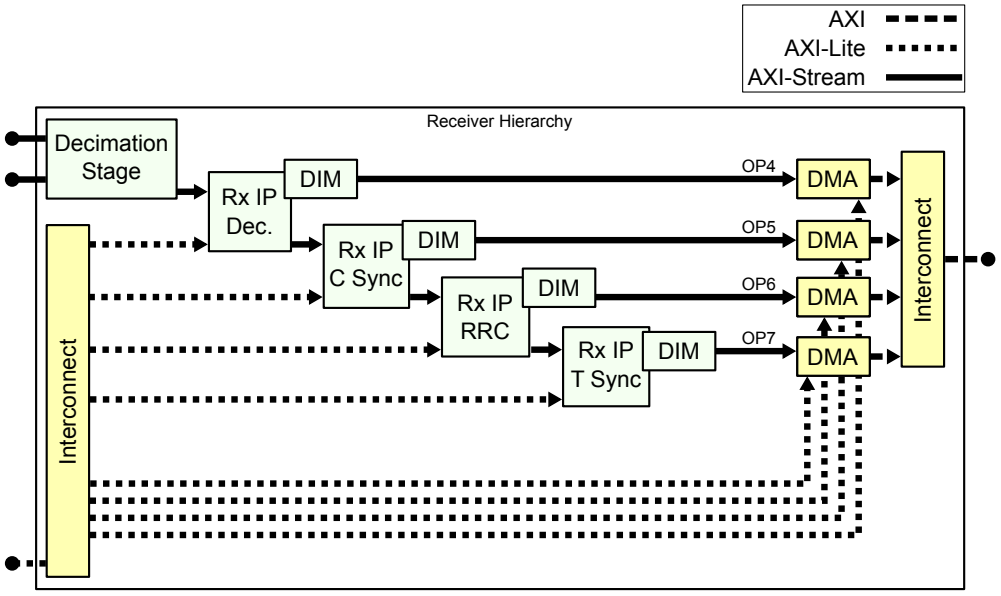

Figure 3.4: High-level block diagram of the receiver Vivado IPI design.

3.4.3 Data Capture and Visualisation

As discussed in Section 3.3, various points throughout the signal paths of the transmit and receive IPs are tapped out for data inspection using the OPs. While in [48] this was done primarily for demonstration purposes, it is also useful during development for debugging, verification, and analysis of the data. To enable data capture for visualisation, a custom packetiser IP was created within Model Composer, named the Data Inspection Module, that worked in tandem with the AMD-supplied AXI DMA IP. This allowed data packets to be moved from the PL to the PS, via shared DRAM, for visualisation

within PYNQ and the Jupyter environment. A high-level block diagram of the DIM is shown in Figure 3.5.

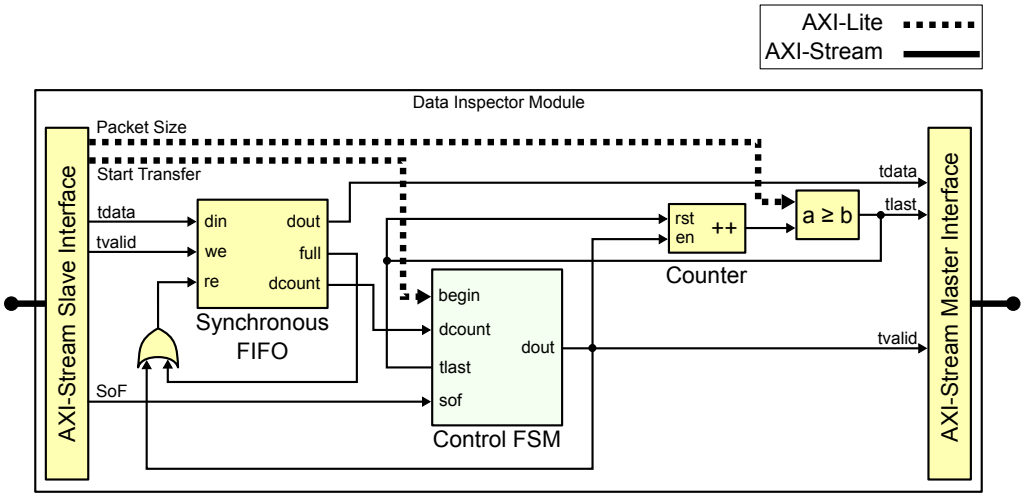

Figure 3.5: Block diagram of the DIM.

As each OP requires its own DIM, it is necessary that the logic controlling it uses as few resources as possible, therefore the DIM was designed with efficiency in mind. The DIM contains a First In, First Out (FIFO) that buffers valid data until the DMA signals it is ready to perform a transfer to the DRAM, with a packet size that can be configured from the PS with the use of an AXI-Lite register, along with an enable signal. The FIFO continuously updates the buffer with new data, ejecting any old data that is not required, until a ready signal is received. If there is not enough data when the ready signal is flagged then the FIFO waits until the buffer has been filled.

The DMA IP uses the AXI-Stream protocol, which has a strict dependence on the *tvalid* and *tlast* control signals and, therefore, the DIM must provide these at the correct time. A counter keeps track of the number of samples that have been passed to the DMA until the packet size has been reached, with a relational logic block providing the *tlast* signal at the end of the packet. A Finite State Machine (FSM) keeps track of the data passing through the DIM, providing the *tvalid* signal only when a valid data transfer is occurring. Together with the *tdata* signal, containing the valid data, these signals are combined into a single Master AXI-Stream interface, which is connected to

the AXI-Stream Slave port of the DMA. An additional start of frame signal is used by the DIM for use with hardware FFTs, allowing correct packetisation of the FFT frames.

3.5 Software Design

To provide support for the RFSoC, the software implementation required the development of two main components that map to the two distinct parts of the hardware: the "hard" IP comprising the RF data converters and clocking infrastructure, and the "soft" IP of the transceiver design developed in Model Composer.

These two components require different approaches. For the "hard" IP, it was necessary to interact with the C-level drivers directly. This required wrapping these drivers using Python's C Foreign Function Interface (CFFI) module to expose them to the PYNQ environment. For the "soft" IP, the use of the standard AXI interfaces in the CPs and OPs enabled the use of PYNQ's built-in MMIO and DMA drivers.

3.5.1 Python Drivers for RFSoC Hardware

It is worth reiterating that, at the time of development, there was no PYNQ support for the RFSoC available. Therefore, one of the main tasks of the project was to add this support to the PYNQ framework, coinciding with the release of the PYNQ v2.5 version of the software. This was achieved by working in collaboration with engineers at AMD to develop the software necessary to integrate PYNQ into the RFSoC platform, including the drivers required to operate its RF components. Since the publication of [48], the RFSoC has become a well-established platform within the PYNQ ecosystem, and several updates have been made to the drivers that interact with the RFSoC hardware since the work described in the following sections was undertaken. With this in mind, some information described in this section will not entirely reflect the current state of PYNQ support for the RFSoC, although much of the code remains unchanged.

Two main components of the RFSoC hardware are required to operate the transceiver: the RF data converters, and the clocking infrastructure that drives them. While the

drivers developed for these components ultimately use Python within the PYNQ environment, it was necessary to develop much of the codebase within the C language. As the clocks that drive the RF data converters are off-chip, and connected to the RFSoC via an Inter-Integrated Circuit (I2C) connection, it was convenient to reuse well-established Linux C drivers that support the I2C protocol, and use the clock datasheets to build a native C driver based on the register map. Conversely, as the RF data converters do not have a fully documented register map, it was necessary to use the vendor baremetal C driver as the base.

Clocking Infrastructure

The RFSoC development board features a set of clock synthesizers that drive the RF data converters: a Texas Instruments (TI) LMK04208, which is used as a reference clock, and three TI LMX2594s that provide the sample clocks to the data converter tiles. I2C is used to program the clocks with register values to change the output frequencies. The register values come in the form of text files generated from the *TICS PRO* software, provided by TI [171]. A simple driver was developed in C that could read the register values from the provided text files and program the different clocks via the I2C interface. A thin wrapper was then developed using the built-in CFFI module, which provides a means of wrapping functions within C code and calling them from within Python. It is worth noting that the PYNQ v3.0.1 driver for the RFSoC clocking infrastructure removed the requirement of the C-based driver and CFFI, providing a native Python driver capable of equivalent functionality [172].

In its current state, the driver is reliant on the use of the text files generated by *TICS PRO* to provide the register values required to reconfigure the clocks. As the software only supports the Windows operating system, and lacks an Application Programming Interface (API), it is not possible to generate these files and program the clocks on-the-fly, meaning that all clock frequencies must be known before operation, so that the text files can be generated in advance. Since the sample rate of the RF-DCs must be related to the frequency of the sample clock, this severely restricts the ability to reconfigure the data converters with arbitrary sample rates. However, as the register

maps for both types of clock synthesizers are fully documented, it would be possible to modify the driver to directly interact with the NCOs and clock dividers within the devices, allowing any sample clock frequency to be generated for the RF-DCs, providing increased flexibility for highly reconfigurable SDR applications.

RF Data Converters

Unlike the clock synthesizers that make up the clocking infrastructure, the RF data converters on the RFSoC do not have a fully documented register map. Therefore, the RF-DCs must be configured with the use of the C driver provided by AMD. The C driver provides a high degree of functionality to the user, allowing almost all of the RF-DC features to be controlled in software. While it would be possible to implement a direct translation of the functions available to Python using CFFI, it was found that a more object-oriented, Pythonic approach was possible, allowing much of the functionality to be abstracted away, providing a simpler interface for the user to configure and reconfigure the data converters.

While the C driver provides a flat list of approximately 50 functions, there is a clear hierarchy to the RF-DCs (made up of blocks within tiles) that can be exploited to create an object-oriented hierarchical Python driver. Furthermore, within Python it is also possible to make a distinction between *attributes*, which get or set values; and *methods*, which perform some action. A Unified Modelling Language (UML) diagram depicting this hierarchical approach can be found in Appendix B.2.

This method of abstraction helps to greatly reduce the amount of required code compared to the baremetal C driver. For example, Listing 3.1 shows the code necessary to update the mixer frequency of an RF-ADC using the C driver. In contrast, Listing 3.2 demonstrates the same operation using the Python abstraction.

As can be seen from these two examples, the Python code not only reduces the number of lines required, but it is significantly easier to read and understand. In particular, the C driver function call to set the mixer settings requires a pointer to an ADC tile instance and a set of index parameters, while the Python driver simply references the specific tile and ADC via the hierarchy, and sets the frequency by accessing

```
int Status;
u16 Tile = 0;
u16 Block = 0;
XRFdc_Mixer_Settings MixerSettings = {0};
XRFdc *RFdcInstPtr = ...
...
Status = XRFdc_GetMixerSettings(RFdcInstPtr,
XRFDC_ADC_TILE, Tile, Block, *MixerSettings);
if (Status != XRFDC_SUCCESS) {
  return XRFDC_FAILURE;
}

MixerSettings.Freq = 1600;

Status = XRFdc_SetMixerSettings(RFdcInstPtr,
XRFDC_ADC_TILE, Tile, Block, &MixerSettings);
if (Status != XRFDC_SUCCESS) {
  return XRFDC_FAILURE;
}
```

Listing 3.1: Example of updating the frequency of an RF-ADC using the C driver.

```
adc_block = ol.rfdc.adc_tiles[0].blocks[0]
adc_block.MixerSettings['Freq'] = 1600
```

Listing 3.2: Example of updating the frequency of an RF-ADC using the Python driver.

that parameter within the `MixerSettings` dictionary. In fact, it is possible to perform this operation in a single line, albeit resulting in less readable code.

3.5.2 Radio Transceiver Drivers

Unlike the "hard" IP drivers described in the previous section, the drivers for the custom transceiver IPs were implemented entirely in Python. This is made possible by directly mapping Python classes to the interfaces defined in Model Composer.

Parameter configuration is handled via MMIO. In the hardware design, CPs, such as signal gain, are assigned specific address offsets on the AXI-Lite bus. The custom Python driver inherits from PYNQ's `DefaultIP` class to bind to the IP core, mapping user-friendly property names to these physical addresses. For example, in the transmitter IP, the output gain register is mapped to offset `0x2C`. When the user updates the `tx.gain`

property, the driver transparently performs a 32-bit write to the corresponding register. This direct mapping eliminates the need for kernel-level driver development, allowing for immediate run-time reconfiguration of the FPGA logic directly from the Python environment.

Live signal introspection is achieved using DMA. The OPs in the hardware are connected to AXI-Stream interfaces via the custom DIM IP. The Python driver manages the transfer of this data into the processor's memory using PYNQ's `DMA` class. To capture a signal, the receiver driver allocates a contiguous data buffer using PYNQ's `allocate` module, and passes the pointer to the DMA. The hardware then streams the data directly to this buffer. Once this transfer is complete, the data is available as a native NumPy array, allowing for immediate processing and visualisation within Python without any additional copying.

As discussed in Section 3.4.1, the transmit and receive IPs created in Model Composer were developed using two different design strategies. The transmitter was designed as a monolithic block, encompassing all of the logic within a single IP, while the receiver was designed heterogeneously, separating the functionality into discrete IPs, interfaced within the Vivado IPI environment. This had a direct effect on how the drivers for each were structured.

When designing drivers for custom IPs, PYNQ allows the user to bind a class to an IP core, or hierarchy of IP cores. For this reason it is customary to write a single driver for each IP. Therefore, as the transmitter IP is monolithic, this naturally leads to the design of a driver that is contained within a single class. This method was found to result in questionable design practices, where a significant portion of the code is copied and pasted (specifically the code for each OP). While it is possible to improve the code by further abstraction, it is nonetheless interesting that this method was found to be "the path of least resistance" for monolithic IPs.

In contrast, as the receiver IP is split into multiple IPs, each with its own OP, the commonality between each IP is quickly identified. This led to the design of a generic OP class that each IP can inherit from, allowing a standard set of methods to be shared between the IP classes, with a smaller subset of methods added to the child classes for

any unique functionality the IP requires (e.g., OPs that use a hardware FFT). As such, it was found that this was the preferred approach to IP design for the methodology.

3.5.3 Data Inspection and Visualisation

Another goal of the transceiver design was to allow users to control the hardware on the FPGA and visualise the data for debugging and testing, all on the RFSoC itself. To achieve this, front-end software was developed that allowed the control of the RF-DCs and the various CPs of the transceiver, as well introspection of the OPs along the signal path within the PYNQ framework, without the need for expensive external hardware, such as spectrum analysers and oscilloscopes. Three main open-source software packages were used: *JupyterLab*, which acted as the main development and interactive environment; and the Python packages *ipywidgets* and *Plotly*, which provided interactive graphical widgets and plotting functionality, respectively.

JupyterLab is a development environment which runs as a web server on all SoC devices running PYNQ. It is structured as a client/server system, which allows designers to write and execute code on the RFSoC via an Ethernet connection between the board (the server) and a host computer (the client), all within a standard web browser. Users can write code within a "notebook" that mixes markdown formatted text, Python code blocks, and the output of any code run within it [173].

As PYNQ allows the use of standard Python code to be run alongside embedded code, it is possible to incorporate Python's extensive library of open-source modules for use within the transceiver software. The *ipywidgets* module enables interactive graphical widgets to be used within the Jupyter environment, which can control various elements of the transceiver with the use of simple callbacks to the drivers. For example, slider controls were created for the gain and centre frequency of the transmitter, allowing users to change these settings on-the-fly. Using this method it is possible to create widgets for any reconfigurable parameter that exists within the RF-DC or custom IP drivers, enabling full control of the system interactively.

Similarly, the *Plotly* module provides rich graphical plotting functionality, allowing users to visually inspect the live data from the OPs, within the Jupyter environment [174].

The use of *Plotly* enabled the transceiver software to accommodate visualisations for time domain, frequency domain, *and* constellation plots; providing an extensive introspection of all data along the signal path. Another benefit of using *Plotly* with *JupyterLab* is the split between the server and the client. As the most resource intensive part of plotting live data is drawing the images on a screen, the use of Jupyter allows this part to be performed on the client side (i.e., the computer), with the server side only required to generate a description of the plotting data which, in the case of *Plotly*, is done with the JavaScript Object Notation (JSON) format. This significantly reduces the computational overhead for the PS, which allowed frame rates of the live data to reach around 20 Frames Per Second (FPS)—more than enough for demonstrative and debugging purposes.

Using this combination of *JupyterLab*, *Plotly*, and *ipywidgets*, an interactive GUI was developed that begins to approach that of the dashboards available for expensive commercial platforms. A screenshot of the GUI developed for the Jupyter environment can be seen in Appendix B.1.

3.6 Results

In this section both qualitative and quantitative results are given that demonstrate the effectiveness of the design methodology presented in this chapter, in terms of productivity and community impact.

Since much of the functionality traditionally realised with external components, such as spectrum analysers and oscilloscopes, is performed directly on the PS, the performance of the software is particularly important in determining the validity of such a design methodology. Moreover, as an interpreted language is used to write the codebase, as opposed to a compiled system-level language, such as C or C++, the performance of Python as an embedded language is also of significant interest.

3.6.1 Community Impact

The transceiver design developed during this work was released as an open-source project with a permissive licence and [48] was published explaining the design methodology and implementation details. This proved to have a positive influence on the research community, spawning a number of additional publications and open-source projects which were motivated, either directly or indirectly, by this work [175–179].

For example, in [175] the authors implemented an oversampled polyphase filter bank on the RFSoC, also releasing the work as an open-source project. The paper notes the difficulties of developing designs that utilise all three components of the RFSoC (PS, PL, and RF-DCs), and concludes that the use of HLS to design the IPs, and PYNQ to develop the PS software, enables non-specialised FPGA engineers to develop full-stack RFSoC designs. Similarly, the authors of [176] implemented an on-chip spectrum analyser on the RFSoC, using HDL Coder within Simulink to develop the HDL IPs and PYNQ as the software framework, following a similar design methodology as this work. In [177], the authors develop a 5G New Radio transceiver for millimetre wave frequencies, using the RFSoC for IF and the PYNQ framework for the PS software, noting how PYNQ is useful for enabling on-chip visualisation of the captured signals on the RFSoC. Finally, in [178], the wideband capabilities of the RFSoC were used to develop an 1.7 GHz bandwidth chirp synthesizer, using PYNQ to generate the chirps on the PS, transferring the data to the PL using PYNQ's DMA driver.

This work has also had influence outside academic circles, being featured in a number of hobbyist blogs [180–182]. Also, anecdotally, it has been used to train engineers on the RFSoC and been the basis for at least one commercial product prototype.

3.6.2 Productivity Analysis

With regards to lines of code, a comparison between Listing 3.1 and Listing 3.2 demonstrates that PYNQ can reduce the amount of code required to update the frequency of an RF-ADC by a factor of nearly ten. Similar results are observed when comparing the example RFSoC code in [183] with the equivalent PYNQ implementation.

While a full quantitative productivity analysis was beyond the scope of this work, the benefits of the design methodology presented in this chapter are clear. In the conventional toolflow, the development of a transceiver is a multi-stage process. Verifying the logic requires the development of custom C/C++ drivers to interface with the FPGA hardware, followed by the compilation of a PetaLinux distribution to manage the runtime environment. Furthermore, signal introspection typically requires external hardware, such as logic analysers, or the development of bespoke GUI applications.

In contrast, the design methodology presented in this chapter removes this overhead. The use of Model Composer allows for bit-accurate verification of the transceiver logic within Simulink prior to synthesis. Following bitstream generation, the PYNQ framework removes the requirement for bare-metal driver development and custom OS cross-compilation. Furthermore, the signal introspection was achieved using standard Python libraries directly on the target device, removing the need to create a custom solution. This demonstrates that the methodology is much simpler and more productive compared to the conventional flow.

3.6.3 Software Performance

Since most of the metrics discussed in this section rely on understanding of the layout of the hardware with regards to the various OPs, as shown in Figure 3.1, it is worth detailing the contents of each, enumerated below:

- OP1: time domain output of the transmitter QPSK symbol generation stage.
- OP2: time domain output of the transmitter RRC filter stage.
- OP3: frequency domain output of the transmitter RRC filter stage, using a hardware accelerated FFT on the FPGA.
- OP4: time domain output of the receiver decimation filter chain stage.
- OP5: time domain output of the receiver coarse synchronisation stage.
- OP6: time domain output of the receiver matched filter stage.

- OP7: time domain output of the receiver timing and frequency synchronisation stage.

It is worth noting that, as OP3 uses a hardware-accelerated FFT to convert data to the frequency domain, it is FFT frames rather than time-domain samples that are transferred to the PS via shared memory. This reduces the computation required on the Arm A53. In contrast, OPs 4-7 on the receiver side all use software FFTs to convert the time domain signals into the frequency domain. This difference between software and hardware FFTs, and how they affect performance, is explored in the following sections.

Plotting Performance

An important metric in the performance of the software is its plotting capabilities, specifically the average frame rate achieved for each type of plot. In the driver code for the OPs, the target frame rate for the time domain and constellation plots was set to 50 ms (20 FPS), while the FFT plots were set to 300 ms (3.3 FPS). The reason for this large differential is due to the nature of the data being plotted. Time domain signals contain far more data than an FFT frame, requiring a much faster redraw rate to keep pace with new samples of the input signal, whereas in the frequency domain the data updates at a much lower rate.

Since the real-time plots are drawn within the web browser on the computer, it is only necessary to measure the rendering performance on the client side, using the Chrome browser's built-in *DevTools*. Using this tool, the performance of the plot redraw time was measured over a 10 second window, while streaming each plot individually. It was found that the concept of a frame differs between Plotly and *DevTools*, meaning it was necessary to evaluate each redraw event by measuring the time between a function that is called only once per redraw operation. For the results described here, the function `o.prepareFn` was used. Figure 3.6 displays a bar chart of the average frame rendering times, while Table 3.1 tabulates this data, alongside the number of data points recorded and the standard deviation, σ. All timing results are given in milliseconds.

As can be seen from the data, the time domain and FFT plots render at almost their set timer periods of 50 ms (20 FPS) and 300 ms (3.3 FPS) respectively. However,

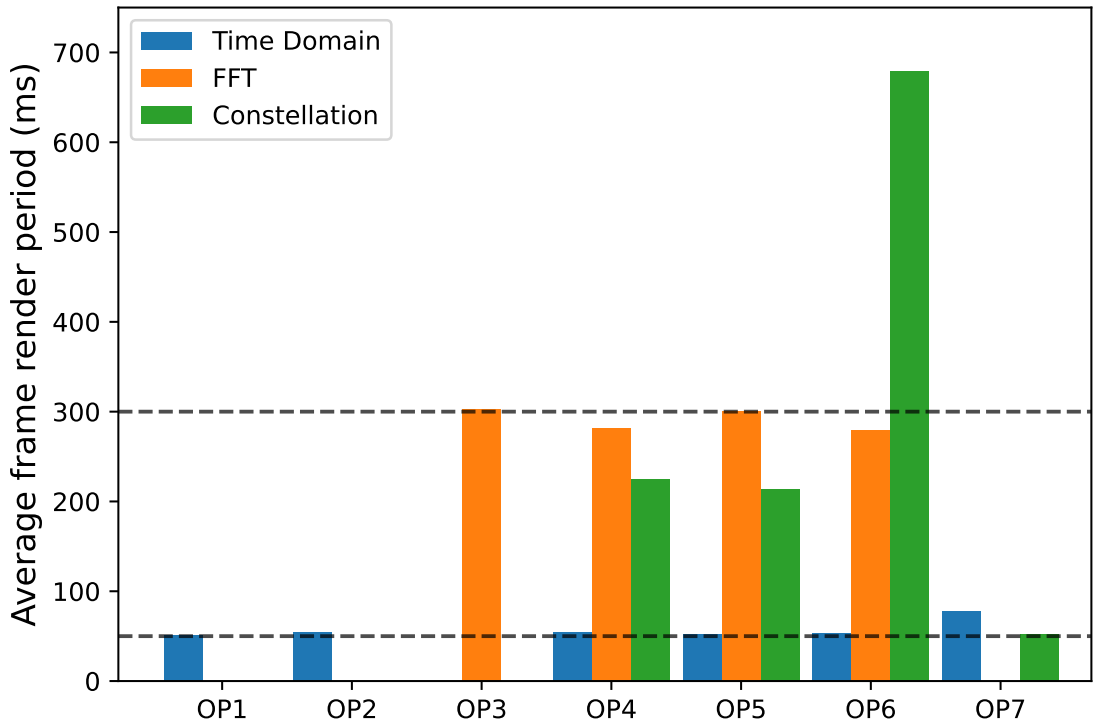

Figure 3.6: Average rendering time for each observation point (OP) and plot type. Dashed lines represent the target frame rates.

the constellation plot performance is largely dependent on the number of data points per plot, due to time being represented as opacity rather than an axis, causing more data points to be redrawn each frame. Furthermore, compared to the other OPs, the time domain plotting rate of OP7 is higher due to additional pre-processing required to recover and classify the bits as I and Q values, slowing performance down to an average of around 13 FPS.

Interestingly, there appears to be very little difference between the software and hardware implementations of the FFT in terms of plotting performance. This indicates that either a larger FFT could be used or a higher frame rate could be achieved. Although the hardware implementation of the FFT would always achieve better performance, due to the reduced amount of computation needed to plot the data in the frequency domain, these results show that, for the purposes of the demonstrator, adequate performance can be achieved with a software implementation. This potentially allows FPGA resources to be freed if required to implement other hardware functions. With that said, if more

Table 3.1: Rendering periods for each plot type and Observation Point (OP)

	Time domain			FFT			Constellation		
OP	Points	Avg (ms)	σ (ms)	Points	Avg (ms)	σ (ms)	Points	Avg (ms)	σ (ms)
Transmitter									
OP1	128	51	25.7	—	—	—	—	—	—
OP2	1024	54	20.2	—	—	—	—	—	—
OP3	—	—	—	1024	303	95.7	—	—	—
Receiver									
OP4	1024	54	24.3	1024	281	72.4	1024	225	61.0
OP5	1024	52	22.5	1024	300	1.7	1024	213	31.5
OP6	4096	53	19.1	4096	279	77.0	4096	679	56.8
OP7	128	77	27.3	—	—	—	128	52	22.6

PS processing power is needed for other functionality, then additional hardware FFTs could be used instead.

It was found that plotting performance was more than sufficient for human-eye debugging. While the data rate used in this demonstrator is relatively low, it has been shown in subsequent work that the visualisation performance can be maintained at much higher data rates, up to the maximum of the RFSoC [176]. This validates the use of the PS on the RFSoC for this purpose instead of requiring expensive external equipment.

PS to PL Latency

Another important metric is the latency between PS and PL when performing the control functionality. This was measured with the use of `timeit`, a built-in function within Python that runs selected code over a large number of iterations and outputs the average execution time. Using this method it was found that configuring the centre frequency of a given RF-DC, performed by a CFFI function call from Python to the C driver, took on average 4.23 ms. In contrast, the MMIO writes to the CPs within the transceiver design on the FPGA took only an average of 11.5 µs, two orders of magnitude faster than the CFFI call. This is an encouraging result due to the fact that PYNQ's MMIO module is written entirely in Python, an interpreted language not known for its speed. Therefore, if the full register map for the RF-DCs was ever

released by AMD, it would be possible to develop a much faster driver for the data converters, written in Python, by interacting with the MMIO directly.

Processing System Memory and Central Processing Unit (CPU) Usage

It is also worth considering how the PS performs during operation in terms of memory and CPU usage. As discussed in Chapter 2, the RFSoC contains a quad-core Arm Cortex A53 processor, running at a maximum clock speed of 1.3 GHz. Figure 3.7 shows a trace of the 4 GB of PS RAM and each of the CPU cores.

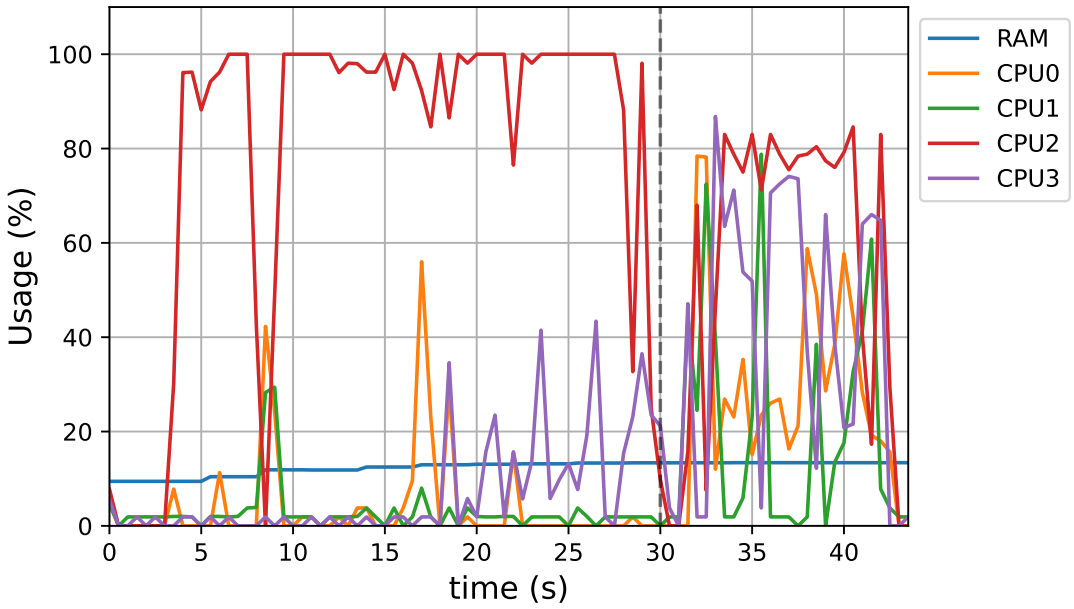

Figure 3.7: Memory and CPU usage of the Arm Cortex A53 on the RFSoC while loading the transceiver design (before 30s), and generating and streaming the live-update plots (after 30s).

At 0 seconds the system is idle, using 9% of RAM and very little CPU activity. These resources are used to run the Ubuntu OS, Jupyter, and a variety of other background processes used by PYNQ—constituting the baseline.

From 0 to 30 seconds the transceiver is initialised, generating the time domain plots for OP1, OP2, and OP4. As can be seen from the data, the memory usage only increases slightly, reaching a maximum of 13%. This is achieved by reusing the same DMA buffer for each of the OPs, rather than relying on the periodic garbage collection performed by Python. Additionally, a single CPU core is at maximum utilisation for most of this

time period. It is important to note that this is not necessarily due to the amount of processing that is actually occurring on the PS. The CPython interpreter, used by Jupyter, does not run code concurrently, even for multi-threaded programs [184].

However, due to the implementation of the DMA software there is a blocking wait period between each DMA transaction, causing the CPU core to idle at full capacity. This could be mitigated by instead using interrupts, allowing other processes to occur in between the DMA transaction wait periods.

After 30 seconds the time domain plot for OP4 is generated, set to run at 50 ms intervals. It can be seen from the data that CPU 2 is now capped at around 80%, due to the blocking process of the DMA. The reason it is not at 100%, as with the previous plots, is due to the reduction in data rate. Only 33 ms of samples are requested for every 50 ms interval, leaving the CPU idle in between these periods.

These results show that the PS of the RFSoC is more than capable of performing the tasks required of the transceiver described in this chapter, including multiple data visualisation functions that would typically be performed off-chip on specialised equipment. While there are many optimisations that could be made to the code written for this project, these initial software results are encouraging for the use of the RFSoC for reconfigurable SDR applications.

3.6.4 Hardware Performance

Table 3.2 shows the FPGA resource utilisation for the complete transceiver design. It was found that a significant portion of logic resources were used by the productivity components of the design, with approximiately 2% of the total LUTs and 4% of total DSPs used by the DMAs for the OPs.

While in low-power devices, such as the Artix-7 series, this overhead might be significant, in the RFSoC this usage is negligible. This demonstrates that by trading a small percentage of FPGA resources, the design methodology presented in this chapter can provide significant productivity benefits. This modest usage leaves a large amount of resources available for implementing multiple instances of the transceiver

(for a multi-channel design), or additional transceivers that serve different modulation schemes.

Table 3.2: FPGA resource utilisation of the transceiver on the XCZU28DR RFSoC.

	LUTs	BRAMs	DSPs	MMCM	PLL
Transmitter	14602 (3.43%)	43.5 (4.03%)	134 (3.14%)	0 (0%)	1 (6.25%)
Receiver	18433 (4.33%)	31.5 (2.92%)	79 (1.85%)	1 (12.5%)	0 (0%)
RF-DCs	2460 (0.58%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Total	35630 (8.38%)	75 (6.94%)	213 (4.99%)	1 (12.5%)	1 (6.25%)

3.7 Concluding Remarks

This chapter has presented a design methodology for the development of SDR systems on the RFSoC, with a focus on productivity.

To evaluate this methodology, a fully functional, reconfigurable RF transceiver for the RFSoC, incorporating both control and visualisation on a single device was implemented. The design made use of the PS, PL, and RF-DCs to develop a working SDR system, where the RF components can be reconfigured and data inspected along the signal path on-the-fly, eliminating the need for external equipment such as spectrum analysers or oscilloscopes. The transceiver was developed using Model Composer for the HDL IP cores, while the software was built around the PYNQ framework, marking the first publication to use PYNQ for RFSoC system design.

The transmitter and receiver chains were implemented using different IP design strategies—monolithic and modular, respectively—which helped to highlight the impact of design methodology on software development. In particular, the modular approach resulted in cleaner and more maintainable driver code, demonstrating the benefits of this design approach for IP development.

Performance results show that Python, while not as efficient as a system-level language such as C, is still capable of meeting the requirements of this design. The use of standard Python libraries, such as Plotly and ipywidgets, for visualisation and control within the Jupyter environment provided a user-friendly interface and shorter

development time compared to traditional methods. The plotting achieved updates at around 20 FPS for time domain plots, which is fast enough for most prototyping and debugging tasks. The difference between software and hardware FFTs was smaller than expected, suggesting that software-based visualisation may be sufficient for many applications. Additionally, resource utilisation results demonstrate that the productivity overhead is negligible relative to the available resources on the RFSoC.

Finally, this work has helped to establish PYNQ as a viable framework for SDR development on the RFSoC, and has since been used in a number of publications and projects. It shows that full-stack SDR systems can be prototyped and tested on a single device without the need for external equipment, and that publishing these designs as open-source encourages collaboration and further work in the field. This has proven useful not just for SDR, but also in other areas, such as radio astronomy [185] and quantum computing [186].

Chapter 4

A Run-time Reconfigurable Frequency Planning Tool for RF-Sampling Radio Devices

4.1 Chapter Overview and Contributions

Traditionally, frequency planning is performed offline, using a combination of simulation and hardware testing to determine suitable parameters for a given system. In dynamic systems, this process must instead be performed in real-time, with parameters updated during run-time in response to changing conditions. As established in Chapter 2, existing frequency planning tools remain limited to offline analysis and manual use, and do not provide direct support for real-time operation in reconfigurable radio systems.

This chapter presents Spur Planning, Evaluation, and Configuration Tool for RF-sampling dEvices (SPECTRE): a scriptable frequency planning tool for RFSoC devices that supports run-time selection of spur-free configurations that can be applied directly on the device for bandlimited signals. The tool is designed to be callable from a higher-level radio control process, such as the reconfiguration phase of a cognitive radio cycle. Building on the theoretical background and problem formulation established in Chapter 2, this chapter focuses on the implementation of SPECTRE and its use as a practical component within a reconfigurable RF system. SPECTRE is then used to

evaluate a small set of simple search strategies for real-time frequency planning, with the results used to provide guidance on algorithm choice and show practical limitations.

The emphasis of this chapter is to establish feasibility and suitability of SPECTRE under real-time constraints, rather than on identifying optimal search methods. In summary, the main contributions of this work are as follows:

- A feasibility-driven demonstration that frequency planning for RF-sampling devices can be performed at run-time on RFSoC hardware, making it suitable for use in cognitive radio applications operating under real-time constraints.
- The design and implementation of a scriptable frequency planning tool for RFSoC devices, allowing frequency planning to be integrated into autonomous radio control processes.
- A comparative evaluation of simple brute-force search strategies, used to assess which classes of algorithms demonstrate execution time and deterministic properties compatible with real-time operation.

All data underpinning the work in this chapter are openly available from the University of Strathclyde KnowledgeBase at [187].

4.2 Frequency Planner Tool Implementation

The task of creating an automated frequency planning tool for RF-sampling devices can be framed as a search problem over a large combinatorial space. Based on the theoretical background given in Chapter 2, given a centre frequency, f_c, and signal bandwidth, BW, the objective is to identify a valid sample rate, f_s and associated PLL frequency that produce a valid frequency plan. This requires searching a large, discrete parameter space of possible sample rates and derived PLL frequencies, while satisfying various design constraints, such as:

- Spurious-free operation: Ensuring no significant spurs overlap with the signal of interest defined by f_c and BW.

- Bandwidth constraints: Ensuring the sample rate is sufficiently high for the specified bandwidth.
- Device constraints: Hardware-specific limitations, such as PLL frequency and sample rate ranges.

The design and architecture of SPECTRE addresses this challenge through constraint-based pruning and simple, sequential search strategies.

4.2.1 Design and Architecture

The initial design of the frequency planning tool was a fully-operational, GUI-driven Python-based application, released as an open-source project. The tool allows users to input system parameters and generate a graphical representation of the frequency plan, similar to many of the existing offline frequency planning tools discussed in Chapter 2.

This earlier implementation comprised three components: a calculation engine for frequency and spur analyses, a user interface providing interactive parameter exploration, and an application layer integrating both calculation and interface components. The source code for this earlier implementation is freely available online, allowing the reader to inspect, use, or modify it as needed [188].

The newly developed tool, SPECTRE, builds upon this earlier design, reusing and extending the calculation module. It comprises three additional functional components: PLL frequency generation, an automated search module, and integration with the RFSoC. The search module controls the overall process, selecting candidate sample rates and PLL frequencies and invoking the calculation and PLL generation modules as needed to evaluate each configuration.

The inputs and outputs of each component are summarised as follows:

- The calculation module takes as input a signal centre frequency, bandwidth, and a selected set of spur mechanisms, and produces as output a frequency plan consisting of the signal band and the corresponding spur frequency ranges.

- The PLL frequency generation module takes as input a candidate sample rate and the target device configuration, and produces as output a list of valid PLL frequencies that are compatible with that sample rate.
- The automated search module takes as input the user-defined signal parameters and device constraints, and produces as output either a valid pair of sample rate and PLL frequency, or an indication that no valid configuration exists.
- Finally, the RFSoC integration module takes the selected sample rate and PLL frequency and applies them directly to the hardware.

Each of these components are detailed in the following subsections. Additionally, complete code listings for these modules are available at [187].

Calculation Module

This module performs the spur calculations using the equations introduced in Chapter 2 Section 2.3.2. Inputs to this module include the centre frequency, bandwidth and the selection of spur types, while the output is the associated frequency plan containing the signal of interest and the user-specified spurs.

PLL Frequency Generation

This module automatically generates a valid set of PLL frequencies based on the user-defined sample rate, and device configuration. It replicates and extends the logic described in the RFSoC RF-DC documentation and the original *xrfdc* driver [168, 183].

As discussed in Chapter 2, the PLL frequency of the RFSoC must be some fraction of the sample rate such that

$$f_s = \frac{f_{pll}}{R}\frac{N}{D}, \tag{4.1}$$

where f_s is the sample rate, f_{pll} is the PLL frequency, R is the input reference clock divider, N is the feedback divider, and D is the output sample clock divider. Rearranging this equation to determine a valid value for f_{pll} gives

$$f_{pll} = \frac{f_s RD}{N}. \tag{4.2}$$

Each divider has a subset of valid values that are device dependent. Therefore, to generate a valid set of PLL frequencies, the module iterates through all valid combinations of R, N, and D. The exact numeric limits and allowed values are taken from the device datasheet and driver [168, 183].

By only generating PLL frequencies that are valid for the given input parameters, this ensures that the search module only evaluates valid configurations, reducing the search space and improving efficiency.

Automated Search Module

This module receives the user-defined centre frequency and bandwidth, generates the corresponding list of candidate sample rates, prunes and randomises the list, then iteratively tests each sample rate against the generated PLL frequencies. In this context, a candidate sample rate simply refers to a valid sampling frequency within the limits supported by the target device and the signal of interest.

Pruning removes values that are unsuitable based on known constraints. At the sample-rate stage, candidate values are restricted to the range supported by the target device and must exceed the signal centre frequency, ensuring that the signal lies within the first two Nyquist zones. At the PLL stage, additional pruning is applied by discarding PLL frequencies that lie within a fixed threshold of the signal centre frequency, as these configurations are known to produce undesirable mixing products. The default threshold is set to 1 MHz, but this can be adjusted by the user as needed.

As this version of the tool is primarily focused on demonstrating feasibility, the search proceeds sequentially through the resulting list in a brute-force manner, and terminates as soon as a valid configuration is identified. The search includes a randomise option that, when enabled, shuffles the order in which the sample rates are tested. Randomisation is included as an alternative ordering of the candidate lists, rather than a change to the underlying search procedure.

For each candidate sample rate and PLL frequency, the module generates a frequency plan using the calculation module, and checks for any spurs that overlap with the signal of interest using the following logic:

$$(S_{min} < P_{max}) \wedge (S_{max} > P_{min}), \tag{4.3}$$

where S_{min} and S_{max} are the minimum and maximum frequencies of the signal of interest, P_{min} and P_{max} are the minimum and maximum frequencies of the spur, and $\wedge$ represents the logical AND operation.

The following steps describe the search process at a high level and are intended to help relate the text to the flowchart shown in Figure 4.1. The goal of the algorithm is simply to find any configuration that meets the spur-free requirements, rather than to compare or rank multiple possible solutions.

1. Initialise the frequency planner with the user-defined signal centre frequency, bandwidth, and spur types to check for in-band interference. Generate a list of valid sample rates for the given inputs, then prune and randomise this list.

2. Select the next sample rate from the list and update the frequency planner model to reflect this choice, generating the corresponding signal band and sample-rate-dependent spurs.

3. Evaluate whether the current sample rate provides a valid frequency plan based on input specifications using the calculation module. If valid, generate a list of compatible PLL frequencies. If invalid, return to step 2 and select the next sample rate.

4. Select the next PLL frequency from the list and generate the corresponding frequency plan.

5. Evaluate whether the current PLL frequency provides a valid frequency plan. If valid, return the sample rate and PLL frequency as the solution. If invalid, return to step 4 and select the next PLL frequency. If all PLL frequencies have been

exhausted, return to step 2 and select the next sample rate. If all sample rates have been exhausted, return an error indicating no valid solution exists.

Figure 4.1 provides a visual overview of this process. The initialisation block corresponds to the setup described in Step 1. The decision point labelled "Valid f_s?" reflects the sample-rate evaluation in Step 3, where spurs associated with the current sample rate are checked against the signal band. The subsequent loop over PLL frequencies corresponds to Steps 4 and 5, with the "Valid PLL Freq?" decision representing the evaluation of PLL-dependent spurs. The loop-back paths in the figure indicate the cases where a candidate sample rate or PLL frequency is rejected and the search continues with the next available option.

The search terminates successfully as soon as a sample rate and PLL frequency are found for which no in-band spur overlap is detected. If all candidate PLL frequencies are rejected for a given sample rate, the algorithm proceeds to the next available sample rate. If all candidate sample rates are exhausted without identifying a valid configuration, the algorithm terminates and reports that no feasible solution exists for the given inputs.

For completeness, a formal algorithmic description defining the variables and control flow of the search procedure is provided in Appendix C.

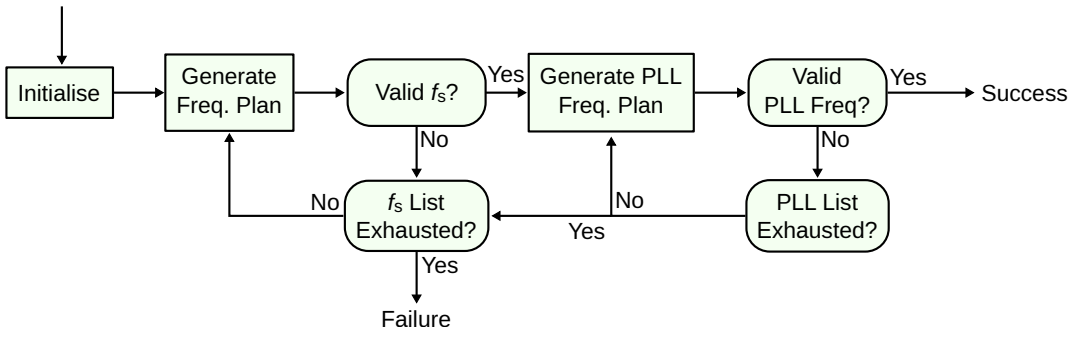

Figure 4.1: Flowchart of the sequential search algorithm through candidate sample rates and PLL frequencies to identify a valid frequency plan.

RFSoC Integration

Once the search algorithm identifies a valid sample rate and PLL frequency, this module can directly reconfigure the RF data converters on the RFSoC device. This capability

enables practical, real-time reconfiguration of the hardware, suitable for automated real-time applications.

This module utilises the PYNQ framework for RF-DC control, including the *xrfdc* and *xrfclk* drivers [50]. In particular, the *xrfdc* driver is used to reconfigure the RF-DCs with the new sample rate and PLL frequency, while the *xrfclk* driver is used to set the reference clock.

4.3 Results

This section presents the results of a series of experiments designed to evaluate SPECTRE using metrics relevant to its intended use in CR systems. This includes the ability to find valid spur-free configurations under changing constraints, and the execution time and predictability of the tool when run on representative hardware.

The results in this chapter are presented without a direct comparison to existing state-of-the-art frequency planning methods, as described in Section 2.3.4. Existing frequency planning tools from device vendors are designed for offline, manual analysis and do not support scripted or autonomous operation. Academic approaches reported in the literature focus on narrower problem settings, such as single-tone signals or transmitter-only architectures, and typically evaluate performance using signal quality metrics, such as SFDR and SNR, at a single fixed configuration. These approaches do not address repeated reconfiguration, execution time, or suitability for use inside an autonomous control loop, as described in Chapter 2 Section 2.3.4, which is the intended use case for SPECTRE.

The experiments were conducted across thousands of test cases with varying centre frequencies, bandwidths, and spur constraints. For each test case, the frequency planner searched for a valid combination of sample rate and PLL frequency that would provide a spur-free signal. The chosen device for these experiments was the XCZU48DR, which determined the available sample rates and PLL frequencies.

To evaluate how the performance of SPECTRE varied as spectral constraints changed, three different spur tiers were defined, each representing a different level of spectrum complexity:

- Low Tier: Includes the most dominant spurs, such as second- and third-order harmonics and primary interleaving spurs.
- Medium Tier: Extends the Low Tier by including additional higher-order harmonics and all interleaving spurs.
- High Tier: Encompasses all known spur types, including interleaved harmonics.

Each tier represents a different level of constraint on SPECTRE. Lower tiers simplify the search process but may result in spurs, not included in the search, overlapping with the signal of interest, leading to lower SNR. The higher tiers, in contrast, consider a wider range of spurs, making it less likely to find a valid configuration, particularly for wideband signals.

Additionally, three different search space algorithms were evaluated. These algorithms were selected to represent distinct ways of exploring a large, discrete search space, rather than to identify an optimal search strategy. The aim was to assess feasibility and practical trade-offs relevant to real-time operation, including whether an exhaustive search is required, the execution time of the planner, and the predictability of its behaviour. In this context, an exhaustive search is one that evaluates all valid configurations in the defined search space, ensuring that a valid solution will be found if one exists.

- Ordered Search: An exhaustive search that evaluates all valid configurations in a fixed, ascending order, terminating as soon as a valid solution is found or the search space is exhausted.
- Random Search: An exhaustive search in which all valid configurations are evaluated exactly once, but in a randomised order. The search terminates when a valid solution is found or all configurations have been evaluated.
- Coarse Grid Search: A non-exhaustive search that evaluates a uniformly subsampled grid of the parameter space. This reduces the number of evaluations but may miss valid configurations that lie between grid points.

These algorithms were chosen to examine different aspects of the frequency planning problem. Ordered search provides a baseline, establishing the upper bound for computational effort. Random search remains exhaustive, while removing ordering bias, allowing the effect of randomisation on execution time and variability to be examined. Coarse grid search is a pragmatic approach aimed at reducing execution time, making it suitable for assessing whether a fast, non-exhaustive strategy can provide acceptable solutions for real-time operation.

Together, these experiments characterise the trade-offs between completeness, execution time, and predictability across system constraints and different algorithmic approaches.

All data collected during these experiments, along with the code used to generate the results, has been made publicly available at the University of Strathclyde's online repository for reproducibility and further research purposes [187].

4.3.1 Frequency Plan Success and Failure Rates

This first set of experiments focused on the more general success and failure rates of SPECTRE across all spur tiers and a wide range of bandwidths. This analysis was necessary to establish a baseline understanding of SPECTRE's performance before evaluating the performance of the different search algorithms. Therefore, to gather these results, the exhaustive ordered search algorithm was used. This ensured that if a valid solution existed, it would be found.

Success Rate by Bandwidth and Spur Tier

This first experiment measured the success rate of finding a valid frequency plan for each spur tier across different bandwidths, from 10 MHz to 100 MHz in increments of 10 MHz. Each bandwidth was tested with a range of centre frequencies, from 1 GHz to 5 GHz in increments of 10 MHz, resulting in 400 centre frequencies tested per bandwidth per tier.

Figure 4.2 shows the success rate of each tier as a function of bandwidth. Each bar shows the contribution of a given tier to the total number of successful frequency plans

found. The results show that for bandwidths up to 70 MHz, all tiers achieved a 100% success rate, indicating that, across these bandwidths, there is always a configuration that will result in a spur-free signal.

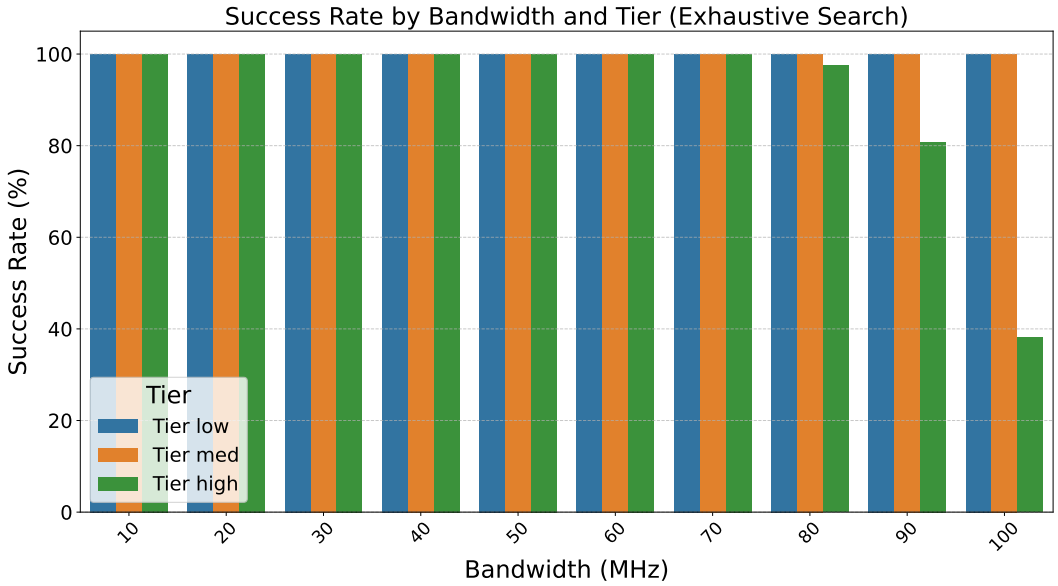

Figure 4.2: Success rates of frequency plans by bandwidth.

However, the high tier success rate begins to drop as the bandwidth passes 70 MHz. This is an expected result and reflects the increased number of spurs considered in the high tier. As more spurs are added, the probability that one will overlap with the signal of interest increases, particularly for wideband signals.

This highlights an important trade-off in system design. While reducing the number of considered spurs increases the likelihood of finding a valid frequency plan, particularly for wideband signals, this may result in reduced signal quality. Although spurs considered in the high tier are typically lower in amplitude, they can still introduce a measurable drop in SNR if they overlap with the signal of interest. For RF systems with stringent SNR requirements, this could result in an unacceptable drop in performance. Therefore, for these types of systems, it may be necessary to reduce the bandwidth of the signal to ensure spur-free operation.

Distribution of Failures by Spur Type

To better understand which spurs were most likely to result in invalid frequency plans, this experiment tracked the cause of each failure across a comprehensive sweep of configurations. This included bandwidths from 10 MHz to 100 MHz and centre frequencies from 1 GHz to 5 GHz.

Rather than simply counting the number of failures, the experiment tracked the individual spur responsible for each configuration failure. This data was then aggregated to identify the spurs that most frequently resulted in unsuccessful frequency plans.

Figure 4.3 shows the ten most frequently occurring spurs that caused invalid frequency plans, sorted by overall percentage. The results show that the most problematic spurs are the harmonics and the interleaving-related harmonics, accounting for over 70% of all failures.

The fifth-order harmonic (HD5) is the most common cause of failure which, along with HD3 and HD4, account for over a fifth of all failures. This is consistent with expectations, as the bandwidth of the harmonics is proportional to the bandwidth of the signal of interest, as discussed in Section 2.3.2, and therefore more likely to cause in-band interference.

Several interleaving-related harmonic spurs also appear prominently in the results. While these are typically much lower in amplitude than the main harmonics, and therefore less likely to cause significant interference, for systems with high SNR requirements, as discussed in the previous section, these spurs may still need to be considered.

4.3.2 Algorithm Performance

With a baseline understanding of SPECTRE, the next set of experiments focused on evaluating the performance of the three search algorithms. The goal of these experiments was to assess which algorithm was best suited for real-time applications by measuring the success rate and computational efficiency of each algorithm under different configurations.

Computational efficiency is defined here as the number of valid frequency plans found (successes) divided by the total number of iterations required to find them, where

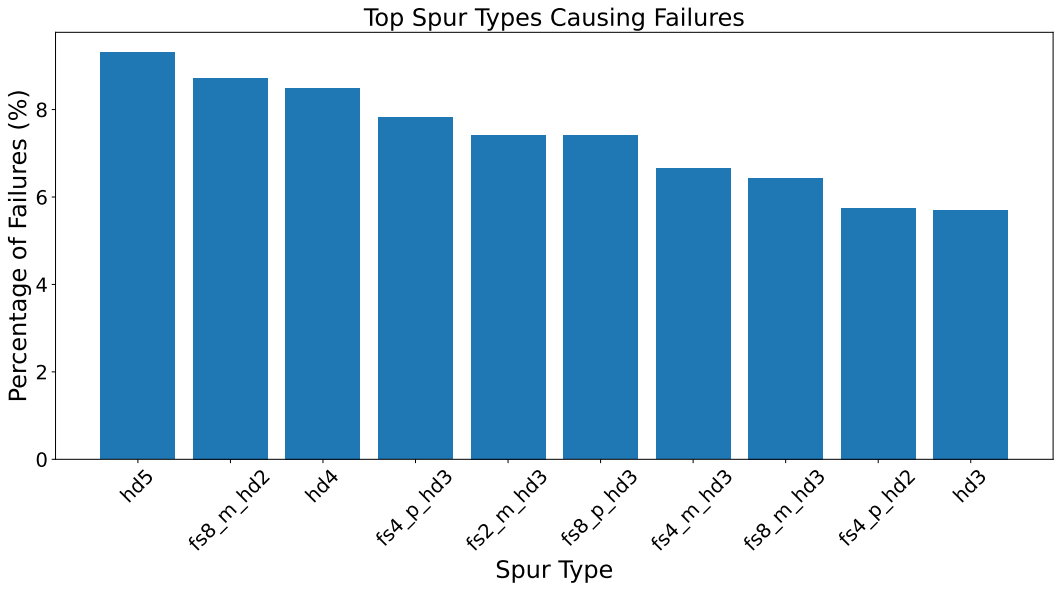

Figure 4.3: Top spur types causing failures. HDn denotes the nth harmonic (nf_{in}). All spurs of the form fs$x$_p/m_hd$n$ correspond to interleaving-related harmonic spurs. fs4 and fs8 indicate sample-rate fractions (fs/4 and f_{s}/8), while p/m denote additive or subtractive mixing. For example, fs8_m_hd2 represents a spur at $f_{\text{s}}/8 - 2f_{\text{in}}$.

an iteration corresponds to the evaluation of a single candidate configuration. This metric is dimensionless, with higher values indicating better performance.

Algorithm Efficiency by Spur Type

This experiment measured the computational efficiency of three search space algorithms—ordered search, random search, and coarse grid search—across the three spur tiers. The purpose of this analysis was to determine how each algorithm performed under increasing signal quality requirements. In particular, the goal was to identify which algorithms were most efficient at finding valid configurations, measured by the number of candidate configurations evaluated before a valid solution was found, as the solution space decreased with increasing spur complexity.

Figure 4.4 shows the mean efficiency of each algorithm within each spur tier. As expected, efficiency decreases as spur complexity increases, reflecting the reduction in solution space caused by the increased number of spurs. The results show that, for all tiers, the random search algorithm consistently outperforms the others in terms of

efficiency, particularly in the low and medium spur tiers where the solution space is larger. The coarse grid algorithm is also shown to perform well, especially in the low and medium spur tiers.

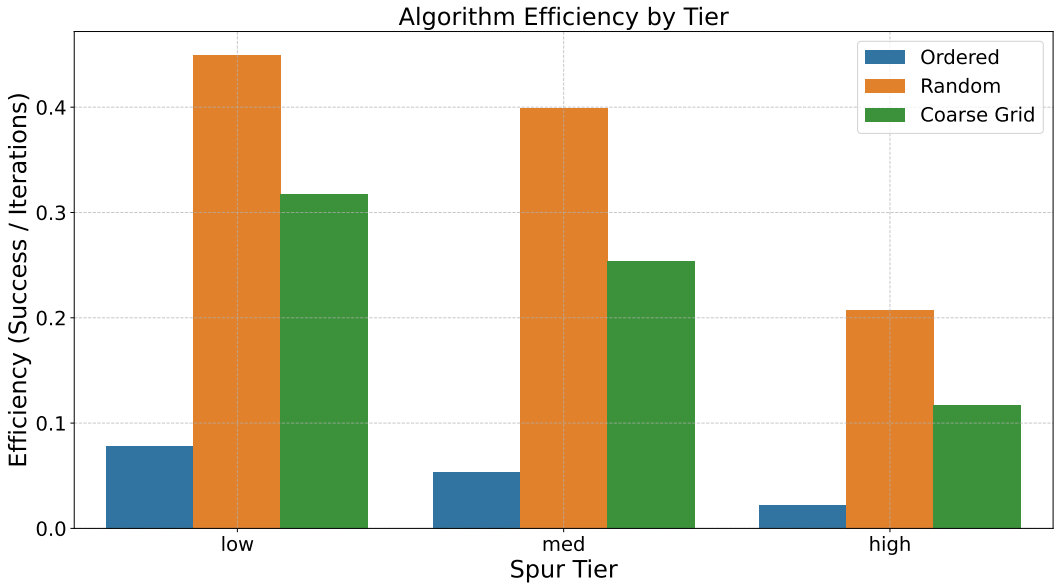

Figure 4.4: Algorithm efficiency by spur tier.

The ordered search algorithm shows the lowest efficiency across all spur tiers. As this algorithm checks configurations in ascending order, it can spend many iterations inside regions of nearby sample rates that all produce invalid plans for a given input. In contrast, the random search algorithm is less likely to stay confined to such regions, as it samples configurations from across the entire solution space in a random order. One potential solution to this is to reverse the order of the ordered search, starting with the highest sample rates and working downwards. However, this would most likely always result in a higher sample rate being selected, which may not be desirable in all cases, particularly for power-constrained applications. This is left as an area for future work.

As the ordered search and random search algorithms are both exhaustive, they are guaranteed to find a valid frequency plan if one exists. In contrast, the coarse grid search algorithm is not guaranteed to find a valid solution, as it only searches a subset of the solution space. Therefore, these results show that the random search algorithm is the most efficient for finding a valid frequency plan.

Success Rate for Each Algorithm by Bandwidth

This experiment measured how each search space algorithm responded to increasing signal bandwidth, focusing on their ability to find valid solutions under growing spectrum constraints. All three algorithms were tested across a range of bandwidths from 10 MHz to 100 MHz in increments of 10 MHz, with a range of centre frequencies tested for each bandwidth, from 1 GHz to 5 GHz in increments of 10 MHz. The goal was to determine which algorithms were prone to failure as the solution space decreases with the increasing bandwidth. All three spur tiers were tested and the results aggregated to provide a general overview of algorithm performance.

Figure 4.5 shows the success rate of each algorithm as a function of bandwidth. As also observed in Figure 4.2, the exhaustive search algorithms, random and ordered, maintain a 100% success rate up to 80 MHz, followed by a sharp decline, reaching around 80% at 100 MHz bandwidth. As both the ordered and random search algorithms are exhaustive, they guarantee that a valid solution will be found if one exists. Therefore, any reduction in success rate is due to the absence of valid solutions, rather than a failure of the algorithm itself.

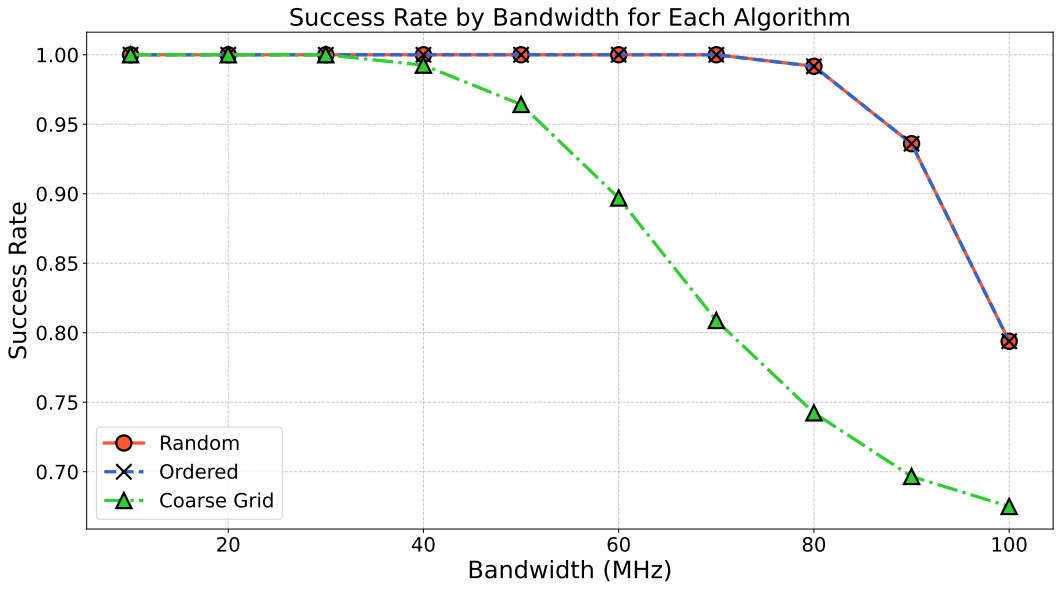

Figure 4.5: Success rate for each algorithm by bandwidth.

In contrast, the coarse grid search algorithm shows a clear decline in performance as bandwidth increases. Starting at around 40 MHz, the success rate begins to drop, eventually falling below 70% at 100 MHz. This is due to the fact that the coarse grid search only searches a subset of the solution space, and is more likely to miss valid solutions as the bandwidth increases.

These results highlight the importance of using an exhaustive search in wideband applications where valid configurations are sparse. They also show that increasing signal bandwidth places a limit on the ability to find spur-free configurations. This implies that systems requiring spur-free operation may need to operate within more restrictive bandwidth limits.

Algorithm Computational Efficiency

The following two experiments measured the computational efficiency of the three search space algorithms—random search, ordered search, and coarse grid search—across approximately 12,000 test cases with varying centre frequencies, bandwidths, and spur tier constraints. One focused on the number of iterations required to search the solution space, while the other measured the total execution time to find a valid solution for each algorithm.

The two separate experiments were performed for complementary reasons. First, iteration count provides a hardware-independent measure of algorithm efficiency. Second, execution time measurements determine the algorithm's suitability for real-time applications.

Iteration-Based Analysis

To evaluate the computational cost of each search algorithm, the number of iterations required to find a valid configuration was recorded for both the sample rate and PLL search phases. Iteration count is used as a proxy for computational effort because each iteration follows the same sequence of checks, and differences between search algorithms arise from how many such checks are required before a valid solution is found. This

allows the algorithms to be compared in a way that is independent of implementation details and execution platform.

Figure 4.6 shows two box plots comparing iteration distributions across the three algorithms for both sample rate and PLL searches. The data is also summarised in Table 4.1, which shows the mean, median, maximum, and standard deviation of iteration counts for each algorithm.

To better understand these plots it is worth reviewing the box plot nomenclature. Each box plot shows the distribution of iteration counts for each algorithm. The blue box represents the interquartile range (IQR) and spans the 25th to 75th percentiles. Its height is a direct measurement of variability. The vertical lines above the main box extend to 1.5x the IQR, indicating where the bulk of the data lies beyond the middle point. Finally, the dots represent outliers beyond this range, denoting rare cases that deviate from the main distribution.

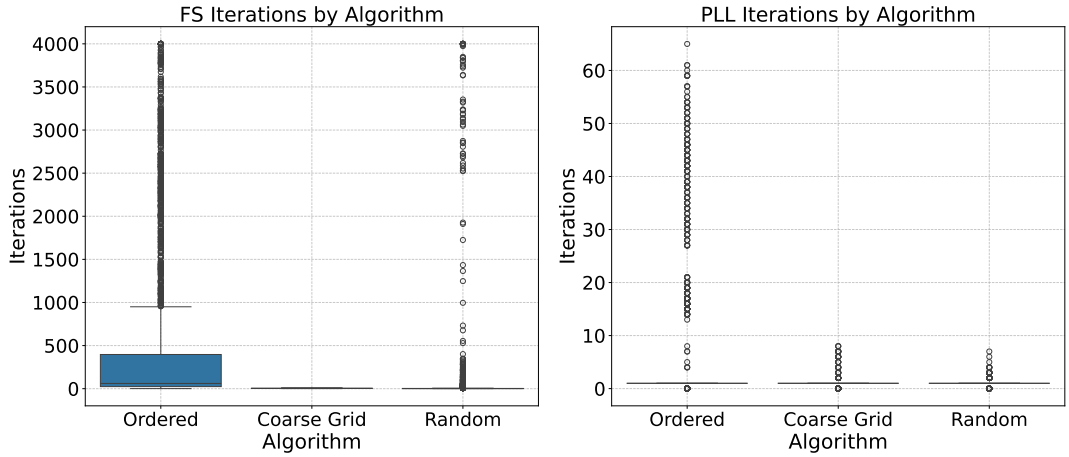

Figure 4.6: Distribution of sample rate, PLL, and total iteration counts for each search algorithm.

The results show that for sample rate iterations, the ordered search algorithm exhibits the highest and most variable number, consistent with its sequential search approach. In contrast, the random search and coarse grid search algorithms show much tighter main distributions. Although the random algorithm displays the lowest median, its standard deviation is almost 200x compared to the coarse grid search.

Table 4.1: Summary of f_s and PLL iterations required by each algorithm.

Algorithm	**Mean**	**Median**	**Max**	**Std**
f_s Iterations				
Coarse Grid Search	4.5	3.0	11	3.3
Random Search	116.5	1.0	3999	593.9
Ordered Search	515.7	59.0	3999	982.7
PLL Iterations				
Coarse Grid Search	1.0	1.0	8	0.7
Random Search	1.0	1.0	9	0.4
Ordered Search	6.1	1.0	66	12.7

In comparison, the PLL list iteration counts are orders of magnitude lower across all search algorithms, with only modest variation for both random and coarse grid searches. This suggests that the majority of computational cost is spent during the sample rate search.

These results highlight not only the mean iteration count, but also the variance and presence of outliers in the data. For real-time applications, deterministic behaviour is as important as average performance. Therefore, the coarse grid search algorithm offers the best overall performance, achieving both low average iteration count and greater predictability.

Execution Time Analysis

While iteration count provides a hardware-agnostic measure of computational effort, execution time reflects real-world performance. For each centre-frequency and bandwidth configuration, execution time was measured for a single run of each search algorithm, from the start of the search until either a valid frequency plan was found or the candidate list was exhausted. Each reported timing therefore corresponds to one complete frequency planner execution for a given configuration and algorithm. A sample size of 1,230 configurations was taken for each algorithm, spanning centre frequencies from 1,000 MHz to 5,000 MHz, signal bandwidths from 10 MHz to 100 MHz, and three spur tiers.

Table 4.2 shows the mean, median, standard deviation, and maximum execution times for each algorithm, run on the RFSoC4x2 development board. These results

confirm the trend observed in the iteration analysis, that the ordered search algorithm is the slowest and most inconsistent, with execution times taking over 34 seconds in some cases. The random search algorithm performs better, but has high standard deviation. The coarse grid search algorithm, in contrast, demonstrates both low execution times and variance, with a mean execution time of 39.60 ms and a median of 21.09 ms—potentially fast enough for real-time performance.

While the random search algorithm's median is comparable to that of the coarse grid search, its high variance suggests that it may not be suitable for real-time applications. Overall, the ordered search and random search are, on average, 101x and 24x slower than the coarse grid search algorithm, respectively.

Table 4.2: Summary of execution times for each search algorithm run on an RFSoC4x2 development board.

Algorithm	**Mean (ms)**	**Median (ms)**	**Max (ms)**	**Std (ms)**
Coarse Grid Search	39.60	21.09	182.69	37.27
Random Search	957.07	30.02	32858.20	7985.28
Ordered Search	4030.91	110.62	34829.90	4684.42

Additional results were obtained by running the same experiment on an AMD Ryzen 9 7940HS x86-based CPU to act as a baseline for comparison. A summary of these execution times are shown in Table 4.3.

Table 4.3: Summary of execution times for each search algorithm run on an AMD Ryzen 9 7940HS CPU.

Algorithm	**Mean (ms)**	**Median (ms)**	**Max (ms)**	**Std (ms)**
Coarse Grid Search	2.28	1.33	12.49	2.11
Random Search	50.98	1.52	1710.53	244.75
Ordered Search	226.81	7.35	1977.36	445.46

These results show that all search algorithms perform better on the x86-based CPU than on the RFSoC4x2 development board. The coarse grid search algorithm is still the fastest, with a mean execution time of 2.28 ms and a median of 1.33 ms. This is consistent with expectations, as the x86-based CPU has a significantly higher clock speed and more powerful hardware than the Arm Cortex A53 available on the RFSoC4x2—with max clocks speeds of 5.2 GHz and 1.2 GHz, respectively.

The purpose of this comparison is to demonstrate that the performance of the frequency planner tool is dependent on the underlying hardware and the implementation of the algorithms. With further optimisation, it may be possible to achieve lower execution times on the RFSoC itself. For example, the use of parallel processing or hardware acceleration could be implemented, or a system programming language such as C/C++ could be used to implement the algorithms, rather than Python. These improvements could further reduce the execution time of the frequency planner tool, making it more suitable for real-time applications.

4.3.3 Summary of Results

The results detailed in this section demonstrate that, while all three search algorithms are capable of finding valid frequency plans, they differ significantly in terms of computational cost and consistency. The ordered search, while exhaustive, exhibits poor efficiency and highly variable performance. The random search improves efficiency but can still suffer from unpredictable performance. The coarse grid search offers the most deterministic performance, combining low execution times with predictable behaviour.

However, it is important to note that the coarse grid search algorithm is not exhaustive and, as has been shown, may fail to find a valid solution even when one exists, especially at higher bandwidths. Therefore, for the best performance, a hybrid approach may be more appropriate, where a coarse grid search is first attempted for speed, falling back to an exhaustive random search only if required. This approach would provide fast operation in most cases, while still ensuring that a valid solution is found if one exists.

4.4 Concluding Remarks

This chapter presented the design, implementation, and evaluation of a run-time reconfigurable frequency planning tool for RF-sampling devices, focusing on deployment within autonomous, real-time radio applications.

First, a novel frequency planning tool, SPECTRE, was introduced, capable of identifying valid sample rate and PLL frequency configurations for arbitrary signal centre frequencies and bandwidths. The tool was then shown to be integrated into the RFSoC, allowing it to reconfigure the RF data converters based on the input and output settings.

To evaluate the performance of the planner, a set of experiments were conducted, focusing on the performance of the planner for real-time applications. The results evaluated the tool's success rates across different spur tiers and signal bandwidths and identified the most common spurs that cause in-band interference. Additionally, the computational efficiency of three simple search algorithms was assessed—ordered, random, and coarse grid searches—using both iteration counts and execution time to evaluate performance.

These results revealed two key trade-offs. First, the most common spurs that cause in-band interference were the harmonics, with lower-amplitude interleaving spur harmonics also appearing frequently—appearing in the low and high spur tiers, respectively. The results showed that while including all spurs in frequency planning ensures spectral integrity, it reduces the likelihood of finding a valid frequency plan, especially for wideband signals. This suggests that for RF systems with stringent SNR requirements, it may be necessary to reduce the bandwidth of the signal to ensure that a spur-free configuration can be found. Second, the random search algorithm demonstrated the most efficient performance between success rate and iteration count, but exhibits high mean and variance in execution time. In contrast, the coarse grid search algorithm demonstrated the fastest execution time and lowest mean and variance. However, unlike the random search, it is not exhaustive and may miss valid solutions.

Based on these results, future work could explore the performance benefits of a hybrid approach, where a coarse grid search is first attempted, followed by an exhaustive random search if a valid configuration is not found.

While the current tool demonstrates strong performance and flexibility, several additional areas remain open for further investigation. This includes the evaluation of alternative search algorithms that could achieve faster and more deterministic results.

Additionally, much of the code written for this proof-of-concept design could be optimised to improve performance. With that said, the current implementation shows promising execution times, even on the resource-constrained RFSoC, with median and maximum values below 22 ms and 185 ms, respectively. The former of these meets the requirements for real-time CR applications as set out in Chapter 1, while the latter still requires some improvement.

Chapter 5

A Natively Fixed-Point Run-time Reconfigurable FIR Filter Design Method for FPGA Hardware

5.1 Chapter Overview and Contributions

In Chapter 3 a method of designing reconfigurable SDR systems on RFSoC devices was discussed where, with the use of software drivers, various parameters of the RF-DCs could be controlled, and the results viewed using hardware-based introspection. In that chapter, the radio transceiver design developed was fixed in hardware, meaning that its parameters could not be changed at run-time (with the exception of the gain control at the end of the transmit path). As discussed in Chapter 2 the goal of SDR is to develop a radio in which some, or all, of its parameters can be reconfigured at run-time. Therefore, this chapter presents a novel, natively fixed-point filter design method for FPGA-based RFIR filters. Based on a hybrid between the window and frequency sampling filter design methods, it allows FPGA-based RFIRs to be reconfigured at run-time, directly on the FPGA.

The work in this chapter is based on an original contribution published in the IEEE Open Journal of Circuits and Systems [55], where the original hardware was developed for Zynq devices. To align it with the objectives of this thesis, the algorithm

implementation from Section 5.2 and the results discussed in Section 5.4 have been updated for deployment on the RFSoC.

In summary, the main contributions of this work are as follows:

- An FPGA-based Filter Designer, capable of generating deterministic, fixed-point, linear-phase filter coefficients and frequency responses; suitable for SDR applications.
- A filter design and reconfiguration algorithm that is capable of execution times of 2.52 µs and transition bandwidths less than 1% of the sample rate; suitable for applications with very low latency requirements.
- To the best of this author's knowledge, the only natively fixed-point filter design method developed specifically for FPGA hardware.

This chapter presents a novel fixed-point FIR filter design method based on this hybrid approach that operates entirely on the FPGA, removing the need for external processors or memory to generate coefficients. This method computes coefficients deterministically, at run-time, with reconfiguration times fast enough to be suitable for real-time operation.

5.2 A Hybrid Approach to Filter Design for FPGA Implementation

In Chapter 2 Section 2.4 various filter design techniques were evaluated for their suitability for FPGA implementation compared against three conditions: the ability to produce arbitrary and deterministic frequency responses; suitability for FPGA hardware; and deterministic execution time. While the window and frequency sampling methods were found not to be suitable on their own, in practice it is possible to combine them to create a filter design technique that can satisfy all three conditions, enabling it to be implemented on FPGAs. In this work, this combination of the two methods has been named a *hybrid* approach, although in some literature it is referred to as the Fourier series method [189].

Starting with the frequency sampling method, $H_d(\omega_k)$ is sampled at equally spaced frequencies of length, N_{FFT}, where $\log_2(N_{FFT})$ is an integer, and is symmetric around $N_{FFT}/2$, making the filter have linear phase response. The Inverse Fast Fourier Transform (IFFT) is used to compute the time domain coefficients, then truncated to the desired filter length, N. Finally, the coefficients are multiplied with a window function, of length N, to create the desired filter.

This hybrid approach alleviates the problems each technique has for FPGA implementation. Firstly, because the filter response is now determined in the frequency domain, any arbitrary response can be easily computed where, in the case of a low-pass filter, the sinc function can be written in the form

$$\text{rect}(\frac{\omega_k}{2\omega_c}) = \begin{cases} 0 & \omega_c/2 < \omega_k < \omega_s - \omega_c/2, \\ 1 & \text{otherwise.} \end{cases} \tag{5.1}$$

Secondly, the use of windowing can both smooth the undesirable filter effects caused by the sharp discontinuities in the transition band, allowing for deterministic filter responses, and enable the filter to be truncated to an odd-symmetric length, enabling Type I filters to be produced. Finally, if the filter length, N, is fixed at compile-time, deterministic execution times can be achieved.

This allows the required mathematical operations for the entire algorithm to be reduced to additions, multiplications, and the IFFT algorithm—all suitable for implementation on FPGAs—demonstrating that this hybrid filter design technique is able to satisfy all three conditions.

5.3 Hardware Design

In the original paper [55] this chapter is based on, the Filter Designer was developed for the PYNQ-Z2 development board, which contains the XC7Z020—a mid-range device within AMD's Zynq SoC family of devices. For the purposes of this thesis the hardware and software design have been adapted for the RFSoC4x2 development board containing a third generation RFSoC device, the XCZU48DR. The results given at the end of this chapter are based on this new, RFSoC-targeted work.

This section describes how using the design methodology introduced in Chapter 3, the Filter Designer algorithm was translated to a Model Composer IP, then incorporated into a larger test environment on the RFSoC using Vivado and the PYNQ framework.

5.3.1 IP Design

The Filter Designer algorithm described in the previous section can be broken down into six steps:

1. Calculate frequency response.
2. Generate frequency response.
3. Perform Inverse FFT.
4. Truncate filter to N samples.
5. Perform window operation.
6. Reconfigure RFIR.

The diagram in Figure 5.1 shows a simplified system-level block diagram of the algorithm, which was developed using Model Composer, showing each of the steps. The rest of this section describes each of these blocks in detail.

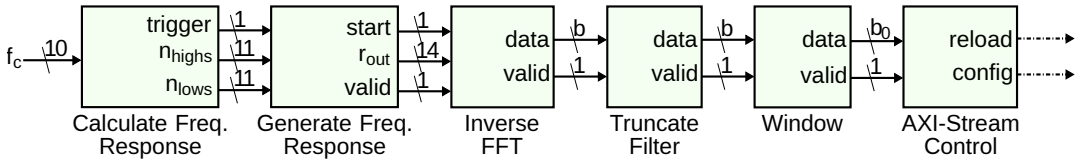

Figure 5.1: Simplified block diagram of the Filter Designer. b is the number of bits dependent on system configuration. b_o is the number of bits constrained at compile-time by the user.

Calculating the Desired Frequency Response

To calculate the desired frequency response of the low-pass filters, two values are required: the cutoff frequency, f_c; and the FFT length, N_{FFT}. Because f_c is a percentage of the normalised sample rate and, at this stage in the algorithm, N_{FFT} is the effective sample

rate, the number of high samples (i.e., the 1 values from Equation 5.1) required for the passband can be calculated as

$$n_{highs} = \|f_c N_{FFT}\|, \tag{5.2}$$

where $\|\|$ denotes rounding to the nearest integer.

As the desired filter is real only (i.e., the coefficients contain no imaginary components), then the frequency response is required to be symmetrical around $N_{FFT}/2$. This means that the total number of high samples required for the entire frequency response is $\|2f_c N_{FFT} - 1\|$. The number of low samples (i.e., the 0 values from Equation 5.1) required for the entire stopband can then be calculated as

$$n_{lows} = N_{FFT} - (2n_{highs} - 1). \tag{5.3}$$

The FFT length, N_{FFT}, is fixed at compile-time, while f_c is taken as an input at run-time. The value of f_c is constrained to a fixed-point wordlength of 10 bits, all of which are fractional. As the resolution of the fractional bits in fixed-point arithmetic is 2^{-b}, the total resolution of the cutoff frequency is approximately 0.1% of the sample rate.

Since the value of N_{FFT} is always a power of two, Equations 5.2 and 5.3 can be calculated with a single multiply and a series of shifts and adds—taking a total of three clock cycles to complete. The output of this step is constrained to an 11-bit wordlength, chosen empirically as the minimum wordlength that consistently produced the best filter performance across a range of cutoff frequencies and FFT lengths.

Generating the Desired Frequency Response

An FSM is used to generate the correct frequency response, containing five states: `IDLE`, `SoF`, `HIGHS`, `LOWS`, and `WAIT`. The FSM has three inputs, n_{highs} and n_{lows} from the previous stage, and a trigger, *Trig*, which pulses high for one clock cycle when the value of f_c changes; and three outputs, *Start*, r_{out}, and *Valid*.

When *Trig* goes high, the `SoF` state is entered, outputting the start of frame signal required by the IFFT IP as well as the initial data and valid signals—automatically moving into the `HIGHS` state after one clock cycle. The `HIGHS` and `LOWS` states are controlled by an internal counter that outputs the appropriate number of high and low samples, as well as the valid signal. After the `LOWS` state has finished, a status flag is set and the `HIGHS` state is entered again to output the remaining high values in order to make the frequency response symmetrical. The `HIGHS` state exits to the `WAIT` state when the status flag is set. The `WAIT` state is used to ensure that the trigger has reset before generating a new frequency response, moving to the `IDLE` state when this condition has been satisfied. Once entered, the `IDLE` state resets all counters and flags and awaits the next trigger event.

A state diagram of the frequency response generator FSM is shown in Figure 5.2, while an example output of the FSM is shown in Figure 5.3. The output of this step is constrained to a 14-bit wordlength.

Inverse FFT

The next step in the algorithm is the inverse FFT, which transforms the filter frequency response into the time domain impulse response. This is performed by the LogiCORE Fast Fourier Transform v9.1 IP block, which is distributed by AMD and available in Model Composer. While this step can be performed by any FPGA-based FFT algorithm, the LogiCORE IP was chosen as it is designed and optimised specifically for AMD FPGAs.

The FFT IP block follows the AXI-Stream standard, requiring *tdata*, *tvalid*, and *tlast* input signals; while providing the same signals on the output. The input *tlast* signal is used to signify the end of an FFT frame, in this case provided by a size N_{FFT}-delayed version of *Start* from the previous stage. The FFT IP is the primary source of latency in the signal path and is directly proportional to the size of N_{FFT}.

As the signal from the previous stage is entirely real and symmetrical the imaginary inputs and outputs can be ignored. It should be noted that, because of quantisation errors, there will be a low amplitude signal present in the imaginary output port. The

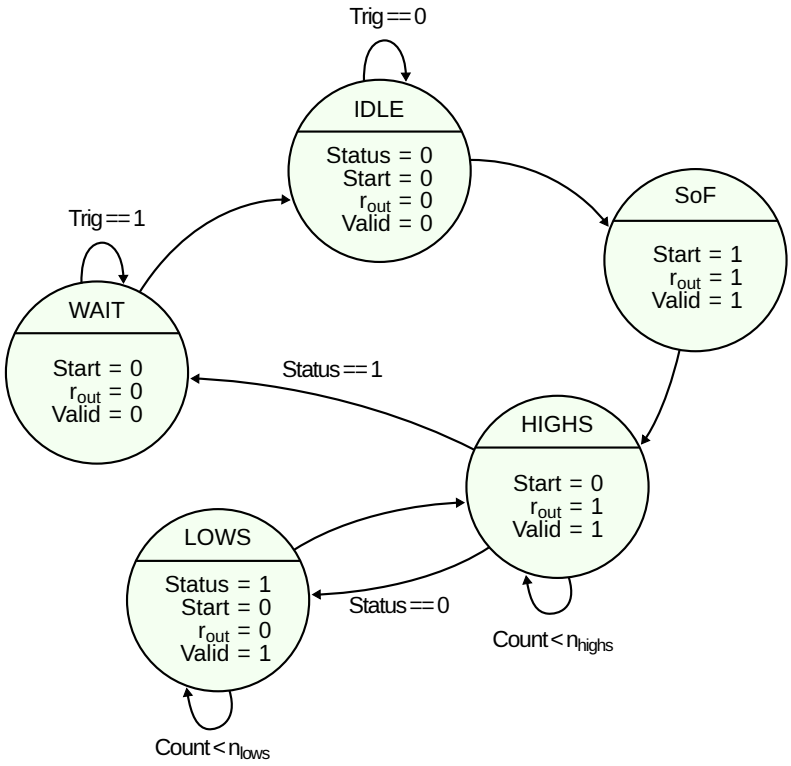

Figure 5.2: State diagram for the frequency response generator FSM.

output wordlength of this step is left unconstrained and is dependent on both the size of N_{FFT} and the FFT IP configuration. For example, using the default IP settings, and with N_{FFT}=1024, the output wordlength is 25 bits.

Truncating to an N-tap Filter

The N_{FFT}-length impulse response output by the FFT is stored in a single-port RAM, implemented using BRAM. As the FFT IP outputs the samples in a *bit-reversed* order, the samples are first stored in the *natural* order before being truncated. Bit reversal simply means that the bits representing the indices of the samples are reversed—not the samples themselves. For example, for an 8-length FFT, each sample index can be represented by 3 bits ($2^3 = 8$). In *bit-reversed* order, the first sample out of the FFT (sample 001_2) is actually the fourth sample (sample 100_2); the second sample (010_2)

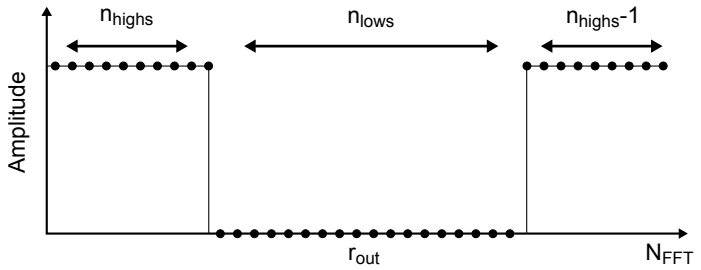

Figure 5.3: Example output, r_{out}, of the frequency response generator FSM.

remains the second sample; the third sample (011_2) is actually the sixth sample (110_2); and so on.

A ROM containing the *bit-reversed* indexes, and attached to the address line of the RAM, is used to write each sample of the FFT output to the correct (*natural* order) memory address. This process allows the N-length filter to be more easily retrieved from the RAM. It should be noted that, while the LogiCORE FFT IP does allow for the sample indexes to be output in the *natural* order, it was found during development that it added much more latency to the output than the method described above.

As Type I filters are symmetrical around their centre tap, only $M = (N + 1)/2$ coefficients are required to accurately replicate them. The M filter coefficients are retrieved from the RAM with the use of an FSM containing three states: `IDLE`, `WAIT`, and `ADDR`. The `WAIT` state is entered when a filter has started loading into the RAM, signified by the *tvalid* signal going high. When the entire N_{FFT}-length filter has been written to RAM, the valid signal goes low and the `ADDR` state is entered, where a counter, initialised at $N_{FFT} - M$, is used to read out the M coefficients from the RAM. The `IDLE` state is used to loop until a new filter is stored in the RAM. This step introduces a delay of $N_{FFT} + 3$ clock cycles. A state diagram of the FSM is shown in Figure 5.4.

Windowing the Filter

At this stage, the M-length filter is multiplied by a window function, which is stored in a ROM and read out using a counter. The window can be of any type, with only M coefficients required to be stored (i.e., half the window function). The multiply

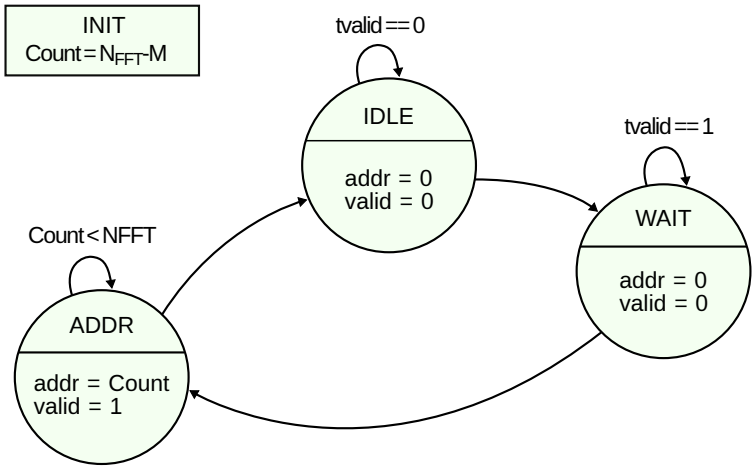

Figure 5.4: State diagram for RAM control FSM.

operation adds a 1-cycle delay, and the wordlength of the signal is constrained at the output.

Reloading the Filter Coefficients

The LogiCORE FIR Compiler IP used in this work requires two AXI-Stream channels to reconfigure the filter: the *Reload* channel, used to download the coefficients into memory; and the *Config* channel, used to prompt the filter to start using the new coefficients. The *Reload* channel requires three signals: *tdata*, *tvalid*, and *tlast*. The *tdata* signal contains the filter coefficients, while *tlast* is used to signify the end of the M-length filter. Once the reload signal has finished, an empty, 8-bit packet is sent to the *Config* channel, along with a *tvalid* signal, for a single cycle. Thus, the entire reprogramming of the filter, using the LogiCORE FIR Compiler IP, requires $M+1$ cycles to complete. A timing diagram of this entire reconfiguration handshake is shown in Figure 5.5.

This handshake is performed by a FSM containing four states: `IDLE`, `VALID`, `LAST`, and `LOAD`. The `VALID` state is entered when the input *tvalid* line goes high, outputting the *Reload* channel *tdata* and *tvalid* AXI-Stream signals for $M-1$ samples (controlled by an internal counter). The `LAST` and `LOAD` states only operate for a single cycle, where they output the final *tlast* signal of the *Reload* channel, and the required *Config* channel

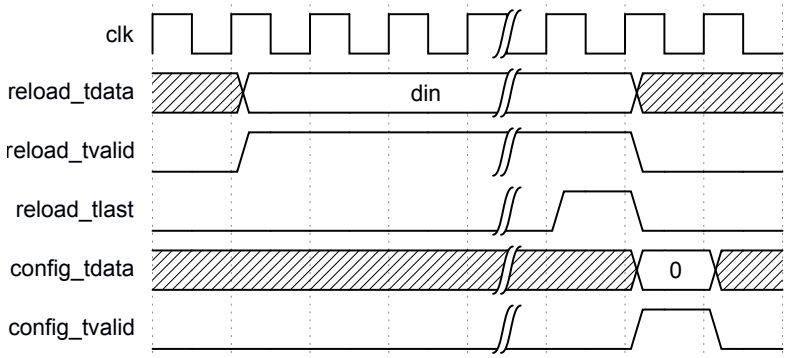

Figure 5.5: AXI-Stream timing diagram for the LogiCORE FIR Compiler.

empty packet, respectively. The FSM then stays in the `IDLE` state until a new *tvalid* signal is detected. A state diagram of the FSM used in this stage is shown in Figure 5.6.

5.3.2 System Design

In order to determine that the Filter Designer operated correctly, and to acquire specific results, it was necessary to incorporate the HDL IP created in Model Composer into a larger SoC design. This section describes both the Vivado hardware design required for the PL, and the software design required for the PS, in order to build this test environment.

Vivado Design

The hardware design was developed in the Vivado IDE, and based around the Filter Designer IP generated in Model Composer. The Filter Designer's AXI-Stream compliant *Reload* and *Config* Master ports were connected to the respective Slave ports of the LogiCORE FIR Compiler, while its memory-mapped port was connected to an AXI Interconnect IP. Only a single input is required, f_c, which is controlled by a 32-bit AXI-Lite register.

AXI-Stream data is passed between the RFIR on the PL and the PS via shared off-chip memory, arbitrated by an AXI DMA controller IP. The FIR Compiler's Slave and Master AXI-Stream data ports are connected to the respective ports on the DMA

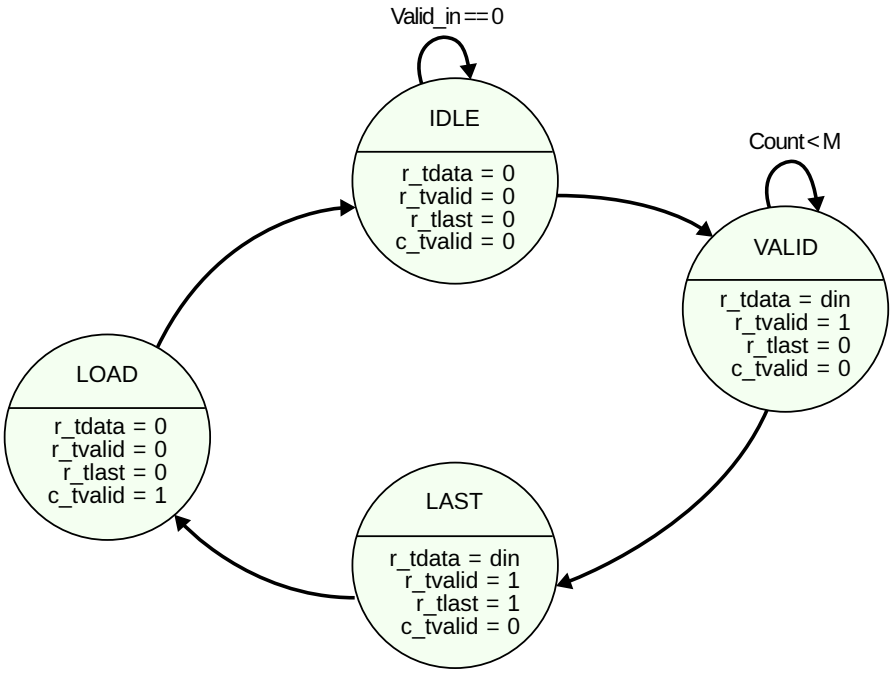

Figure 5.6: State diagram for the AXI-Stream signal control.

IP, while the DMA's memory-mapped port is connected to the AXI Interconnect IP. The DMA was set up in read/write mode in order to both send *and* receive data.

The Zynq UltraScale+ PS IP was configured with one General Purpose (GP) Master port, and one High Performance (HP) Slave port, while a single PL clock was used for all IPs. Two clock frequencies were used while collecting result data, 100 MHz and 250 MHz. A simplified block diagram of the Vivado IPI diagram is shown in Figure 5.7.

Software Design

As the Filter Designer IP has only a single input, controlled by an AXI-Lite register, the software required to operate it is simple. However, when putting the Filter Designer IP into the larger test environment—as described in the previous section—the software requirements become more complex. For this reason, the software for the design was developed within the PYNQ framework.

As discussed in Chapters 2 and 3, PYNQ provides built-in drivers and APIs for various IPs, as well as providing the means to create your own drivers for custom IPs.

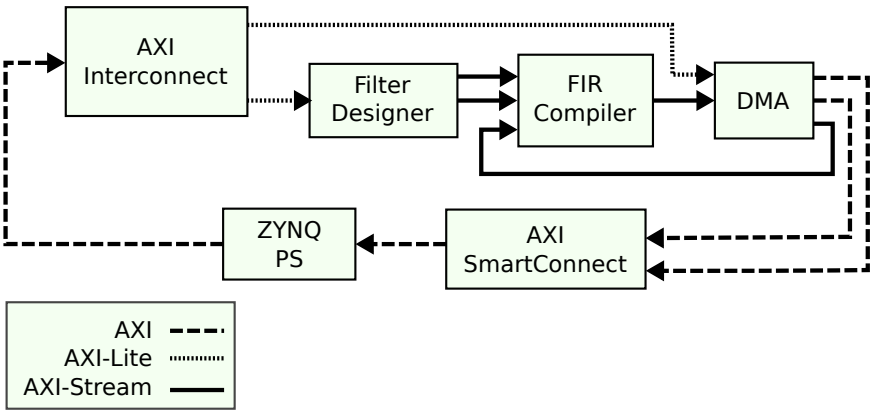

Figure 5.7: Simplified Vivado IPI block diagram for the test environment.

As the only requirement to initiate a filter on the Filter Design IP is to write the cutoff frequency to an AXI-Lite register, then only the PYNQ MMIO module is required.

In order for data to be sent to the RFIR and the filtered data to be received back again, the PYNQ DMA module is used. This requires a Contiguous Memory Array (CMA) to be created, then filled with the data to be sent to the filter via shared memory. An equal-sized CMA is also required to accept the received data, again, through shared memory. This setup allowed test data to be easily sent to, and received from, the FIR Compiler in order to confirm that the filter had been correctly reconfigured.

The PYNQ framework was additionally used to generate the results in Section 5.4.6, where the execution times between the filters computed entirely in fixed-point arithmetic using the Filter Designer IP (i.e., natively), and fixed-point filters that are initially designed using floating-point arithmetic and are later converted to a finite wordlength (i.e., non-natively). This distinction between *native* and *non-native* methods of designing filters is continued throughout the rest of this chapter.

5.4 Results

In this section a number of filters designed by the Filter Designer are discussed, the effect that wordlength restriction and FFT size have on filter quality is determined, and a comparison is made between these filters and filters designed non-natively. Additionally,

the execution time of the Filter Designer is investigated, and similar comparisons are made between native and non-native implementations.

For the purposes of this work, filter quality is determined by stopband attenuation and transition bandwidth, where the former is measured at the -3 dB point at the end of the mainlobe; and the latter is measured at the highest sidelobe in the stopband (see Figure 2.15). To allow the data collected to be viewed more clearly, stopband attenuation measurements are displayed as a second-order polynomial fit, where the general trend of the data is shown, rather than the exact data itself, and at least some deviation from this trend should be expected. An example of this is shown in Figure 5.8, where the actual captured data is displayed alongside the trendline of the graph—in the rest of this results section, only the trendlines are shown. Any measurements shown between data points derive from the actual data and not from the trend. To reiterate this point, the trendlines are used only as a means for the reader to better understand the data, not to display the measurements taken.

All results in this section were retrieved in simulation, except where data from hardware was necessary (e.g., execution times, hardware utilisation etc.). A subset of simulated results were confirmed in hardware and found to be equivalent.

5.4.1 Effects of Wordlength Constraint on Filter Quality

Throughout the signal path of the Filter Designer the wordlength is left unrestricted, except at the output where it can be constrained by the user at compile-time. To understand what effect constraining the output wordlength has on filter quality, stopband attenuation and transition bandwidth were measured over a range of filter lengths for six different output wordlengths, ranging from 16 bits to 32 bits. For these results, N_{FFT} and f_c were fixed at 1024 and 0.33, respectively.

As can be seen from Figure 5.9, at shorter wordlengths, somewhat counter-intuitively, stopband performance worsens as the filter length increases. This effect is due to the fact that quantisation errors are accumulative. Thus, the more coefficients there are, the more quantisation errors there will be, therefore affecting the filter performance. This effect can be counteracted by increasing the wordlength, where better performance

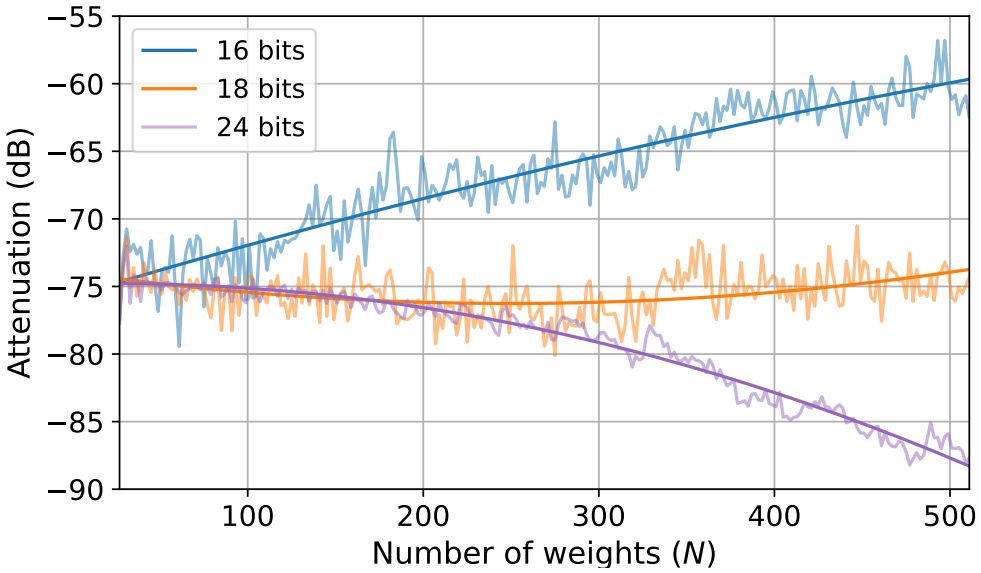

Figure 5.8: Recorded data overlaid with trendlines. The data shows stopband attenuation over N with respect to wordlength constraint. $N_{FFT} = 1024$, $f_c = 0.33$.

in the stopband is seen as N and wordlength increase—displaying stopband attenuation as high as 88 dB—with no increase in performance seen beyond 24 bits.

The results in Figure 5.10 show that wordlength has little effect on transition bandwidth, displaying a similar downward trend as N increases for all wordlengths tested—typical for the Blackman window used. The smallest transition bandwidth measured in these results was 0.68% of the sample rate.

From these results it is clear that, although longer wordlengths are required to achieve the best performance in the stopband, this is only the case for large values of N and only up to 24 bits, where the benefits of increasing the wordlength plateau. For all wordlengths tested between 18 bits and 32 bits, the stopband performance is almost identical for $N < 200$. For wordlengths between 20 bits and 32 bits, similar stopband performance for $N < 350$ can be seen. This trend continues with only a few dB difference in stopband performance between 21 bits and 24 bits at $N = 500$. With that said, in order to obtain the best performance, the remaining results in this section use an output wordlength of 24 bits.

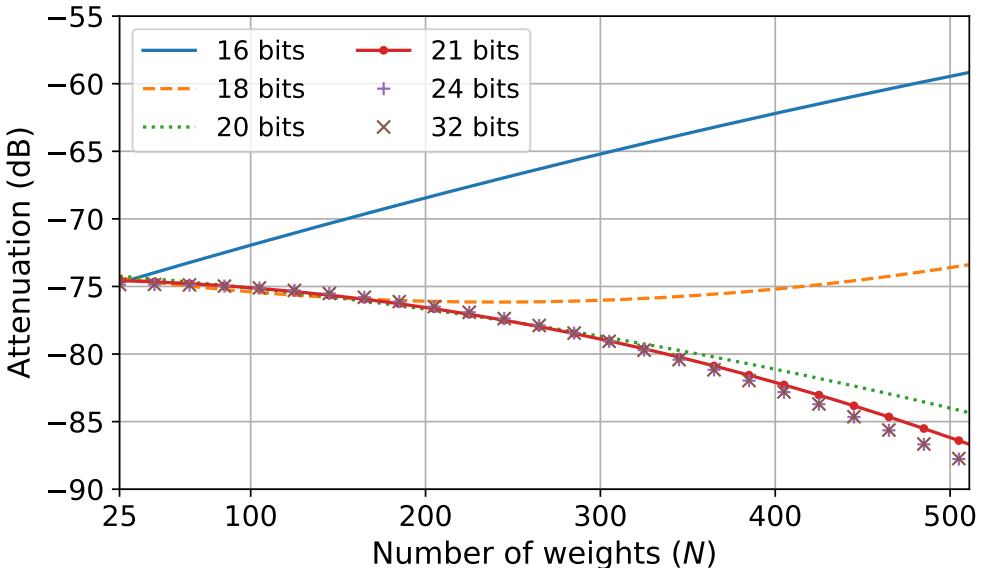

Figure 5.9: Trendlines of stopband attenuation over N with respect to wordlength constraint. $N_{FFT} = 1024$, $f_c = 0.33$

5.4.2 Effects of $N_{\mathbf{FFT}}$ on Filter Quality

As the most resource-heavy component of the Filter Designer is the IFFT, it is worth exploring how FFT length affects filter quality, in order to minimise resources while still achieving the best possible filter performance. Results were obtained by measuring stopband attenuation and transition bandwidth over a range of filter lengths for five values of N_{FFT}. It is known from [165] that, when using the frequency sampling method, filter quality degrades when the FFT length is less than twice that of the filter length. For this reason, it is worth framing the results based on the ratio between the two (i.e., N/N_{FFT}), and to limit the maximum value of N to half that of each value of N_{FFT} used.

By using this N/N_{FFT} ratio, Figure 5.11 shows that very little performance is gained in the stopband by increasing the FFT length and, in fact, performance plateaus at $N_{FFT} = 1024$, with the highest attenuation measured at 88 dB. If this result were to be taken alone it would seem unnecessary to use longer FFTs, as this would increase hardware resources for no obvious gain in performance. However, when the results

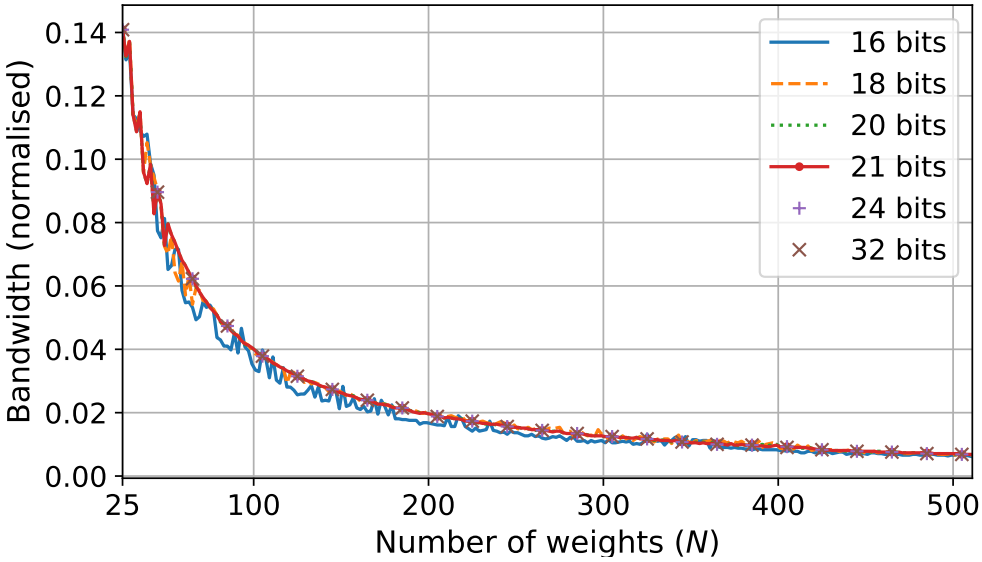

Figure 5.10: Transition bandwidth over N with respect to wordlength constraint. $N_{FFT} = 1024$, $f_c = 0.33$

from Figure 5.12 are taken into account, it can be seen that, to achieve the narrowest transition bandwidths, longer FFT lengths are still required.

Results from Figures 5.11 and 5.12 were measured with a fixed value of f_c. It is worth noting though that stopband attenuation is not constant for all possible values of f_c. To determine the effect f_c had on filter quality, stopband attenuation was measured again with 25 values of f_c taken for each value of N. From this data the mean and standard deviation were calculated, as shown in Figure 5.13. These results show that deviation from the mean is much larger for shorter FFT lengths, further demonstrating that larger FFT lengths are required to keep filter quality consistent.

5.4.3 Comparison Between Native and Non-Native Coefficients

When considering the quality of natively designed filters, it is worth comparing them to filters designed by non-native means. It is, of course, expected that a filter designed in floating point would, given the same parameters, outperform one designed entirely in fixed-point. However, it is beneficial to examine how much they diverge when considering the trade-offs between them.

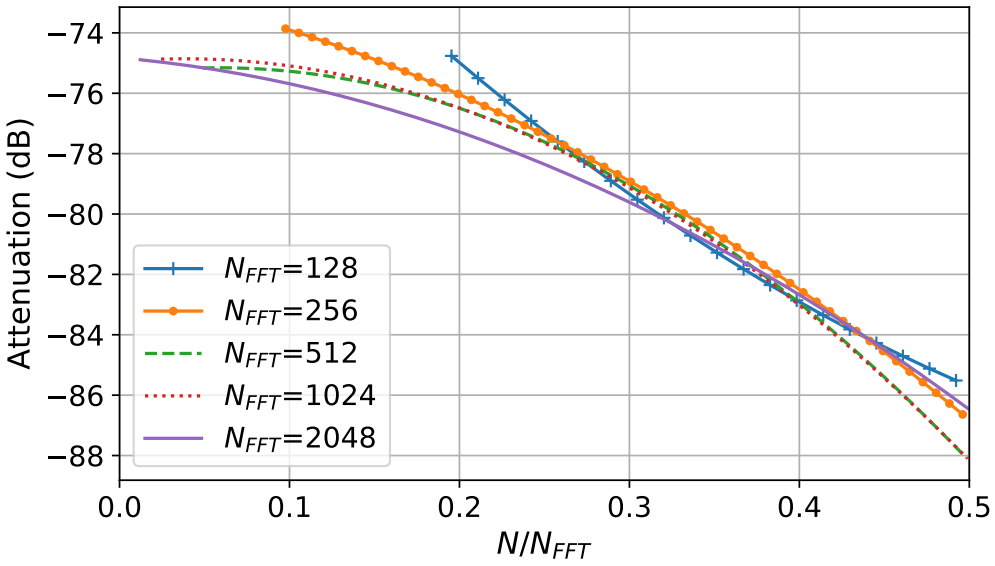

Figure 5.11: Trendlines of stopband attenuation over N/N_{FFT} with respect to N_{FFT}. f_c=0.33, wordlength=24.

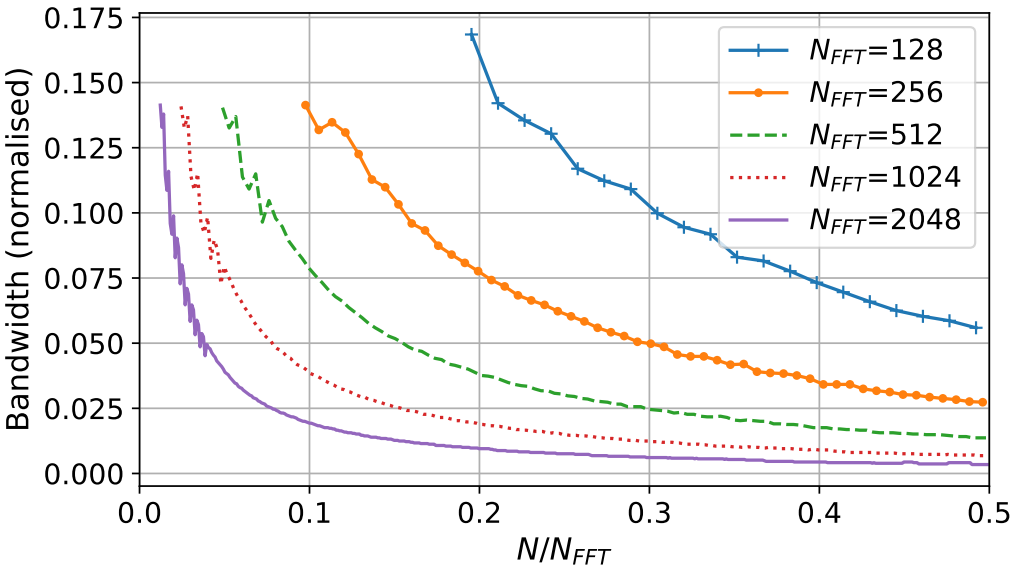

Figure 5.12: Transition bandwidth over N/N_{FFT} with respect to N_{FFT}. f_c=0.33, wordlength=24.

To make this comparison, the MATLAB filter design function `fir2` was chosen as it uses an algorithm similar to the Filter Designer described in this chapter, albeit using

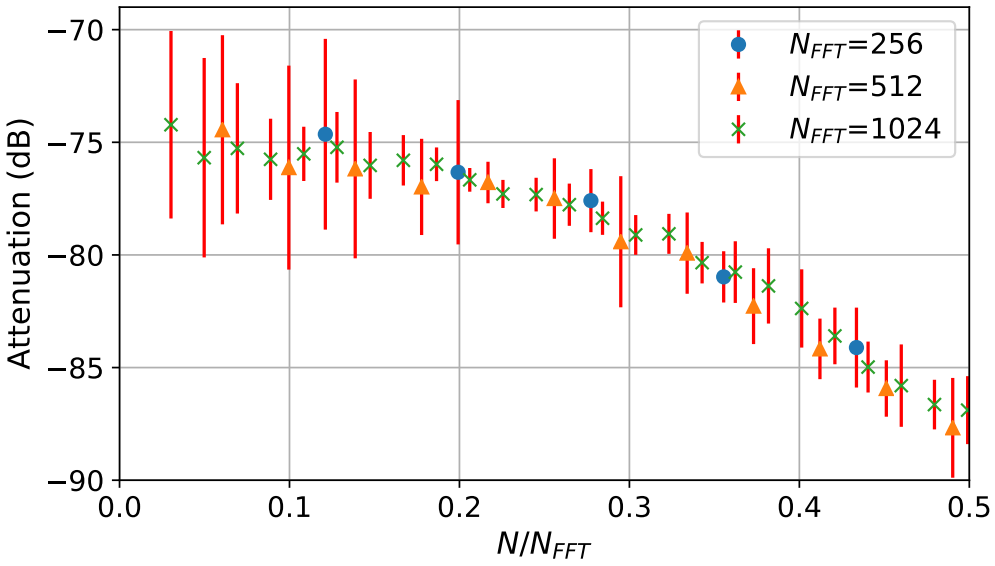

Figure 5.13: Mean (symbol) and standard deviation (error bar) of stopband attenuation over N/N_{FFT} with respect to N/N_{FFT}. Each data point uses 25 values of f_c to calculate the mean. Wordlength=24.

floating point arithmetic. The `fir2` function has some additional functionality that allows for constraining the transition band, which typically comes at the detriment of stopband performance. For this reason the native filters are compared to the non-native filters designed both without and with these constraints (denoted as A and B, respectively).

If the performance in the stopband is compared between the native filters and the unconstrained non-native filters, as shown in Figure 5.14, it can be seen that the non-native filters greatly out-perform the native equivalents, with up to 30 dB difference at its maximum. However, when comparing the transition bandwidth of the same filters, as shown in Figure 5.15, the native filters out-perform the non-native filters—albeit not to the same extent.

The reason why the native filter performs better in the transition band in this scenario is due to the implementation of `fir2` where, in this case, the transition band was left unconstrained. However, if the transition band of the non-native filter is constrained such that it performs similarly to the native filters, as shown by the *Non-*

Native (A) curve in Figure 5.15, the results from Figure 5.14 show that the difference in stopband attenuation between native and non-native filters is reduced dramatically; from 30 dB at its maximum down to 9 dB. This demonstrates that native filters can approach the quality of non-native filters under certain conditions.

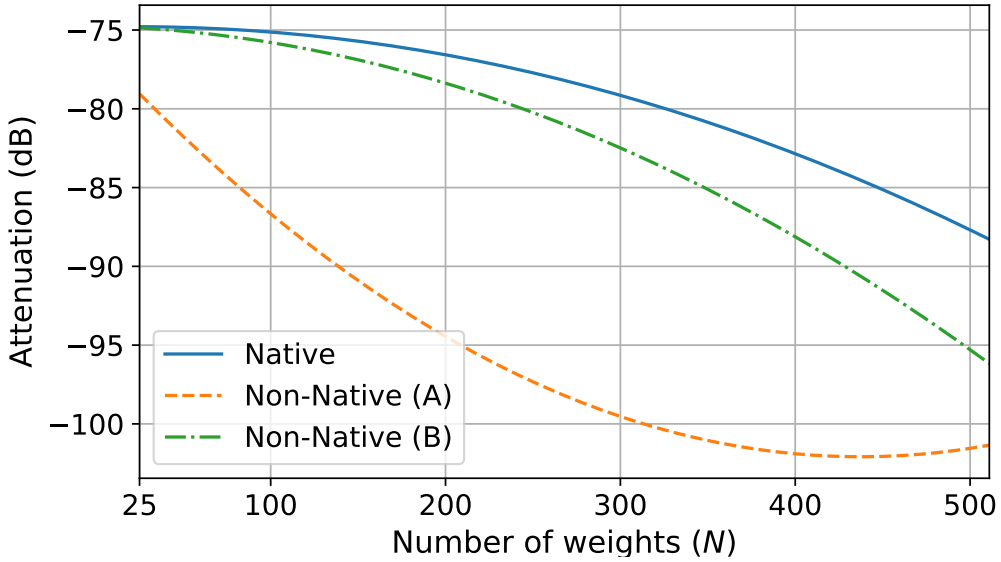

Figure 5.14: Trendline comparison of stopband attenuation over N between native filters and two variants of non-native fixed-point filters. N_{FFT}=1024, f_c=0.33, wordlength=24.

5.4.4 Effects of N and $N_{\mathbf{FFT}}$ on FPGA Resources

Along with a single run-time configurable parameter, the Filter Designer also has three compile-time configurable parameters: the filter length, N; the FFT length, N_{FFT}; and the wordlength. As wordlength is only constrained at the output of the Filter Designer, it is worth exploring how N and N_{FFT} affect the FPGA resources on the XCZU48DR RFSoC device. Three FFT lengths were used; 256, 512, and 1024; and for each value of N_{FFT}, three filter lengths were used: 10%, 20%, and 30% of N_{FFT} (to the nearest odd value).

As can be seen from Table 5.1, N has very little effect on hardware resources, with the changes in LUTs most likely due to optimisations made by the compiler. This is because, for most of the signal path, the filter length is equal to the FFT length, only

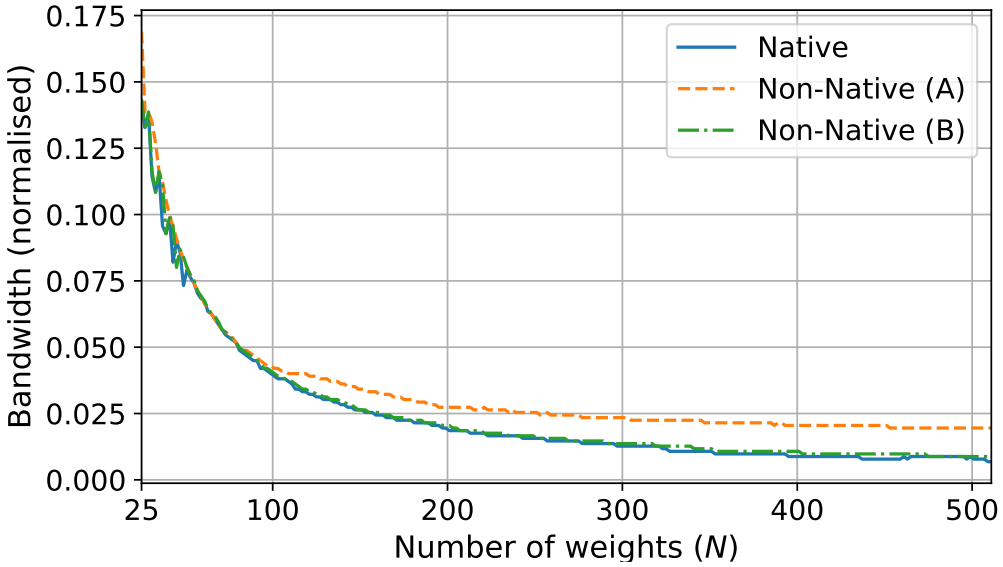

Figure 5.15: Comparison of transition bandwidth over N between native filters and two variants of non-native fixed-point filters. N_{FFT}=1024, f_c=0.33, wordlength=24.

being truncated before the window operation. For every increase in N_{FFT}, around 400 more LUTs are used, with a small increase in DSPs and BRAMs also seen due to more samples being be stored in the single-port RAM IPs and the FFT IP.

With the RFSoC having considerably more resources than the device used in [55], the resource utilisation is particularly low—using as little as 0.39% of LUTs, 0.28% of BRAMs, and 0.26% of DSPs—leaving plenty of space for additional logic on the hardware.

5.4.5 Effects of N and N_{FFT} on Execution Time

Using the same hardware implementation as the previous section, measurements were taken to determine how N and N_{FFT} affected execution time. A simple HDL counter was used to measure the number of clock cycles between writing to the f_c AXI-Lite register and observing the *tvalid* signal of the *Config* port going low—signifying that the reconfiguration of the RFIR was complete. Two clock frequencies were used: 100 MHz, and 250 MHz.

Table 5.1: FPGA utilisation for varying values of N and N_{FFT}. Wordlength=24 bits.

N	N/N_{FFT}	LUTs	BRAM	DSPs
		N_{FFT}=256		
25	0.1	1655 (0.39%)	3 (0.28%)	11 (0.26%)
51	0.2	1656 (0.39%)	3 (0.28%)	11 (0.26%)
101	0.3	1657 (0.39%)	3 (0.28%)	11 (0.26%)
		N_{FFT}=512		
51	0.1	2025 (0.48%)	3 (0.28%)	14 (0.33%)
153	0.2	2028 (0.48%)	3 (0.28%)	14 (0.33%)
205	0.3	2027 (0.48%)	3 (0.28%)	14 (0.33%)
		N_{FFT}=1024		
101	0.1	2462 (0.58%)	3.5 (0.32%)	14 (0.33%)
307	0.2	2467 (0.58%)	3.5 (0.32%)	14 (0.33%)
409	0.3	2467 (0.58%)	3.5 (0.32%)	14 (0.33%)

As can be seen from Table 5.2, execution times increase almost linearly for changes in both N and N_{FFT}. This result is to be expected as latency within the signal path is directly proportional the filter length. The reason why there is not an exact doubling between increases in N_{FFT} is due to optimisations in the FFT algorithm which allow it, to a certain extent, to minimise the amount of calculations needed.

These execution times are especially short when compared to software methods of reconfiguring RFIRs, discussed in the following sections, measuring as little as 2.52 µs for a clock rate of 250MHz; removing any bottleneck to other parts of the radio.

5.4.6 Comparing Native and Non-Native Execution Times

In order to meaningfully compare execution times between native and non-native design methods, a system is assumed where both are controlled by the PS on the RFSoC. The native design method requires an AXI-Lite register to be written to on the PS, while the rest of the process is performed on the PL. For the non-native design method, filter coefficients must be calculated on the PS, then transferred to the PL via shared memory. To perform the AXI-Stream handshake required by the FIR Compiler (as described in Section 5.3.1), two AXI DMAs are used: one for the *Reload* channel and

Table 5.2: Execution time of Filter Designer with respect to N and N_{FFT}. Wordlength=24 bits.

N	N/N_{FFT}	Cycles	$t(\mu s)$ @ 100 MHz	$t(\mu s)$ @ 250 MHz
		N_{FFT}=256		
25	0.1	630	6.3	2.52
51	0.2	643	6.43	2.57
101	0.3	668	6.68	2.67
		N_{FFT}=512		
51	0.1	1164	11.64	4.66
153	0.2	1215	12.15	4.86
205	0.3	1241	12.41	4.96
		N_{FFT}=1024		
101	0.1	2219	22.19	8.88
307	0.2	2322	23.22	9.29
409	0.3	2373	23.73	9.49

the other for the *Config* channel. PYNQ is the assumed target software framework. The steps required to reload the FPGA-based RFIR using non-native filter coefficients are enumerated below:

1. Calculate filter coefficients for a given set of parameters.
2. Convert floating point coefficients to integer values.
3. Copy coefficients to *Reload* channel DMA buffer.
4. Initiate *Reload* channel DMA transfer.
5. Copy reconfiguration packet to *Config* channel DMA buffer.
6. Initiate *Config* channel DMA transfer.

These steps were separated into two functions: `design_filter`, which performs steps 1 and 2 on the list; and `reload_filter`, which performs the remaining steps. The following sections describe how the execution times of the non-native filter design method were measured, and compare the results to those from Section 5.4.5.

Measuring DMA Transaction Time

To perform the AXI-Stream handshake needed to reconfigure the FIR Compiler, two AXI-DMAs are required: one for the *Reload* channel and the other for the *Config* channel. To transfer data using the PYNQ DMA class, it is necessary to create a CMA to store the data, copy the data over to the array, initiate the DMA transfer, and wait for the transfer to complete. As two DMAs are required to reconfigure the RFIR, this wait time between DMA transfers can be a significant bottleneck.

In order to measure the time taken to perform the steps in the `reload_filter` function, the built-in Python function `timeit` was used. This function runs selected code over a large number of iterations and outputs the mean and standard deviation. Using this method, execution times of 1.30 ms and 1.38 ms were measured for N=51 and N=101, respectively. This is approximately 50% faster than the results measured in [55], which used a Zynq SoC device containing the Arm Cortex A9, compared to the Arm Cortex A53 on the RFSoC.

To measure the DMA transaction time in hardware, a small HDL counter IP was created and used to measure the number of clock cycles between when the *tvalid* signal of the *Reload* DMA channel went high and the *tvalid* signal of the *Config* channel went low, signifying that both DMA transactions had completed. As this measurement had to be recorded manually only 20 readings were taken, where a mean of 309 µs and 333 µs were recorded for N=51 and N=101, respectively, at a clock rate of 100 MHz. These results are shown in Table 5.3.

Table 5.3: Hardware-measured execution time of `reload_filter` function. Wordlength=32 bits, clock=100 MHz, sample size=20.

N	Mean (Cycles)	Mean (µs)	σ (µs)
51	30,861	309	24.28
101	33,266	333	85.89

The discrepancy between the hardware results from Table 5.2 and the software results shown in Table 5.3, arises because the time taken to create and fill the DMA buffers is not recorded in the hardware results. This signifies that the use of DMAs can

create a bottleneck. Additionally, the results were found to be non-deterministic, with a standard deviation, σ, of 24.28 µs and 85.89 µs for N=51 and N=101, respectively.

This non-deterministic behaviour is due to the overhead of the Linux operating system running on the PS, which gives no guarantees on when specific operations will be executed, or even if two operations will happen consecutively. This could be mitigated with the use of a real-time kernel or operating system, or even by removing the operating system entirely with the use of baremetal code. However, doing so inevitably leads to more complex design specifications and longer development time.

Measuring Software Filter Design Time

In the previous section, only the time taken to reconfigure the filter was taken into account; omitting the time taken to design the filter and convert it to integer values—all performed in the `design_filter` function. To design the filter, the `firwin2` function (part of Python's *SciPy* library) was used. `firwin2` uses the window method to design filters, and was chosen due to the minimal input parameters it requires, and its relative speed compared to other design methods available in the same library. Moreover, it is similar in functionality to the MATLAB function, `fir2`, discussed in Section 5.4.3.

Using `timeit` again, execution times of 1.71 ms and 2.41 ms for N=51 and N=101, respectively were recorded for the `design_filter` function on its own. When both `design_filter` and `reload_filter` functions were timed together, execution times of 3.25 ms and 4.03 ms were recorded for $N = 51$ and $N = 101$, respectively—three orders of magnitude slower than the Filter Designer results recorded in Table 5.2.

As mentioned previously, these long execution times could be the result of the Linux overhead. To test this hypothesis, a basic implementation of the window method was developed and run as a baremetal program on the Arm processor, and was found to have similar results. The best time over 1000 iterations was recorded as 0.8 ms and 3.14 ms for N=51 and N=101, respectively.

These results show that the time taken to design and reconfigure an RFIR using the native method is three orders of magnitude faster than attempts to do the same

using non-native means. A comparison of the execution times recorded using the native method and those recorded for the non-native methods are shown in Figure 5.16.

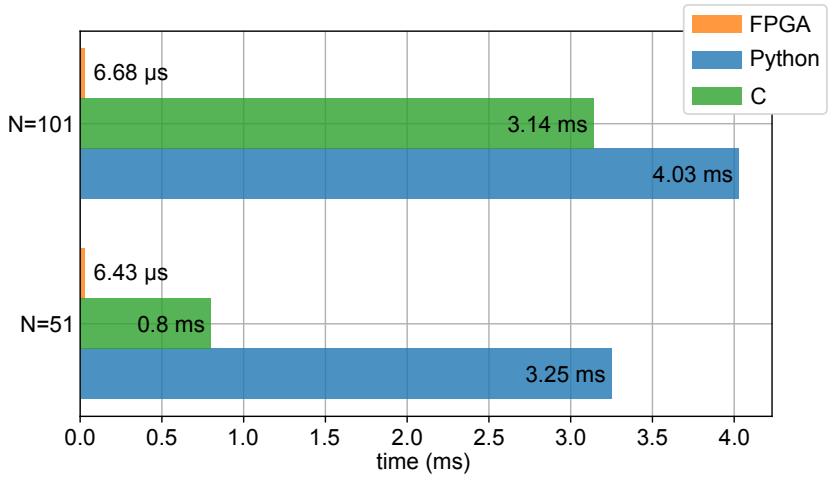

Figure 5.16: Comparison between software (Python and C) and hardware (FPGA) execution times of the filter design algorithms. The Python software times are taken from the `timeit` results, which include both the `design_filter` and `reload_filter` functions. The C software time only includes the execution time of designing the filter. The *FPGA* times are taken from Table 5.2, where $N_{FFT} = 256$, with a system clock rate of 100 MHz.

5.5 Concluding Remarks

This chapter has presented a novel FPGA-based design methodology for reconfigurable FIR filters, operating natively on the FPGA fabric at run-time. The motivation for this work was established in Chapter 4, where RFIRs were shown to be essential for real-time frequency planning in fully autonomous systems, such as DSA and CR.

A range of optimal and suboptimal filter design methods were evaluated against three conditions to assess their suitability for FPGA implementation. This led to a hybrid approach that combines both frequency sampling and window methods to compute fixed-point filter coefficients entirely on-chip. By generating coefficients directly on the FPGA, this method avoids the latency and design complexity introduced by off-chip memory and processor communication—removing a key bottleneck in reconfigurable systems.

An extensive set of results was obtained, demonstrating the algorithm's ability to produce high-quality filter coefficients—achieving stopband attenuation up to 88 dB and transition bandwidths less than 1% of the sample rate. When compared with non-native approaches, the proposed method achieved execution times as short as 2.52 µs—over three orders of magnitude faster—while offering the deterministic behaviour required for real-time applications. Importantly, under certain conditions, the native fixed-point method was shown to approach the quality of the non-native equivalent.

As discussed at the start of this chapter, deterministic, on-the-fly filter design has obvious benefits for SDR applications, where faster reconfiguration improves overall throughput of the system. While some constraints remain, such as fixed compile-time parameters for filter length and wordlength, the ability to generate filters deterministically on the FPGA opens up possibilities for a new class of low-latency, reconfigurable SDR platforms, capable of supporting next generation wireless systems.

Chapter 6

Conclusions

6.1 Résumé

In summary, this thesis has presented a design methodology and algorithms for the development of SDR and CR applications for RF-sampling SoCs, namely the RFSoC. The work aimed to address the issues surrounding spectrum scarcity through two main objectives. The first was to develop a methodology that enables the creation of fast and efficient algorithms, while remaining as accessible as possible—achieved in part by the release of each project as open-source. The second was to develop algorithms focused on reconfigurability, determinism, and productivity. The following chapter summaries demonstrate how these objectives were met through theoretical and practical contributions.

In **Chapter 2**, a background and literature review was provided to establish the context for the research. This chapter explored the evolution of SDR and its role as a key enabler for CR and DSA. It detailed the internal architecture of the RFSoC, highlighting how its integration of high-speed RF-DCs and programmable logic addresses the hardware requirements of modern radio systems. Finally, the chapter identified critical gaps in existing workflows regarding design productivity, real-time frequency planning, and deterministic filter reconfiguration, which served as the motivation for the subsequent technical chapters.

In **Chapter 3**, a design methodology was developed for the RFSoC, using Model Composer for custom FPGA IP design and PYNQ as the embedded run-time environment. The use of heterogeneous versus monolithic IP design methods was explored to determine their impact on driver development. Additionally, PYNQ was extended to support the RFSoC, facilitating the development of drivers for both integrated hardware peripherals and custom FPGA designs. To evaluate this methodology, a reconfigurable QPSK transceiver was implemented on the RFSoC, leveraging the PS, PL, and RF-DCs. This design included signal inspection and visualization logic for debugging, as well as the capability to control and reconfigure the RF data converters via software running on the PS.

In **Chapter 4** a run-time reconfigurable frequency planning tool, SPECTRE, was developed—capable of real-time operation on the RFSoC. A scriptable tool was introduced, capable of identifying spur-free regions of the spectrum based on input parameters, such as centre frequency and bandwidth. The tool was extended to support autonomous configuration of the RF-DCs on the RFSoC—enabling CR applications. A range of simple search algorithms were implemented and evaluated in terms of success rate, computational efficiency, and performance.

In **Chapter 5** a novel, natively fixed-point filter design method was presented, specifically targeting FPGA-based reconfigurable FIR filters, and implemented on a RFSoC device. The algorithm developed uses a hybrid of window and frequency sampling filter design methods, capable of generating deterministic, fixed-point coefficients natively on the FPGA. The algorithm was developed in Model Composer, and tested on RFSoC hardware using the PYNQ framework for verification. Additionally, a comprehensive analysis of the coefficients generated by this filter design method was given, acting as a design guide for engineers.

6.2 Summary of Results

This section summarises the results of the work presented in this thesis, highlighting the key findings from each chapter.

The design methodology presented in **Chapter 3** demonstrated that complex SDR systems can be rapidly prototyped on the RFSoC using a combination of Model Composer and the PYNQ framework. By implementing a fully functional transceiver, the work established that signal control and visualisation can be achieved entirely on a single device, eliminating the need for external instrumentation. A comparison of monolithic and modular IP design strategies highlighted that a modular approach results in significantly more maintainable software drivers. Furthermore, the results validated the use of Python for real-time interaction, achieving visualisation rates of 20 FPS with negligible resource overhead. This work established PYNQ as a viable framework for RFSoC development, facilitating subsequent applications in fields ranging from radio astronomy to quantum computing.

The frequency planning tool, SPECTRE, developed in **Chapter 4** addressed the issue of identifying spur-free regions of the spectrum in real time. Three search strategies were evaluated: ordered, random, and coarse grid. The results showed that, while including all possible spurs ensured spectral integrity, it made it more difficult to find valid frequency plans at high bandwidths—with success rates dropping to below 40% at 100 MHz bandwidths. The random search method provided the best performance in terms of success rate and iteration count, but lacked deterministic behaviour. In contrast, the coarse grid search exhibited more predictable performance, with execution times on RFSoC hardware shown to be below 40 ms, but tended to miss valid solutions at higher bandwidths. Therefore, a hybrid search method was proposed, combining coarse grid and random search methods, which would balance coverage and performance.

The filter design method presented in **Chapter 5** was motivated by the need for run-time filter generation in CR applications. By combining window and frequency sampling techniques, a natively fixed-point filter design method was developed and implemented on an FPGA using Model Composer and evaluated using PYNQ. The method generated filters directly on the FPGA, removing the need for external filter generation tools and allowing for faster reconfiguration times. Results showed stopband attenuation as low as 88 dB, narrow transition bandwidths of less than 1% of the sample rate, and execution times three orders of magnitude faster than equivalent non-native

methods. Resource usage was less than 1%, even for longer length filters, making this method suitable for SDR and CR applications.

These results demonstrate that the RFSoC is a powerful platform for building low-latency, reconfigurable radio systems using accessible design tools, with enough flexibility to support both real-time performance and high productivity.

6.3 Further Work

While the results in this thesis have established the RFSoC as a useful platform for SDR and CR applications, there are several areas where further investigation could build on the methods and tools developed in this work.

In **Chapter 3**, it was found there was little difference in performance between hardware and software implementations of the FFT used for signal visualisation. This suggests that larger FFT sizes or higher frame rates could be supported, potentially improving resolution or responsiveness during debugging. Additionally, while the generated HDL from Model Composer was effective in terms of functionality, it would be of interest to formally evaluate the quality of this generated code compared to other HLS tools and handwritten RTL. Metrics such as resource usage, timing closure, and throughput are of particular interest, and found lacking in the academic literature—especially for more complex applications.

In **Chapter 4**, additional results showed that SPECTRE achieved significantly faster execution times on a host-based processor, with the coarse grid algorithm running more than 10x faster than on the RFSoC. This opens the possibility of moving SPECTRE to a host computer and using it to control the RFSoC remotely. Doing so could allow more complex and computationally intensive algorithms to be used without the constraints of the comparatively low-power Arm processor on the RFSoC. With that said, improvements could still be made to the current implementation, such as optimising the code for parallel processing or hardware acceleration, or even using a system-level language, such as C++, to improve performance. Additional search algorithms could

also be evaluated to improve coverage or reduce variance. All of these would improve SPECTRE's suitability for real-time operation.

In **Chapter 5**, one of the challenges described when discussing the Least Squares filter design method was the need for a pseudo-inverse operation, which is historically difficult to implement on FPGAs. However, since the publication of the original paper, MathWorks have released a *Real Burst QR Decomposition* IP block as part of their HDL Coder software. This block could be used to revisit the problem, allowing the Least Squares algorithm to be implemented on the FPGA. However, as mentioned in the chapter, for longer length filters, matrix size may still be a limiting factor. In addition, newer FPGA families such as the AMD Versal include DSP58 slices with native floating-point support [190]—making it feasible to implement more advanced design methods in hardware. Exploring these options would be a valuable area of research, complementing the work presented in the original paper.

Finally, since the original publications and early stages of this thesis were completed, the landscape of machine learning and artificial intelligence has changed significantly. Tools and frameworks that were unavailable just a few years ago are now being used to solve optimisation and design problems across a wide range of engineering disciplines. Given the emphasis in this work on automation, adaptive system design, and real-time decision-making, it would be worth investigating how modern AI/ML techniques could be applied to the methods presented here. With that said, it is important to note that while AI/ML techniques can be used for optimisation and decision-making, they often require large amounts of data and computational resources, which may not be feasible in all scenarios. Therefore, careful consideration should be given to the specific use case and the available resources when exploring these techniques, as traditional methods still continue to provide reliable and deterministic solutions.

6.4 Final Remarks

As wireless demand continues to grow, the need for a more efficient use of the radio spectrum becomes increasingly important. Technologies such as DSA, SDR, and CR, offer promising solutions to this problem but, until recently, the hardware required to

support real-time reconfigurability has been limited. This thesis has shown that RF-sampling SoCs, such as the RFSoC, provide the necessary hardware to build practical, low-latency radio platforms that can adapt to changing conditions and requirements.

The methodology and algorithms developed in this thesis—from a full transceiver design to an integrated frequency planning tool and RFIR filter design method—demonstrate a framework for building future wireless systems that make better use of the available spectrum. Together, they address the challenges of adaptive and autonomous wireless system design, with a focus on reconfigurability, determinism, and productivity.

Appendices

Appendix A

RFSoC4x2 Development Platform

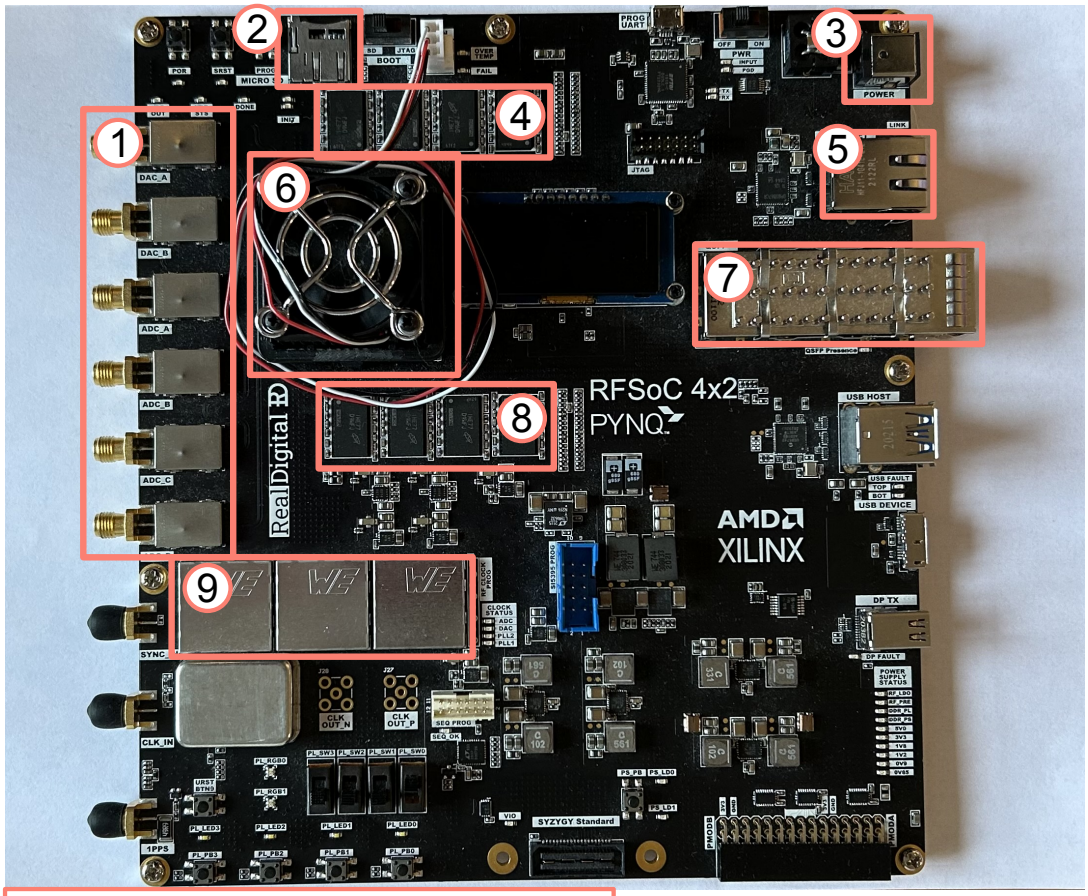

Figure A.1: Labelled photograph of the RFSoC4x2 development platform.

Appendix B

RFSoC QPSK Transceiver Design

B.1 QPSK Transceiver Jupyter Notebook

B.2 RFSoC C Driver UML Diagram

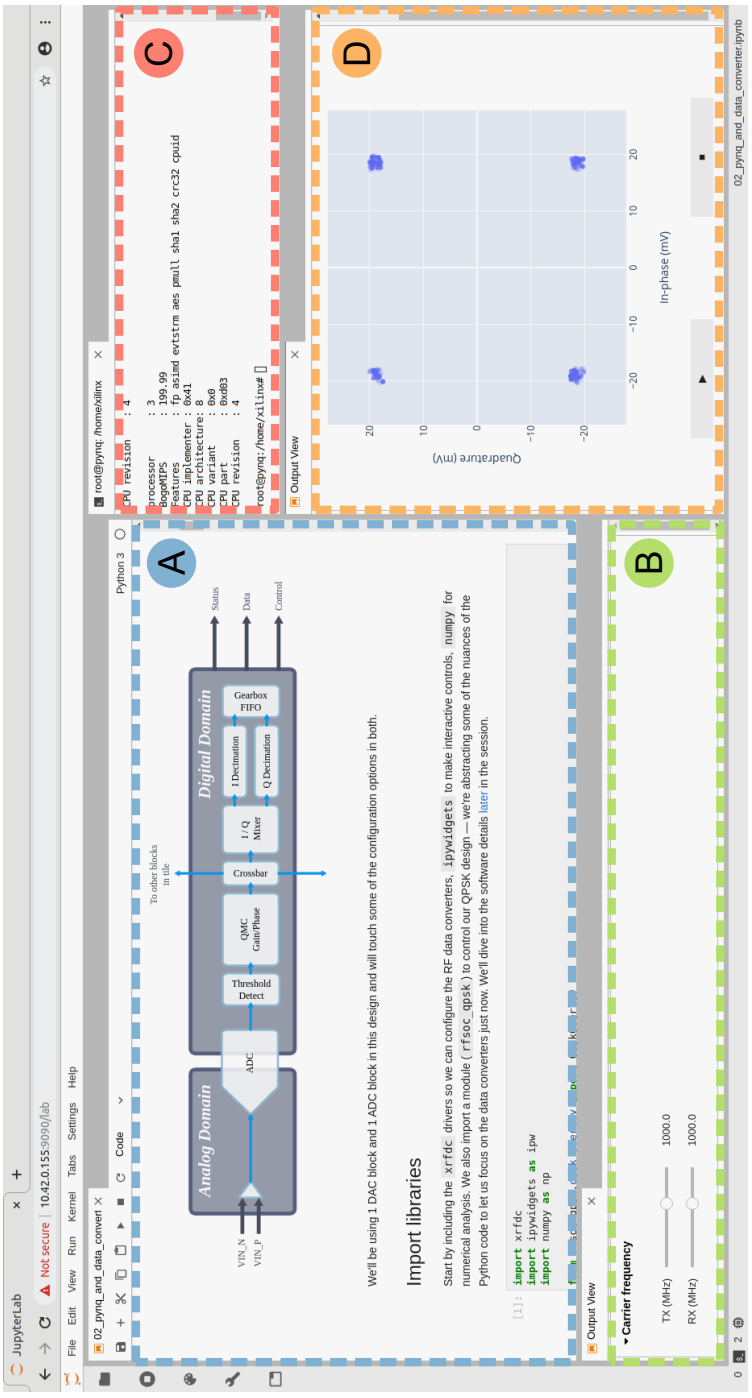

Figure B.1: An example of a Jupyter notebook running the transceiver design, providing real-time control of the hardware and visualisation of the signal path, displaying A) the main view of the notebook showing markdown formatted text and Python code blocks, B) *ipywidget* controls, C) a Linux terminal session, and D) a live constellation plot.

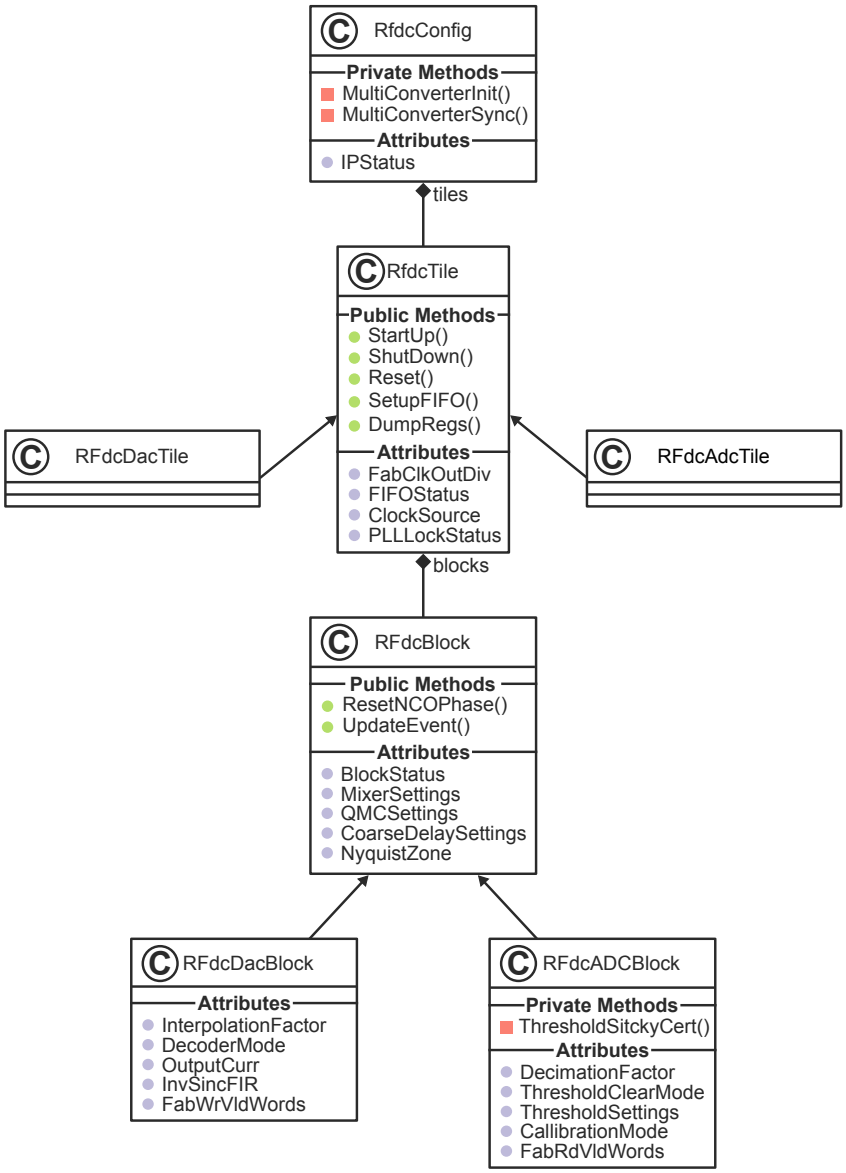

Figure B.2: A UML class diagram of the data converter driver written in Python.

Appendix C

Frequency Planner Search Algorithm

Algorithm 1 Frequency planner search over sample rate and PLL frequency

Require: Signal centre frequency f_c, bandwidth BW
Require: Device sample-rate limits $f_{s,\min}^{dev}$, $f_{s,\max}^{dev}$, step Δf_s
Require: Spur sets: sample-rate stage $\mathcal{S}_{fs}$ (tiered), PLL stage $\mathcal{S}_{pll}$
Require: PLL prune threshold Δf_{pll} (default 1 MHz)
Ensure: A valid pair (f_s, f_{pll}) or FAILURE

$f_{s,\min} \leftarrow \max\Big(f_{s,\min}^{dev},\, f_c\Big), \quad f_{s,\max} \leftarrow f_{s,\max}^{dev}$
Build the list of candidate sample rates $\mathcal{F}_s \leftarrow \{f_{s,\min}, f_{s,\min} + \Delta f_s, \ldots, f_{s,\max}\}$
Optionally shuffle $\mathcal{F}_s$ (randomised ordering)
for all $f_s \in \mathcal{F}_s$ **do**
    Set planner parameters (f_c, BW, f_s)
    if any spur in $\mathcal{S}_{fs}$ overlaps the signal band (Eq. 4.3) **then**
        continue
    end if
    Generate candidate PLL frequencies $\mathcal{F}_{pll} \leftarrow$ VALIDPLLLIST(f_s)
    Remove candidates within Δf_{pll} of f_c:
    $\mathcal{F}_{pll} \leftarrow \{f_{pll} \in \mathcal{F}_{pll} : |f_{pll} - f_c| > \Delta f_{pll}\}$
    Optionally shuffle $\mathcal{F}_{pll}$ (randomised ordering)
    if $\mathcal{F}_{pll} = \emptyset$ **then**
        continue
    end if
    for all $f_{pll} \in \mathcal{F}_{pll}$ **do**
        Set planner PLL frequency to f_{pll}
        if no spur in $\mathcal{S}_{pll}$ overlaps the signal band (Eq. 4.3) **then**
            return (f_s, f_{pll})
        end if
    end for
end for
return FAILURE

Bibliography

[1] R. Akeela and B. Dezfouli, "Software-defined radios: Architecture, state-of-the-art, and challenges," *Computer Communications*, vol. 128, pp. 106–125, 2018.

[2] R. Subbaraman, N. Bhaskar, S. Crow, M. Khazraee, A. Schulman, and D. Bharadia, "Observing wideband rf spectrum with low-cost, resource limited sdrs," in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pp. 617–618, 2022.

[3] G. Kakkavas, K. Tsitseklis, V. Karyotis, and S. Papavassiliou, "A software defined radio cross-layer resource allocation approach for cognitive radio networks: From theory to practice," *IEEE Transactions on Cognitive Communications and Networking*, vol. 6, no. 2, pp. 740–755, 2020.

[4] A. Martian, F. L. Chiper, O. M. K. Al-Dulaimi, M. J. A. Al Sammarraie, C. Vladeanu, and I. Marghescu, "Comparative analysis of software defined radio platforms for spectrum sensing applications," in *2020 13th International Conference on Communications (COMM)*, pp. 369–374, IEEE, 2020.

[5] D. M. Molla, H. Badis, L. George, and M. Berbineau, "Software defined radio platforms for wireless technologies," *IEEE Access*, vol. 10, pp. 26203–26229, 2022.

[6] AMD Inc., "Zynq UltraScale+ RFSoC Product Selection Guide (XMP105)," 2023.

[7] L. B. Jackson, *Digital Filters and Signal Processing*. Kluwer Academic Publishers, 1986.

[8] G. Staple and K. Werbach, "The end of spectrum scarcity [spectrum allocation and utilization]," *IEEE Spectrum*, vol. 41, no. 3, pp. 48–52, 2004.

[9] Accenture, "Securing the Future of U.S. Wireless Networks: The Looming Spectrum Crisis," report, CTIA, March 2025.

[10] M. Wellens, J. Wu, and P. Mahonen, "Evaluation of spectrum occupancy in indoor and outdoor scenario in the context of cognitive radio," in *2007 2nd International Conference on Cognitive Radio Oriented Wireless Networks and Communications*, pp. 420–427, IEEE, 2007.

[11] A. B. MacKenzie, J. H. Reed, P. Athanas, C. W. Bostian, R. M. Buehrer, L. A. DaSilva, S. W. Ellingson, Y. T. Hou, M. Hsiao, J.-M. Park, *et al.*, "Cognitive radio and networking research at virginia tech," *Proceedings of the IEEE*, vol. 97, no. 4, pp. 660–688, 2009.

[12] E. Hossain, D. Niyato, and Z. Han, *Dynamic Spectrum Access and Management in Cognitive Radio Networks*. Cambridge University Press, 2009.

[13] P. Leaves, K. Moessner, R. Tafazolli, D. Grandblaise, D. Bourse, R. Tonjes, and M. Breveglieri, "Dynamic spectrum allocation in composite reconfigurable wireless networks," *IEEE Communications Magazine*, vol. 42, no. 5, pp. 72–81, 2004.

[14] I. F. Akyildiz, W.-Y. Lee, M. C. Vuran, and S. Mohanty, "Next generation/dynamic spectrum access/cognitive radio wireless networks: A survey," *Computer networks*, vol. 50, no. 13, pp. 2127–2159, 2006.

[15] I. F. Akyildiz, W.-Y. Lee, M. C. Vuran, and S. Mohanty, "A survey on spectrum management in cognitive radio networks," *IEEE Communications magazine*, vol. 46, no. 4, pp. 40–48, 2008.

[16] J. Mitola and G. Maguire, "Cognitive radio: making software radios more personal," *IEEE Personal Communications*, vol. 6, no. 4, pp. 13–18, 1999.

[17] A. M. Wyglinski, M. Nekovee, and Y. T. Hou, "When radio meets software," in *Cognitive radio communications and networks*, Elsevier Inc., 2010.

[18] S. Haykin, "Cognitive radio: brain-empowered wireless communications," *IEEE Journal on Selected Areas in Communications*, vol. 23, no. 2, pp. 201–220, 2005.

[19] C. R. Stevenson, G. Chouinard, Z. Lei, W. Hu, S. J. Shellhammer, and W. Caldwell, "Ieee 802.22: The first cognitive radio wireless regional area network standard," *IEEE communications magazine*, vol. 47, no. 1, pp. 130–138, 2009.

[20] Y.-C. Liang, *Dynamic spectrum management: from cognitive radio to blockchain and artificial intelligence*. Springer Nature, 2020.

[21] S. Sodagari, "Real-time scheduling for cognitive radio networks," *IEEE Systems Journal*, vol. 12, no. 3, pp. 2332–2343, 2017.

[22] K. Kunert, M. Jonsson, and U. Bilstrup, "Deterministic real-time medium access for cognitive industrial radio networks," in *2012 9th IEEE International Workshop on Factory Communication Systems*, pp. 91–94, IEEE, 2012.

[23] H. Kopetz and W. Steiner, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, third ed., 2022.

[24] G. C. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer, fourth ed., 2024.

[25] W. Yi, M. Mohaqeqi, and S. Graf, "Mimos: A deterministic model for the design and update of real-time systems," in *International Conference on Coordination Languages and Models*, pp. 17–34, Springer, 2022.

[26] E. A. Lee, "Determinism," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5, pp. 1–34, 2021.

[27] Y. Anne, "Ensuring real-time determinism in complex embedded robotic systems," *Sch J Eng Tech*, vol. 11, pp. 855–859, 2025.

[28] Y. Tawk, J. Costantine, and C. Christodoulou, "Cognitive-radio and antenna functionalities: A tutorial [wireless corner]," *IEEE Antennas and Propagation Magazine*, vol. 56, no. 1, pp. 231–243, 2014.

[29] T. Weingart, D. C. Sicker, and D. Grunwald, "A statistical method for reconfiguration of cognitive radios," *IEEE Wireless Communications*, vol. 14, no. 4, pp. 34–40, 2007.

[30] A. Ghassemi, S. Bavarian, and L. Lampe, "Cognitive radio for smart grid communications," in *2010 First IEEE international conference on smart grid communications*, pp. 297–302, IEEE, 2010.

[31] A. N. Mody and G. Chounard, "Enabling urural broadband wireless access using cognitive radio technology," 2010.

[32] S. Gupta and V. Malagar, "Ieee 802.22 standard for regional area networks," in *2017 International Conference on Next Generation Computing and Information Systems (ICNGCIS)*, pp. 126–130, IEEE, 2017.

[33] A. Mukherjee, J.-F. Cheng, S. Falahati, H. Koorapaty, D. H. Kang, R. Karaki, L. Falconetti, and D. Larsson, "Licensed-assisted access lte: coexistence with ieee 802.11 and the evolution toward 5g," *IEEE Communications Magazine*, vol. 54, no. 6, pp. 50–57, 2016.

[34] M. Hirzallah, M. Krunz, B. Kecicioglu, and B. Hamzeh, "5g new radio unlicensed: Challenges and evaluation," *IEEE Transactions on Cognitive Communications and Networking*, vol. 7, no. 3, pp. 689–701, 2020.

[35] R. Farrell, M. Sanchez, and G. Corley, "Software-defined radio demonstrators: An example and future trends," *International Journal of Digital Multimedia Broadcasting*, vol. 2009, p. 547650, 2009.

[36] K. Le, P. Maddala, C. Gutterman, K. Soska, A. Dutta, D. Saha, P. Wolniansky, D. Grunwald, and I. Seskar, "Cognitive radio kit framework: Experimental platform for dynamic spectrum research," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 17, no. 1, pp. 30–39, 2013.

[37] G. Eichinger, K. Chowdhury, and M. Leeser, "Crush: Cognitive radio universal software hardware," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, (Oslo, Norway), 2012.

[38] J. Manco, I. Dayoub, A. Nafkha, M. Alibakhshikenari, and H. B. Thameur, "Spectrum sensing using software defined radio for cognitive radio networks: A survey," *IEEE Access*, vol. 10, pp. 131887–131908, 2022.

[39] K. R. Chowdhury and T. Melodia, "Platforms and testbeds for experimental evaluation of cognitive ad hoc networks," *IEEE Communications Magazine*, vol. 48, no. 9, pp. 96–104, 2010.

[40] M. Catt and D. Henderson, "Direct rf fpgas built with multi-chip packaging overcome technology challenges," in *IEEE High Performance Extreme Computing Conference (HPEC)*, (Wakefield, MA, USA), 2024.

[41] Y. Yuan, J. Tu, L. Lu, and S. Chen, "Design of cognitive radio hardware platform based on rfsoc," in *9th International Conference on Intelligent Computing and Signal Processing (ICSP)*, (Xian, China), 2024.

[42] O. Chu, "Research on design and implementation of 5g wireless communication system based on rfsoc and rf sampling," in *IEEE 6th International Conference on Automation, Electronics and Electrical Engineering (AUTEEE)*, pp. 1158–1164, 2023.

[43] D. Siafarikas and J. L. Volakis, "Toward direct rf sampling: Implications for digital communications," *IEEE Microwave Magazine*, vol. 21, no. 9, pp. 43–52, 2020.

[44] S. Henthorn, T. O'Farrell, M. R. Anbiyaei, and K. L. Ford, "Concurrent multiband direct rf sampling receivers," *IEEE Transactions on Wireless Communications*, vol. 22, no. 1, pp. 550–562, 2023.

[45] C. Liu, L. Ruckman, and R. Herbst, "Evaluating direct rf sampling performance for rfsoc-based radio-frequency astronomy receivers," *arXiv preprint arXiv:2309.08067*, 2023.

[46] B. Farley, C. Erdmann, B. Vaz, J. McGrath, E. Cullen, B. Verbruggen, R. Pelliconi, D. Breathnach, P. Lim, A. Boumaalif, P. Lynch, C. Mesadri, D. Melinn, K. P. Yap, and L. Madden, "A programmable RFSoC in 16nm FinFET technology for wideband communications," in *2017 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, pp. 1–4, 2017.

[47] C. Randieri and V. F. Puglisi, "Design of a software defined radio using soc builder," in *CEUR Workshop Proceedings*, vol. 3398, pp. 73–77, 2022.

[48] J. Goldsmith, C. Ramsay, D. Northcote, K. W. Barlee, L. H. Crockett, and R. W. Stewart, "Control and Visualisation of a Software Defined Radio System on the Xilinx RFSoC Platform Using the PYNQ Framework," *IEEE Access*, vol. 8, pp. 129012–129031, July 2020.

[49] K. Mao, "Fast prototyping of massive mimo user equipment using pynq," 2024.

[50] L. H. Crockett, D. Northcote, and R. Stewart, eds., *Software Defined Radio with Zynq UltraScale+ RFSoC*. Strathclyde Academic Media, Jan. 2023.

[51] C. Beyerstedt, J. Meier, F. Speicher, R. Wunderlich, and S. Heinen, "Analysis of the frequency conversion of spurious tones in frequency dividers and development of an event-driven model for system simulations," *Advances in Radio Science*, vol. 17, pp. 101–107, 2019.

[52] Analog Devices, Inc., "Frequency Folding Tool." `https://www.analog.com/en/design-center/interactive-design-tools/frequency-folding-tool.html`, 2015. Accessed: Feb. 8, 2026.

[53] Keysight, "WhatIF Frequency Planner." `https://edadocs.software.keysight.com/display/genesys2009/WhatIF+Frequency+Planner`. Accessed: Feb. 8, 2026.

[54] A. Roetter and D. A. Belliveau, "Single-tone IM distortion analyses via the Web: a spur chart calculator written in Java," *Microwave Journal*, vol. 40, no. 11, pp. 106–114, 1997.

[55] J. Goldsmith, L. H. Crockett, and R. W. Stewart, "A natively fixed-point run-time reconfigurable FIR filter design method for FPGA hardware," *IEEE Open Journal of Circuits and Systems*, vol. 3, pp. 25–37, Feb. 2022.

[56] V. Valenta, R. Maršálek, G. Baudoin, M. Villegas, M. Suarez, and F. Robert, "Survey on spectrum utilization in Europe: Measurements, analyses and observations," in *2010 Proceedings of the Fifth International Conference on Cognitive Radio Oriented Wireless Networks and Communications*, pp. 1–5, 2010.

[57] "Mobile networks and spectrum: Meeting future demand for mobile data," discussion paper, Ofcom, Feb. 2022.

[58] "United Kingdom Frequency Allocation Table," tech. rep., Ofcom, Jan. 2017.

[59] H. B. Weldu, B. Kim, and B.-h. Roh, "Deterministic approach to rendezvous channel setup in cognitive radio networks," in *The International Conference on Information Networking 2013 (ICOIN)*, pp. 690–695, 2013.

[60] M. A. Azza, A. El Moussati, and R. Barrak, "Implementation of cognitive radio applications on a software defined radio plateform," in *2014 International Conference on Multimedia Computing and Systems (ICMCS)*, pp. 1037–1041, 2014.

[61] J. de Curtò and I. de Zarzà, "Large model cognitive radio: Software-defined radios and advanced language models," in *2024 International Conference on Ubiquitous Computing and Communications (IUCC)*, pp. 8–14, 2024.

[62] COTS Staff, "Deterministic Software Solutions for Complex Architectures Involving Software-Defined Radio (SDR)," *The Journal of Military Electronics and Computing*, May 2022.

[63] J. Mitola, "Software radios: Survey, critical evaluation and future directions," *IEEE Aerospace and Electronic Systems Magazine*, vol. 8, no. 4, pp. 25–36, 1993.

[64] J. Mitola, "The software radio architecture," *IEEE Communications Magazine*, vol. 33, no. 5, pp. 26–38, 1995.

[65] "Wireless Transceiver Design: Physical (PHY) Layer Overview." `https://www.mathworks.com/discovery/wireless-transceiver.html`, 2025. "The PHY layer consists of an RF front end and a baseband processor.".

[66] L. Goeller and D. Tate, "A Technical Review of Software Defined Radios: Vision, Reality, and Current Status," in *2014 IEEE Military Communications Conference*, pp. 1466–1470, IEEE, 2014.

[67] C. Partridge, "Realizing the future of wireless data communications," *Communications of the ACM*, vol. 54, no. 9, pp. 62–68, 2011.

[68] M. Sneps-Sneppe, D. Namiot, and E. Tikhonov, "On Software Defined Radio Issues," in *2022 Workshop on Microwave Theory and Techniques in Wireless Communications (MTTW)*, pp. 35–40, 2022.

[69] Xilinx, Inc., *DS190, Zynq-7000 SoC Data Sheet: Overview*, July 2018.

[70] S. Cammerer, G. Marcus, T. Zirr, F. Aït Aoudia, L. Maggi, J. Hoydis, and A. Keller, "Sionna research kit: A gpu-accelerated research platform for ai-ran," in *2025 IEEE International Conference on Machine Learning for Communication and Networking (ICMLCN)*, pp. 1–2, IEEE, 2025.

[71] G. Sklivanitis, A. Gannon, S. N. Batalama, and D. A. Pados, "Addressing next-generation wireless challenges with commercial software-defined radio platforms," *IEEE Communications Magazine*, vol. 54, no. 1, pp. 59–67, 2016.

[72] J. Liu, H. Wu, Z. Li, B. Ding, and T. Wang, "Cr-grt: A novel sdr platform optimized for real-time cognitive radio applications," in *Proceedings of the 21st ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pp. 333–341, 2018.

[73] D. Cabric, S. M. Mishra, and R. W. Brodersen, "Implementation issues in spectrum sensing for cognitive radios," in *Conference Record of the Thirty-Eighth Asilomar Conference on Signals, Systems and Computers, 2004.*, vol. 1, pp. 772–776, Ieee, 2004.

[74] G. Nychis, T. Hottelier, Z. Yang, S. Seshan, and P. Steenkiste, "Enabling mac protocol implementations on software-defined radios.," in *NSDI*, vol. 9, pp. 91–105, 2009.

[75] A. A. Abidi, "The Path to the Software-Defined Radio Receiver," *IEEE Journal of Solid-State Circuits*, vol. 42, no. 5, pp. 954–966, 2007.

[76] A. Parssinen, J. Jussila, J. Ryynanen, L. Sumanen, and K. A. Halonen, "A 2-GHz wide-band direct conversion receiver for WCDMA applications," *IEEE Journal of Solid-State Circuits*, vol. 34, no. 12, pp. 1893–1903, 1999.

[77] A. Brown and B. Wolt, "Digital L-band receiver architecture with direct RF sampling," in *Proceedings of 1994 IEEE Position, Location and Navigation Symposium-PLANS'94*, pp. 209–216, IEEE, 1994.

[78] U. Jayamohan, "Not your grandfather's ADC: RF sampling ADCs offer advantages in systems design," tech. rep., Analog Devices, Inc., 2015.

[79] J. J. McCue, B. Dupaix, L. Duncan, B. Mathieu, S. McDonnell, V. J. Patel, T. Quach, and W. Khalil, "A Time-Interleaved Multimode Delta-Sigma RF-DAC for Direct Digital-to-RF Synthesis," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 5, pp. 1109–1124, 2016.

[80] D. Gruber, M. Clara, R. Sanchez, Y.-s. Wang, C. Duller, G. Rauter, P. Torta, and K. Azadet, "10.6 a 12b 16GS/s RF-sampling capacitive DAC for multi-band software-radio base-station applications with on-chip transmission-line matching network in 16nm FinFET," in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64, pp. 174–176, IEEE, 2021.

[81] M. Guibord, "How to complete your RF sampling solution," tech. rep., Texas Instruments Incorporated, 2016.

[82] P. Delos, "A Review of Wideband RF Receiver Architecture Options," tech. rep., Analog Devices, Inc., 2017.

[83] H. Saheb and S. Haider, "Scalable high speed serial interface for data converters: Using the JESD204B industry standard," in *2014 9th International Design and Test Symposium (IDT)*, pp. 6–11, IEEE, 2014.

[84] "AMD Versal RF Series Product Brief: Wideband Spectrum. Massive DSP Compute. Single Chip.," 2024. Product Brief.

[85] Intel Corporation, "Intel Analog-Enabled Direct-RF Device Portfolio." `https://www.intel.com/content/www/us/en/products/docs/programmable/direct-rf-series-fpga-white-paper.html`, 2023. Accessed: Feb. 8, 2026.

[86] Xilinx Inc., "An Adaptable Direct RF-Sampling Solution, WP489," Feb. 2019.

[87] Xilinx, Inc., *DS889, Zynq Ultrascale+ RFSoC Data Sheet: Overview*, Apr. 2021.

[88] "RFSoC 4x2 | Real Digital." `https://www.realdigital.org/hardware/rfsoc-4x2`, June 2025. Accessed: Feb. 8, 2026.

[89] M. Šiaučiulis, D. Northcote, J. Goldsmith, L. H. Crockett, and Š. Kaladè, "100gbit/s rf sample offload for rfsoc using gnu radio and pynq," in *2023 21st IEEE Interregional NEWCAS Conference (NEWCAS)*, pp. 1–5, IEEE, 2023.

[90] R. W. Stewart, L. Crockett, D. Atkinson, K. Barlee, D. Crawford, I. Chalmers, M. McLernon, and E. Sozer, "A low-cost desktop software defined radio design environment using matlab, simulink, and the rtl-sdr," *IEEE Communications Magazine*, vol. 53, no. 9, pp. 64–71, 2015.

[91] A. S. Bañacia and Q. P. R. M. Gelu, "A simplified IEEE 802.22 PHY layer in Matlab-Simulink and SDR platform," in *2016 International Conference on Electronics, Information, and Communications (ICEIC)*, pp. 1–4, IEEE, 2016.

[92] H. Hassan, C. K. H. C. K. Yahaya, N. Ahmad, and N. I. S. Bakhtir, "Low complexity SDR transceiver design using Simulink, Matlab and Xilinx," in *2012 International Conference on ICT Convergence (ICTC)*, pp. 55–60, IEEE, 2012.

[93] A. A. Tabassam, F. A. Ali, S. Kalsait, and M. U. Suleman, "Building software-defined radios in matlab simulink-a step towards cognitive radios," in *2011 UkSim 13th International Conference on Computer Modelling and Simulation*, pp. 492–497, IEEE, 2011.

[94] C. Codau, R. Buta, E. Puschita, T. Palade, P. Dolea, R. Simedroni, and A. Pastrav, "Implementation of an sdr-based fmcw radar receiver using labview nxg," in *2022 International Workshop on Antenna Technology (iWAT)*, pp. 114–117, IEEE, 2022.

[95] N. A. S. Bauomy and E. A. Elbeh, "Design of sdr simulation for wireless communication between ground station and cubesat implemented by labview," in *2020 8th International Japan-Africa Conference on Electronics, Communications, and Computations (JAC-ECC)*, pp. 60–63, IEEE, 2020.

[96] B. Raghunandan, A. Mahesh, and M. Mahesha Babu, "Wireless communication system design using labVIEW and software defined radio," in *2023 7th International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS)*, pp. 1–6, IEEE, 2023.

[97] A. Mate, K.-H. Lee, and I.-T. Lu, "Spectrum sensing based on time covariance matrix using gnu radio and usrp for cognitive radio," in *2011 IEEE Long Island Systems, Applications and Technology Conference*, pp. 1–6, IEEE, 2011.

[98] J. Tapparel, O. Afisiadis, P. Mayoraz, A. Balatsoukas-Stimming, and A. Burg, "An open-source lora physical layer prototype on gnu radio," in *2020 IEEE 21st International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, pp. 1–5, IEEE, 2020.

[99] B. I. Supriyatno, T. Hidayat, A. B. Susksmono, and A. Munir, "Development of radio telescope receiver based on gnu radio and usrp," in *2015 1st International Conference on Wireless and Telematics (ICWT)*, pp. 1–4, IEEE, 2015.

[100] R. Doost-Mohammady, O. Bejarano, L. Zhong, J. R. Cavallaro, E. Knightly, Z. M. Mao, W. W. Li, X. Chen, and A. Sabharwal, "Renew: Programmable and observable massive mimo networks," in *2018 52nd Asilomar conference on signals, systems, and computers*, pp. 1654–1658, IEEE, 2018.

[101] I. Lapin, G. S. Granados, J. Samson, O. Renaudin, F. Zanier, and L. Ries, "Stare: Real-time software receiver for lte and 5g nr positioning and signal monitoring," in *2022 10th Workshop on Satellite Navigation Technology (NAVITEC)*, pp. 1–11, IEEE, 2022.

[102] C. Shepard, R. Doost-Mohammady, J. Ding, R. E. Guerra, and L. Zhong, "Argosnet: A multi-cell many-antenna mu-mimo platform," in *2018 52nd Asilomar Conference on Signals, Systems, and Computers*, pp. 2237–2241, IEEE, 2018.

[103] A. Ibanez, J. Sanchez, D. Gomez-Barquero, J. Mika, S. Babel, and K. Kuehnhammer, "5g broadcast sdr open source platforms," in *2022 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, pp. 01–06, IEEE, 2022.

[104] D. C. Popescu and R. Vida, "A primer on software defined radios," *Infocommunications Journal*, vol. 14, no. 3, 2022.

[105] National Instruments Corp., "Ettus USRP X410 User Manual," 2026.

[106] AMD, Ltd., *UG892, Vivado Design Suite User Guide: Design Flows Overview*, May 2023.

[107] AMD, Ltd., *UG1400, Vitis Unified Software Platform Documentation: Embedded Software Development*, July 2023.

[108] AMD, Ltd., *UG1144, PetaLinux Tools Documentation: Reference Guide*, May 2023.

[109] A. Gondhalekar, T. Twomey, and W.-c. Feng, "On the characterization of the performance-productivity gap for fpga," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–8, IEEE, 2022.

[110] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, *et al.*, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2015.

[111] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[112] AMD, Ltd., *UG1399, Vitis High-Level Synthesis User Guide*, July 2023.

[113] "Welcome to the MyHDL documentation." `https://docs.myhdl.org/en/stable/`, 2018. Accessed: Feb. 8, 2026.

[114] S. Lahti and T. D. Hämäläinen, "High-level synthesis for fpgas-a hardware engineer's perspective," *IEEE Access*, 2025.

[115] The Mathworks, Inc., *Embedded Coder User's Guide*, Mar. 2023.

[116] The Mathworks, Inc., *HDL Coder User's Guide*, Mar. 2023.

[117] S. Srilakshmi and G. Madhumati, "A Comparative Analysis of HDL and HLS for Developing CNN Accelerators," in *2023 Third International Conference on Artificial Intelligence and Smart Energy (ICAIS)*, pp. 1060–1065, IEEE, 2023.

[118] H. S. Lee and J. W. Jeon, "Comparison between HLS and HDL image processing in FPGAs," in *2020 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*, pp. 1–2, IEEE, 2020.

[119] S. Kayed and G. Elsayed, "An optimizing technique for using MATLAB HDL Coder," *Bulletin of the National Research Centre*, vol. 47, no. 1, p. 94, 2023.

[120] AMD, Ltd., *UG1483, Vitis Model Composer User Guide*, May 2023.

[121] D. Haessig, J. Hwang, S. Gallagher, and M. Uhm, "Case-study of a xilinx system generator design flow for rapid development of sdr waveforms," in *Proc. SDR Forum Technical Conference, Orange County*, p. 6, 2005.

[122] C. Lavin, B. Nelson, J. Palmer, and M. Rice, "An fpga-based space-time coded telemetry receiver," in *2008 IEEE National Aerospace and Electronics Conference*, pp. 250–256, IEEE, 2008.

[123] L. H. Crockett, D. Northcote, C. Ramsay, F. D. Robinson, and R. W. Stewart, *Exploring Zynq MPSoC: With PYNQ and Machine Learning Applications*. Strathclyde Academic Media, Apr. 2019.

[124] AMD, Inc., "PYNQ: Python productivity for Adaptive Computing platforms." `https://pynq.readthedocs.io/en/latest`. Accessed: Feb. 8, 2026.

[125] Z. Alomari, O. E. Halimi, K. Sivaprasad, and C. Pandit, "Comparative studies of six programming languages," *arXiv preprint arXiv:1504.00693*, 2015.

[126] A. G. Schmidt, G. Weisz, and M. French, "Evaluating rapid application development with python for heterogeneous processor-based fpgas," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 121–124, IEEE, 2017.

[127] B. Hutchings and M. Wirthlin, "Rapid implementation of a partially reconfigurable video system with pynq," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, IEEE, 2017.

[128] M. Tsukada, M. Kondo, and H. Matsutani, "A neural network-based on-device learning anomaly detector for edge devices," *IEEE Transactions on Computers*, vol. 69, no. 7, pp. 1027–1044, 2020.

[129] A. Vaishnav, K. D. Pham, J. Powell, and D. Koch, "Fos: A modular fpga operating system for dynamic workloads," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 4, pp. 1–28, 2020.

[130] W. V. Kritikos, A. G. Schmidt, R. Sass, E. K. Anderson, and M. French, "Redsharc: A programming model and on-chip network for multi-core systems on a programmable chip," *International Journal of Reconfigurable Computing*, vol. 2012, no. 1, p. 872610, 2012.

[131] L. G. León-Vega, D. Avila-Torres, I. León-Vega, and J. Castro-Godínez, "Cynq: Speeding up fpga applications with simplicity," in *2024 31st IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pp. 1–4, IEEE, 2024.

[132] F. J. Harris, *Multirate signal processing for communication systems*. CRC Press, 2022.

[133] P. E. Allen, R. Dobkin, and D. R. Holberg, *CMOS analog circuit design*. Elsevier, 2011.

[134] S. Pavan, R. Schreier, and G. C. Temes, *Understanding delta-sigma data converters*. John Wiley and Sons, 2017.

[135] W. Kester, "Taking the Mystery out of the Infamous Formula, "SNR= 6.02N+ 1.76 dB," and Why You Should Care," tech. rep., Analog Devices, Inc., 2009.

[136] N.-S. Kim, "A 10-bit, 600 ms/s multi-mode direct-sampling dac-based transmitter," *IEEE Access*, vol. 10, pp. 125696–125706, 2022.

[137] "Recommendation ITU-R SM.329-9: Spurious emissions," Tech. Rep. SM.329-9, International Telecommunication Union, Geneva, Switzerland, July 2001.

[138] Keysight Technologies, "Transmit and Receive In-Band and Out-of-Band Spurious Emissions," Accessed: Feb. 8, 2026.

[139] O. Hanay, E. Bayram, M. S. Elsayed, and R. Negra, "Systematic frequency planning for a high SFDR digital-IF RF-DAC-based transmitter," in *2015 Nordic Circuits and Systems Conference (NORCAS): NORCHIP & International Symposium on System-on-Chip (SoC)*, pp. 1–4, IEEE, 2015.

[140] W. Kester, *Data conversion handbook*. Newnes, 2005.

[141] A. M. Ali, *High speed data converters*. Institution of Engineering and Technology, 2016.

[142] F. Maloberti, *Data converters specifications*. Springer, 2007.

[143] Texas Instruments Incorporated, "RF-Sampling Frequency Planner, Analog Filter, and DDC Excel™ Calculator." `https://www.ti.com/tool/FREQ-DDC-FILTER-CALC`. Accessed: Feb. 8, 2026.

[144] AMD, Inc., "RFSoC Frequency Planner Quick Start Guide." `https://www.xilinx.com/content/dam/xilinx/publications/quick-start/rfsoc-frequency-planner-quick-start-guide.pdf`, 2022. Accessed: Feb. 8, 2026.

[145] D. Gandhi and C. Lyons, "Mixer spur analysis with concurrently swept LO, RF and IR: tools and techniques.(Technical Feature).," *Microwave Journal*, vol. 46, no. 5, pp. 212–217, 2003.

[146] J. L. Flores, "The Distances Chart: A New Approach To Spurs Calculation.," *Microwave Journal*, vol. 53, no. 2, 2010.

[147] C. Huang, L.-x. Ren, E.-k. Mao, and P.-k. He, "A systematic frequency planning method in Direct Digital Synthesizer (DDS) design," in *2009 International Conference on Wireless Communications & Signal Processing*, pp. 1–4, IEEE, 2009.

[148] R. W. Hamming, *Digital filters*. Prentice Hall, 1989.

[149] M. Kumm, K. Möller, and P. Zipf, "Dynamically reconfigurable FIR filter architectures with fast reconfiguration," in *2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pp. 1–8, IEEE, 2013.

[150] S. Y. Park and P. K. Meher, "Efficient FPGA and ASIC Realizations of a DA-Based Reconfigurable FIR Digital Filter," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 7, pp. 511–515, 2014.

[151] K. Gunasekaran and M. Manikandan, "Low power and area efficient reconfigurable FIR filter implementation in FPGA," in *2013 International Conference on Current Trends in Engineering and Technology (ICCTET)*, pp. 300–303, 2013.

[152] S. Prasanna, B. P. James, and V. Dhandapani, "High speed and area efficient coded input BCSM shared LUT-based FIR filter architecture," *International Journal of System Assurance Engineering and Management*, pp. 1–12, 2024.

[153] R. Jia, H.-G. Yang, C. Y. Lin, R. Chen, X.-G. Wang, and Z.-H. Guo, "A computationally efficient reconfigurable FIR filter architecture based on coefficient occurrence probability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 8, pp. 1297–1308, 2015.

[154] S. Yoshima, Y. Sun, Z. Liu, K. R. Bottrill, F. Parmigiani, D. J. Richardson, and P. Petropoulos, "Mitigation of nonlinear effects on WDM QAM signals enabled by optical phase conjugation with efficient bandwidth utilization," *Journal of Lightwave Technology*, vol. 35, no. 4, pp. 971–978, 2016.

[155] T. W. Parks and C. S. Burrus, *Digital filter design.* Wiley, 1987.

[156] M. Gerken, "On fixed-point quantization schemes," in *38th Midwest Symposium on Circuits and Systems. Proceedings*, vol. 1, pp. 350–353, IEEE, 1995.

[157] M. Haseyama and D. Matsuura, "A filter coefficient quantization method with genetic algorithm, including simulated annealing," *IEEE Signal Processing Letters*, vol. 13, no. 4, pp. 189–192, 2006.

[158] J. J. Nielsen, "Design of linear-phase direct-form FIR digital filters with quantized coefficients using error spectrum shaping," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, no. 7, pp. 1020–1026, 1989.

[159] T. Parks and J. McClellan, "Chebyshev Approximation for Nonrecursive Digital Filters with Linear Phase," *IEEE Transactions on Circuit Theory*, vol. 19, no. 2, pp. 189–194, 1972.

[160] J. Proakis and D. Manolakis, *Digital Signal Processing.* Prentice Hall, 2007.

[161] D. Kodek, "Design of optimal finite wordlength FIR digital filters using integer programming techniques," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 28, no. 3, pp. 304–308, 1980.

[162] B. W. Jung, H. J. Yang, and J. Chun, "Finite wordlength digital filter design using simulated annealing," in *2008 42nd Asilomar Conference on Signals, Systems and Computers*, pp. 546–550, IEEE, 2008.

[163] J. I. Ababneh and M. H. Bataineh, "Linear phase FIR filter design using particle swarm optimization and genetic algorithms," *Digital Signal Processing*, vol. 18, no. 4, pp. 657–668, 2008.

[164] E. Hewitt and R. E. Hewitt, "The Gibbs-Wilbraham Phenomenon: An Episode in Fourier analysis," *Archive for history of Exact Sciences*, pp. 129–160, 1979.

[165] L. Rabiner and K. Steiglitz, "The design of wide-band recursive and nonrecursive digital differentiators," *IEEE Transactions on Audio and Electroacoustics*, vol. 18, no. 2, pp. 204–209, 1970.

[166] University of Strathclyde - Software Defined Radio Research Laboratory, "rfsoc_qpsk." `https://github.com/strath-sdr/rfsoc_qpsk`. Accessed: Feb. 8, 2026.

[167] F. Gray, "Pulse code communication," *United States Patent Number 2632058*, 1953.

[168] Xilinx, Inc., *PG269, Zynq Ultrascale+ RFSoC RF Data Converter Gen 1/2/3*, Nov. 2020.

[169] M. Rice, *Digital communications: a discrete-time approach*. Pearson Education, 2009.

[170] Xilinx, Inc., *PG149, FIR Compiler v7.2, LogiCORE IP product guide*, Oct. 2022.

[171] Texas Instruments Incorporated, "Texas Instruments Clocks and Synthesizers (TICS) Pro Software." `https://www.ti.com/tool/TICSPRO-SW#related-design-resources`. Accessed: Feb. 8, 2026.

[172] AMD, Inc., "xrfclk.py." https://github.com/Xilinx/PYNQ/blob/93ddd21ab623883590a8c8b07f0b157b2855da4b/sdbuild/packages/xrfclk/package/xrfclk/xrfclk.py. Accessed: Feb. 8, 2026.

[173] F. Perez, B. E. Granger, and C. Obispo, "An open source framework for interactive, collaborative and reproducible scientific computing and education," 2013.

[174] Plotly, "Plotly Python Open Source Graphing Library." https://plot.ly/python/. Accessed: Feb. 8, 2026.

[175] J. P. Smith, J. I. Bailey, J. Tuthill, L. Stefanazzi, G. Cancelo, K. Treptow, and B. A. Mazin, "A High-Throughput Oversampled Polyphase Filter Bank Using Vivado HLS and PYNQ on a RFSoC," *IEEE Open Journal of Circuits and Systems*, vol. 2, pp. 241–252, 2021.

[176] D. Northcote, L. McLaughlin, L. H. Crockett, and R. W. Stewart, "Capture and Visualisation of Radio Signals with an Open Source, Single Chip Spectrum Analyser," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 387–387, 2021.

[177] C.-Y. Chang and H.-T. Chou, "FPGA Implementation of 5G NR PDSCH Transceiver for FR2 Millimeter-wave Frequency Bands," in *2022 IEEE 4th Eurasia Conference on IOT, Communication and Engineering (ECICE)*, pp. 60–63, IEEE, 2022.

[178] O. Reyhanigalangashi, D. Taylor, S. Kolpuke, D. N. Elluru, F. Abushakra, A. K. Awasthi, and S.-P. Gogineni, "An RF-SoC-based ultra-wideband chirp synthesizer," *IEEE Access*, vol. 10, pp. 47715–47725, 2022.

[179] J. K. Becker and D. Starobinski, "Snout: A Middleware Platform for Software-Defined Radios," *IEEE Transactions on Network and Service Management*, 2022.

[180] A. Taylor, "MicroZed Chronicles: Pynq on the RFSoC!." https://medium.com/@aptaylorceng/microzed-chronicles-pynq-on-the-rfsoc-eca22fc857fc, 2019. Accessed: Feb. 8, 2026.

[181] A. Taylor, "MicroZed Chronicles: RFSoC Studio." https://www.adiuvoengineering.com/post/microzed-chronicles-rfsoc-studio, 2021. Accessed: Feb. 8, 2026.

[182] A. Taylor, "MicroZed Chronicles: PYNQ, RFSoC & SDFEC." https://www.adiuvoengineering.com/post/microzed-chronicles-rfsoc-studio, 2021. Accessed: Feb. 8, 2026.

[183] Xilinx, Inc., "EmbeddedSW." https://github.com/Xilinx/embeddedsw. Accessed: Feb. 8, 2026.

[184] Python, "GlobalInterpreterLock." https://wiki.python.org/moin/GlobalInterpreterLock. Accessed: Feb. 8, 2026.

[185] Mazin Lab, "MKIDGen3." https://github.com/MazinLab/MKIDGen3. Accessed: Feb. 8, 2026.

[186] L. Stefanazzi, K. Treptow, N. Wilcer, C. Stoughton, C. Bradford, S. Uemura, S. Zorzetti, S. Montella, G. Cancelo, S. Sussman, *et al.*, "The QICK (Quantum Instrumentation Control Kit): Readout and control for qubits and detectors," *Review of Scientific Instruments*, vol. 93, no. 4, 2022.

[187] J. Goldsmith, "SPECTRE Frequency Planner Results Dataset and Codebase." https://pureportal.strath.ac.uk/en/datasets/8a4ee1df-ac13-459b-8ec2-2cf2bcff336d, 2025. Accessed: Feb. 8, 2026.

[188] University of Strathclyde - Software Defined Radio Research Laboratory, "RFSoC Frequency Planner." https://github.com/strath-sdr/rfsoc_frequency_planner. Accessed: Feb. 8, 2026.

[189] R. G. Lyons, *Understanding digital signal processing*. Prentice Hall, 2001.

[190] Xilinx Inc., "Versal ACAP DSP Engine Architecture Manual, AM004," Sept. 2022.