IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received June 17, 2020, accepted July 8, 2020, date of publication July 13, 2020, date of current version July 24, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3008954

Control and Visualisation of a Software Defined Radio System on the Xilinx RFSoC Platform Using the PYNQ Framework

JOSH GOLDSMITH, (Graduate Student Member, IEEE), CRAIG RAMSAY, DAVID NORTHCOTE, (Member, IEEE), KENNETH W. BARLEE, (Member, IEEE), LOUISE H. CROCKETT, AND ROBERT W. STEWART
Department of Electronic and Electrical Engineering, University of Strathclyde, Glasgow G1 1XW, U.K.

Corresponding author: Josh Goldsmith (joshua.goldsmith@strath.ac.uk)

The work of Craig Ramsay was supported by the Engineering and Physical Sciences Research Council (EPSRC) under Grant EP/N509760/1.

ABSTRACT The availability of commercial Radio Frequency System on Chip (RFSoC) devices brings new possibilities for implementing Software Defined Radio (SDR) systems. Such systems are of increasing interest given the pace of innovation in wireless technology, and the pressure on RF spectrum resources, leading to a growing need to access the spectrum in more dynamic and innovative ways. In this paper, we present an SDR demonstration system based on the Xilinx RFSoC platform, which leverages the Python-based 'PYNQ' (Python Productivity for Zynq) software framework. In doing so, we highlight features that can be extremely useful for prototyping radio system design. Notably, our developed system features Python-based control of hardware processing blocks and Radio Frequency (RF) data converters, as well as direct visualisation of communications signals captured within the chip. The system architecture is reviewed, hardware and software components are discussed, functionality is demonstrated, and aspects of the system's performance are evaluated. Finally, it is noted that this combined RFSoC + PYNQ approach is readily extensible for other SDR systems; we highlight our online shared resources, and invite other engineers to investigate and build upon our work.

INDEX TERMS Software defined radio, SDR, RFSoC, system-on-chip, python, PYNQ, ZCU111.

I. INTRODUCTION

Software Defined Radio (SDR) has gathered interest in recent years, particularly with the advancing capabilities of baseband processing technology. SDR refers to radios whose behaviour is implemented or controlled in some manner via software. The ability to dynamically change modulation scheme, carrier frequency, or some other aspect of the radio's operation is compelling, especially in the context of next generation cognitive and Dynamic Spectrum Access (DSA) schemes, where a high degree of adaptability is required. SDR features can also be extremely useful at the prototyping stage, when programmable functionality can be leveraged to accelerate the development and testing processes.

The implementation of an SDR system requires suitably capable hardware devices for radio signal processing, paired with software for orchestration. Often, software is also used to directly implement some of the required signal processing functionality. Field Programmable Gate Arrays (FPGAs) and FPGA-based System on Chip (SoC) devices provide a natural platform for SDR, given their highly reprogrammable nature, capability to support high data rates and multiple channels, and the availability of processors for running software.

A significant moment in the development of SoC technology was the release of the Xilinx Zynq UltraScale+ RFSoC in 2017. This device is an Integrated Circuit (IC) that incorporates FPGA Programmable Logic (PL), a Processing System (PS) based on Arm applications and real-time processors, a set of high throughput Soft-Decision Forward Error Correction (SD-FEC) blocks, and a set of high speed Radio Frequency (RF) data converters (RF Analogue-to-Digital and Digital-to-Analogue Converters (RF-ADCs and RF-DACs)), sampling at multiple GHz [1]. With such high sampling rates, the analogue/digital interface of the radio

The associate editor coordinating the review of this manuscript and approving it for publication was Remigiusz Wisniewski.

system can, in many cases, be pushed to the very front of the radio system, enabling conversion between the analogue and digital domains at RF frequencies. Therefore, modulation and demodulation between baseband and RF can be undertaken entirely in the digital domain, which removes the need for any external (analogue) RF or Intermediate Frequency (IF) mixing stages. The ability to perform all processing digitally, on the same device as the data converters, enables high precision, deterministic operation.

This paper focuses on software control of, and interaction with, the radio signal processing hardware (in other words, the RF data converters and functionality implemented in the PL). Specifically, we demonstrate that the PS can be leveraged to achieve dynamic user control through software, and facilitate the visualisation of signals captured from the device. Software control is inherent to Software Defined Radio, and permits enhanced flexibility and operational adaptability; meanwhile, visualisation of captured signals is especially useful for prototyping, debugging, demonstration and educational purposes.

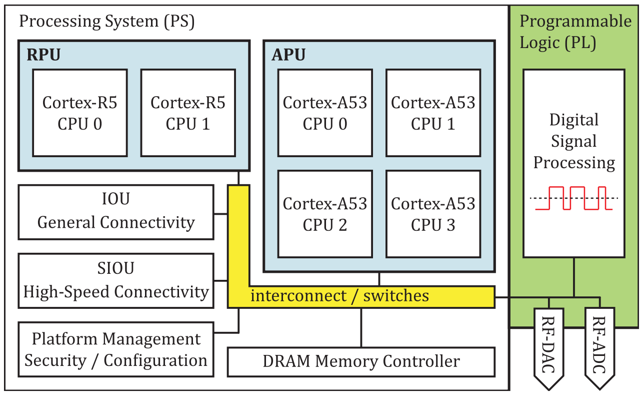


FIGURE 1. Overview of the Zynq RFSoC architecture (specifically, the XCZU28DR-2E device architecture is shown).

Fig. 1 presents an overview of the RFSoC, based on the architecture of the XCZU28DR-2E device (the target for the work described in this paper, and hereafter referred to simply as 'RFSoC'). In the PS, there is an Applications Processing Unit (APU), which houses an Arm Cortex-A53 Multi-Processor Core (MPCore) consisting of four CPUs. The APU is suitable for hosting operating systems such as Linux and FreeRTOS, and executing drivers for peripheral interfaces such as Ethernet and USB. The Real-Time Processing Unit (RPU) consists of two Arm Cortex-R5 processor cores, which enable low latency and deterministic processing capabilities. The PS also contains a Dynamic Random Access Memory (DRAM) controller for communicating with external memory chips and an Input Output Unit (IOU) that is responsible for general connectivity to external peripherals. The IOU encloses several peripheral interface cores, which include a range of different digital interface standards, memory interface protocols, and Ethernet Media Access Controller (MAC) and Universal Serial Bus (USB) 3.0 interfaces. There is also a Serial IOU (SIOU) for high speed connectivity, a dedicated platform management unit, and security configuration cores. Finally, the PL contains Kintex UltraScale+ FPGA logic fabric, which includes both RF-ADC and RF-DAC data converters, and SD-FEC hard silicon Intellectual Property (IP) blocks. There are 8x RF-ADCs and 8x RF-DACs in the XCZU28DR-2E device, which can operate at up to 4096 Msps and 6554 Msps respectively. The data converter blocks are integrated subsystems that each contain a mixer with a programmable Numerically Controlled Oscillator (NCO) for modulation/demodulation, along with a programmable decimation filter (for the RF-ADCs) or interpolation filter (for the RF-DACs) [2], shown in Fig. 2. The 8x SD-FECs available on the device allow for encoding of low density parity check (LDPC) codes at a rate of up to 20 Gb/s, and decoding of both LDPC and Turbo codes at a rate of up to 3 Gb/s and 1.8 Gb/s respectively [1].

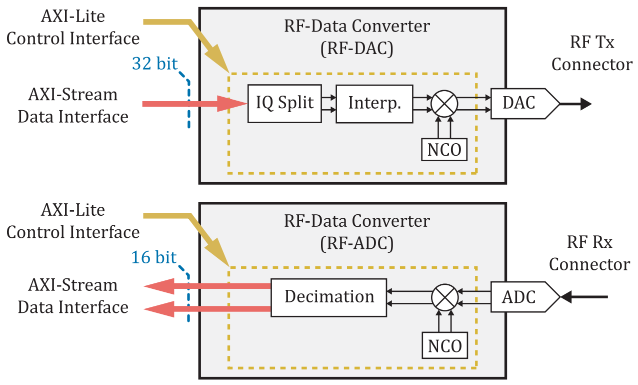

AXI-Lite Control Interface
32 bit
AXI-Stream Data Interface
RF-Data Converter (RF-DAC)
IQ Split
Interp.
DAC
NCO
RF Tx Connector
AXI-Lite Control Interface
16 bit
AXI-Stream Data Interface
RF-Data Converter (RF-ADC)
Decimation
ADC
NCO
RF Rx Connector

FIGURE 2. High level diagram of the RF DACs (top) and ADCs (bottom), in the configuration used in our design, showing the mixer and rate conversion stages.

Software control of SDR systems is often achieved by developing custom software. This provides flexibility, but with a high degree of design effort and cost, both in terms of developing a proprietary solution and subsequently maintaining it. A more desirable scenario would be to exploit an existing software framework that reduces the design effort involved, and provides access to reusable components. We therefore choose to employ PYNQ (Python Productivity for Zynq), which is an open-source, Python-based framework that can be deployed on all Xilinx SoC devices [3]. PYNQ features a software stack that resides on the PS portion of the Zynq RFSoC device, and facilitates user interaction via a network connection and a standard web browser.

Fig. 3 shows the structure of the PYNQ project and separates out the contributions of this paper. All pink coloured elements are existing PYNQ components provided by Xilinx, while the blue elements are existing non-PYNQ components. Green elements are outputs of this paper, developed by the authors. Finally, all of the green/pink striped elements are outputs of this paper, working towards PYNQ support for RFSoC, produced in collaboration with Xilinx. Some of the terms introduced in Fig. 3 may be new to most readers, and will be clarified in the following sections.

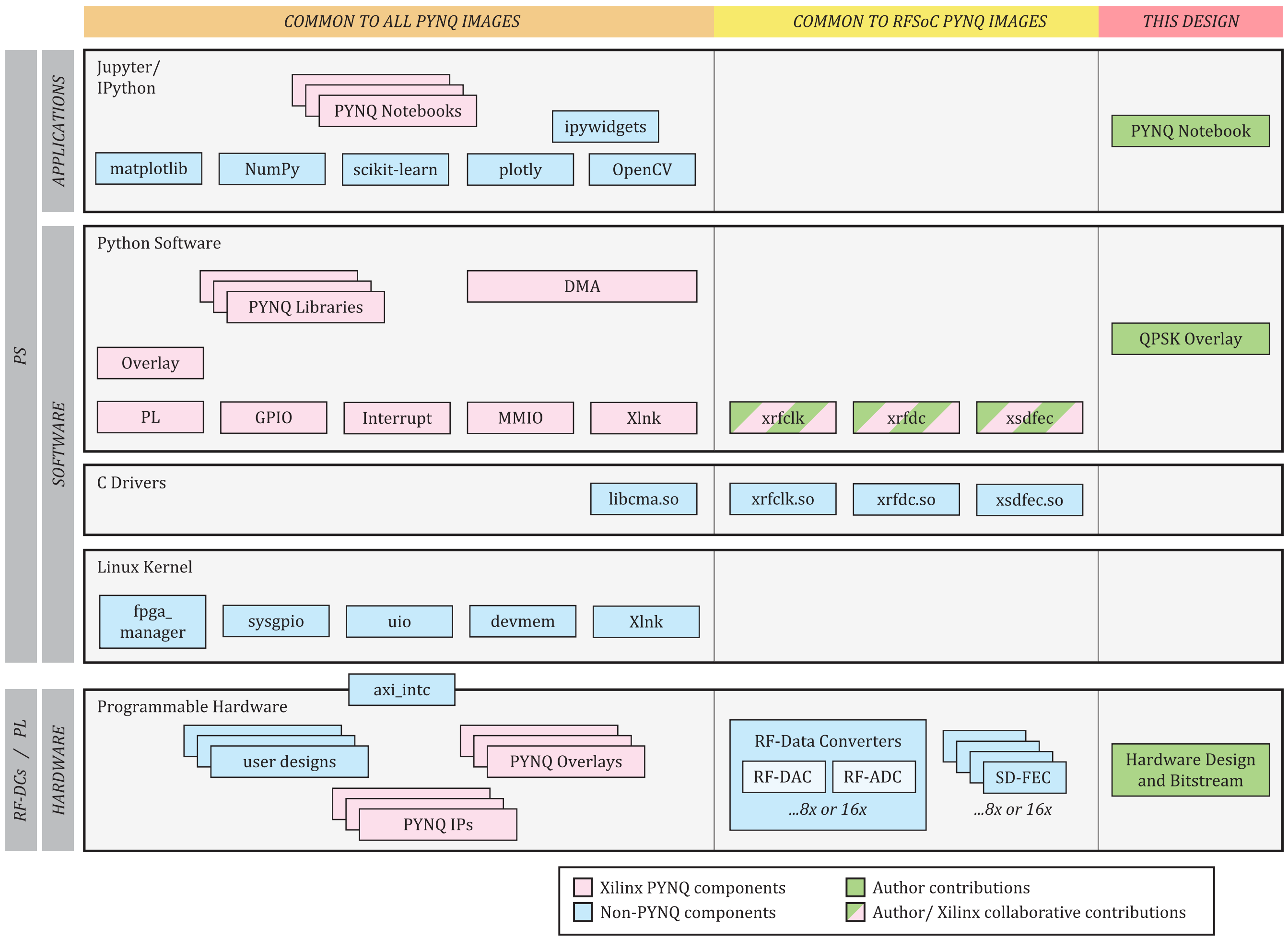


FIGURE 3. Overview of the existing PYNQ ecosystem, our collaboration with Xilinx for PYNQ's RFSoC support, and the QPSK design presented in this paper.

A. APPLICATIONS AND USE CASES FOR THE RFSoC

The RFSoC has a number of potential applications, including in telecommunications (for both wired and wireless technologies), instrumentation and measurement, phased array Radar, and in the evolving LiDAR/ 3D imaging sector. In particular, with the array of high speed RF Data Converters (RF-DCs), dedicated SD-FEC cores and large FPGA for parallel processing, it could be argued that the RFSoC is the ideal device to enable 5G (and beyond) wireless infrastructure.

In 4G and 5G cellular applications, for example, the ultra-wide operating range of the RFSoC's multi-channel RF-DCs enables direct RF support for all Sub-6 GHz mobile bands; and the device can also facilitate implementations of Next Generation- Radio Access Network (NG-RAN) split architectures, and aid in the generation of mmWave (>24 GHz) 5G-New Radio (5G-NR) signals.

Fig. 4 compares traditional 4G RAN and 5G split NG-RAN network architectures. The 4G RAN features two main components: the BaseBand Unit (BBU), which implements the cellular stack; and the Radio Unit (RU), which performs RF modulation. The unique architecture of the RFSoC could allow it to operate as a combination of the BBU and RU all-in-one, and subject to appropriate analogue RF signal conditioning (analogue filters and amplifiers attached to the various RF-DC inputs and outputs), a single RFSoC could operate as a multi-cell, multi-band, Multiple Input Multiple Output (MIMO) basestation, where traditionally multiple BBUs and RUs are required for each cell and band. And, with up to 16x RF-DACs and 16x RF-ADCs, the RFSoC can also support *massive* MIMO.

In 5G NG-RAN split architecture networks, the part of the stack traditionally implemented on a BBU is split across a Control Unit (CU) and a Data Unit (DU), in regional and edge data centres. The numerous computing resources combined with the high speed Evolved Common Public Radio Interface (eCPRI) interfaces and SD-FEC modules mean the RFSoC is a suitable solution for each of these components. As with the 4G configuration, when generating Sub-6 GHz 5G-NR signals, the RFSoC can directly support all bands. For mmWave (>24 GHz) 5G-NR signals, the RFSoC can be used as an IF-SDR, to generate multi-GHz wide signals for subsequent modulation onto mmWave carriers [6].

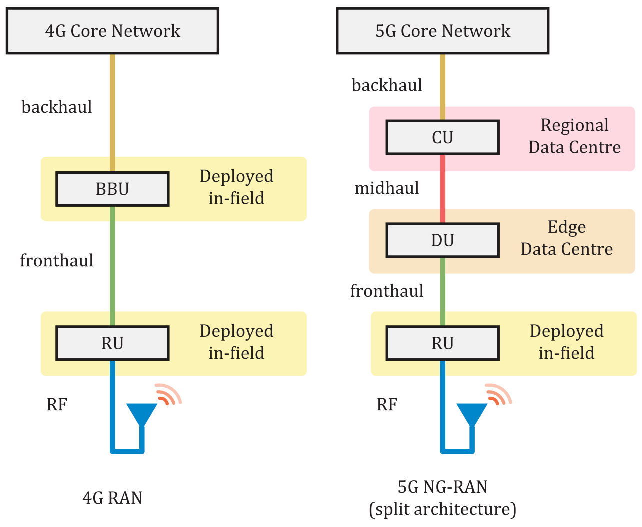


FIGURE 4. **A comparison of 4G RAN and 5G NG-RAN cellular network architectures. The numerous computing resources combined with the high speed interfaces mean the RFSoC is a suitable solution for many of these components. (For further information on these RAN architectures, please see [4]–[6]).**

With its wide operating bandwidth and flexible radio front end, combined with the tight integration of the PS, PL and RF-DCs, the RFSoC is also an enabler for cognitive DSA-radio applications. The RFSoC could be used to implement autonomous radios which are spectrally aware, and able to make decisions about how best to configure themselves to transmit and receive information in the given radio environment. The SDR could choose which vacant spectrum to target, what channel bandwidths and modulation schemes to use, and other parameters such as the most suitable transmit power, encryption technique and error coding scheme to use. All of this could be carried out by a single-chip RFSoC radio node.

In cabled communication systems, the RFSoC is well suited to the development of next-generations of Hybrid Fibre Communications (HFC) and Data Over Cable Service Interface Specification (DOCSIS) 'cable broadband' networks. It also has a place in V2X and autonomous vehicle systems. The RFSoC ICs could be used for instrumentation and measurement too, for example in wideband oscilloscopes or spectrum analysers, and our work in this paper is relevant to these applications. Here, we consider the combination of PYNQ and RFSoC for the development and testing of an SDR system, highlighting the potential of this combination of technologies for run-time introspection and performance evaluation of digital radio systems.

B. RELATED SDR TECHNOLOGY

At the time of writing, single-IC RFSoC devices are a recent innovation. A number of other SDR platforms are available, with different characteristics, many of which have been investigated for system prototyping and evaluation. These platforms range from low-cost options (at a level accessible to students and hobbyists), to more expensive, professional grade SDR development equipment. It is the higher end of the spectrum that is most comparable to this work.

Professional SDR hardware encompasses both standalone development kits, and modules that are interfaced with an FPGA or SoC development board to form a complete system. Examples of professional SDR hardware includes the set of Universal Software Radio Peripheral (USRP) devices from Ettus Research [7], the PicoSDR series from Nutaq [8], the LimeSDR series from Lime Microsystems [9], and the RF SOM (System on Module) and set of FMCOMMS modules produced by Analog Devices [10], [11]. FMCOMMS modules can be connected to an FPGA/Zynq development board that is equipped with an FPGA Mezzanine Card (FMC) port, such as the ZedBoard [12] or ZC706 [13].

The factor differentiating the RFSoC development platform from the above mentioned SDRs is its status as a single IC radio—the others feature two major components: an FPGA or SoC for processing at baseband and IFs, paired with a separate IC for front-end SDR functionality (the front-end SDR IC performs the task of modulating and demodulating signals between IF and RF, with associated filtering). The single-IC opportunities of the RFSoC can be exploited to achieve control and visualisation at all stages of the transmit (Tx) and receive (Rx) signal paths (i.e. baseband, IF and RF).

Prominent software environments for SDR design and prototyping include MathWorks professional tools (incorporating MATLAB, Simulink, and its toolboxes and support packages) [14]; and the open-source GNURadio [15] and Pothos tools [16]. National Instruments also offers SDR functionality within its LabView environment, and notably includes support for MIMO systems in conjunction with its SDR hardware products [17]. Furthermore, many of these products include example designs, which are compatible with many of the SDRs mentioned above; such as the QPSK transceiver for the AD9361 offered by Mathworks [18], [19].

In our work, the MATLAB and Simulink environment was used (in conjunction with the System Generator design tool provided by Xilinx), for designing and simulating transmit and receive signal processing functionality. The embedded software components of an SDR can be custom-developed using these tools, or in the case of prototyping, are often provided as part of the software support for a particular platform. In our case, the PYNQ software framework [3] was leveraged, third party open-source libraries were included, and custom software was developed.

Beyond the desktop task of developing a single Tx-Rx system, dedicated large scale experimental facilities may be required for validation in more complex scenarios. For instance, multiple testbeds have been established as part of the Orchestration and Reconfiguration Control Architecture (ORCA) project, enabling a variety of experiments to be performed using SDR hardware and software [20], [21]. At the time of writing, ORCA testbeds do not feature RFSoC capabilities, but inclusion of this technology would be a natural progression for related research.

C. LITERATURE REVIEW

The term *Software Defined Radio* refers to radio systems in which some or all of the Physical Layer (PHY) components traditionally implemented with dedicated hardware (e.g. mixers, filters, synchronisation circuitry) are instead realised using DSP algorithms implemented in software or on programmable hardware [22], [23]. There is much interest in the research community around the effective design of SDR systems, in particular the joint programmable hardware/software co-design of such systems. For instance, researchers in [24] discuss their approach to an SDR design, based on the Analog Devices FMCOMMS3, ZedBoard/ZC706, and MathWorks and Xilinx software tools. Other studies have investigated the potential of dynamic partial reconfiguration techniques for implementing flexible SDRs [25].

SDR techniques have also been proven to be effective outwith the traditional domain of radio communication for applications such as ground penetrating radar in [26], and ultrasound transceivers in [27].

Certain aspects of functionality that are similar to our design have been implemented in other products, or investigated by other researchers. For instance, hardware acceleration of the FFT (Fast Fourier Transform) algorithm needed to perform spectrum analysis is an established idea, and a number of published works have reported on SDR implementations based on the SDR platforms mentioned earlier (and others) that use this technique [28]–[30].

However, the themes we explore in this paper, namely control and visualisation via software hosted on the same IC, were not considered by any of these earlier studies. Additionally, the RFSoC device adopted in this work inherently provides a set of enhanced capabilities, compared to other available SDR platforms, which have received little attention in academic research to date.

With that said, some research has been published that involves designs targeted to RFSoC devices; such as RF modulation classification in [31], and digital beamforming in [32], [33] but these, similarly, do not explore the central themes of this paper.

Perhaps the closest comparable work, in terms of signal visualisation and user interaction, relates to the Red Pitaya platform. Users of the Red Pitaya have the option of interaction via Jupyter Notebooks (discussed in Section IV-C1) for functionality such as signal generation and spectral analysis, which are hosted on the Zynq processor in a similar manner to our system, and its use has also been demonstrated for SDR applications [27], [34]. However, the Red Pitaya does not adopt the PYNQ software framework, its RF capabilities are very limited compared to the RFSoC, and it does not constitute a single IC solution.

The software-based aspects of our design were created using PYNQ, which was first released in 2016. To date, PYNQ has been used for a variety of applications and research studies, including image processing [35] and machine learning [36]. The PYNQ framework has also proved effective in teaching, as it accelerates the development process and provides students with a direct, intuitive interface that leverages the popularly taught Python language. However, to date, PYNQ has not featured prominently in communications research.

To the best of the authors' knowledge, our work is the first that has employed and evaluated PYNQ for SDR applications, and also the first to use PYNQ on the RFSoC. We seek to exploit the same fundamental benefits (particularly in terms of productivity, accessibility, and direct interaction with hardware) as have been noted in other academic studies for our radio design scenario.

D. OBJECTIVES

In this paper, we present a design that combines RFSoC technology with the PYNQ framework to achieve a full Tx-Rx SDR demonstrator system. As established above, this represents a new combination of technologies which has not been investigated before. Our objectives in doing so are to:

- Prove the concept that control and internal signal inspection can be implemented using a single-IC radio system, by pairing the RFSoC and PYNQ.
- Develop and evaluate the additional hardware infrastructure that requires to be incorporated into an SDR design for use with PYNQ.
- Evaluate open-source software libraries and features for programming, interaction, and visualisation.
- Evaluate the SDR aspects that are enabled via PYNQ, in terms of functionality, performance, and user interaction.
- Release the design as an open-source project, available online, enabling the community to learn from, and build upon, our work [37].

E. PAPER ORGANISATION

Key elements of the system design are presented in the remainder of this paper, which is organised as follows: In Section II, an overview of the system architecture is provided; Sections III and IV present the hardware and software portions of the design, respectively; Section V discusses the configuration of the demonstrator system and evaluates the results obtained; and the paper is concluded in Section VI.

II. SYSTEM ARCHITECTURE OVERVIEW

The RFSoC is a single IC device comprising of Arm applications and real-time processors coupled to FPGA logic fabric and high speed RF-DCs; using the AXI-4 (Advanced eXtensible Interface) protocol. The RF-DCs feature two pairs of eight RF-DACs and RF-ADCs that operate up to 4096 Msps and 6554 Msps respectively. As the purpose of our radio system is to investigate software control and signal visualisation using RFSoC and PYNQ, it is important to stress that we do not use the full capabilities of the RFSoC device in terms of maximising throughput or sample rate.

For the purpose of demonstrating our proof-of-concept SDR system using PYNQ, we use a simple and easily recognisable communications scheme with a low data rate.

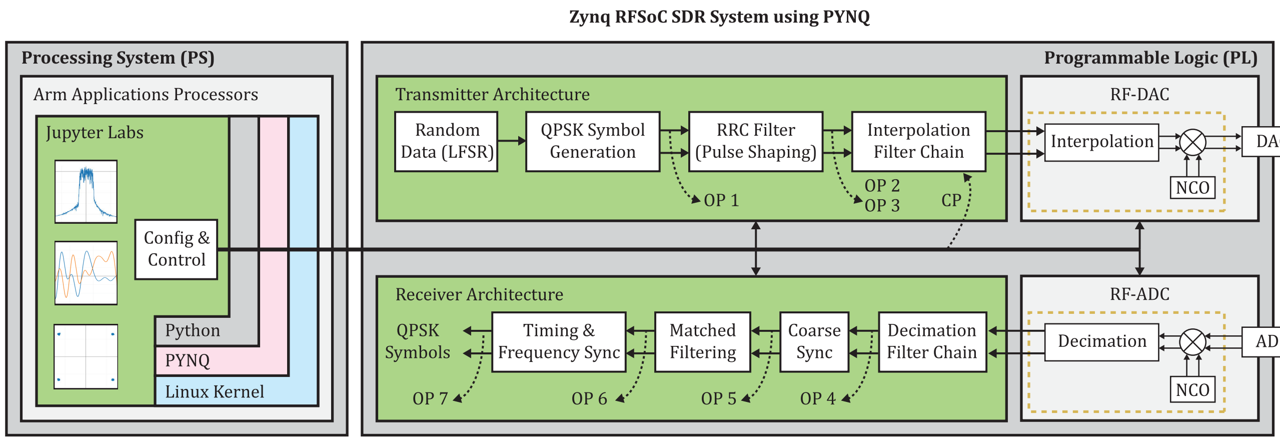


FIGURE 5. Overview of our SDR system on the RFSoC with PYNQ. (Note, this is a functional block diagram, and does not show the IQ split in the RF-DAC).

Hence, Quadrature Phase Shift Keying (QPSK) with a bit rate of 1 kb/s has been adopted. As a single channel example, only one RF-ADC and RF-DAC pair is needed from the bank of eight available on the target device.

Our radio architecture is implemented entirely using the ZCU111 development kit [38], which includes the ZCU111 development board and XM500 add-on board [39]. The XM500 add-on board contains a set of SMA interfaces and RF baluns, which convert between differential RF signals and single-ended RF signals. It is important to note that apart from the RF baluns, there are no other electronic components required between the RFSoC and SMA interfaces. All RF signal processing is performed entirely in the XCZU28DR-2E RFSoC device on the ZCU111 development board.

Our radio design is partitioned across the PL and PS portions of the RFSoC in order to leverage the unique capabilities of each resource. The PL is responsible for accelerating Digital Signal Processing (DSP) algorithms and interfacing to the RF data converter. The PS executes the PYNQ software stack to achieve dynamic user control, analysis, and real-time inspection of our radio system.

An overview of the entire radio architecture is illustrated in Fig. 5. As shown, the QPSK transmitter and receiver are implemented in the PL of the same RFSoC device, while the PYNQ software stack is contained within the PS. The transmitter and receiver RF data paths are independent of one another, and do not share clocks or signals.

The remainder of this section outlines the radio architecture and describes the role of PYNQ in our system. Further technical details are provided in Sections III and IV, where we explain key design concepts of both.

A. PL SYSTEM OVERVIEW

The PL implements a sequence of DSP algorithms that form the basis of our radio transmitter and receiver. These algorithms are shown on the right-side of Fig. 5, where they are directly connected to the RFSoC's data converters for transmitting or receiving data.

The transmit chain consists of random symbol generation, Root Raised Cosine (RRC) pulse-shaping, and multi-stage interpolation—increasing the data rate from 1 kb/s to 128 Msps—before being transferred to the RF-DAC.

The RF-DAC contains a Digital Upconverter (DUC) that interpolates the signal by a factor of 8 to achieve 1024 Msps. The interpolated signal passes through a complex RF mixer, which performs modulation to an RF carrier. The carrier frequency is dynamically selected at run-time using the PYNQ framework. Finally, the modulated signal is converted from digital to analogue for RF transmission.

At the receive side, the signal is digitised by the RF-ADC at a rate of 1024 Msps. It is then demodulated and decimated by a factor of 8, using its internal Digital Downconverter (DDC). Although the receiver is implemented on the same device as the transmitter, it is not possible to share clocks between the RF-DACs and RF-ADCs. Therefore, a full receiver that includes carrier synchronisation and symbol timing synchronisation is required.

The signal from the RF-ADC is passed through a reciprocal, multi-stage decimation reducing the rate to 4 ksps, followed by coarse frequency synchronisation and a matched filter. Finally, timing synchronisation is then performed to recover the received QPSK symbols.

As illustrated in Fig. 5, signals in our radio architecture are marked for control or visualisation using Control Points (CPs), and Observation Points (OPs) respectively. These points within our radio system are interfaced to the PYNQ framework, allowing us to manipulate or observe a signal in the architecture. For example, the transmitter contains an OP after the RRC filter. This OP can be used to obtain time or frequency domain samples of the pulse-shaped signal for inspection.

B. PS SYSTEM OVERVIEW

PYNQ is an umbrella term encompassing an entire software ecosystem for Xilinx SoC devices, which includes the operating system, a Python software package, and a Jupyter web server.

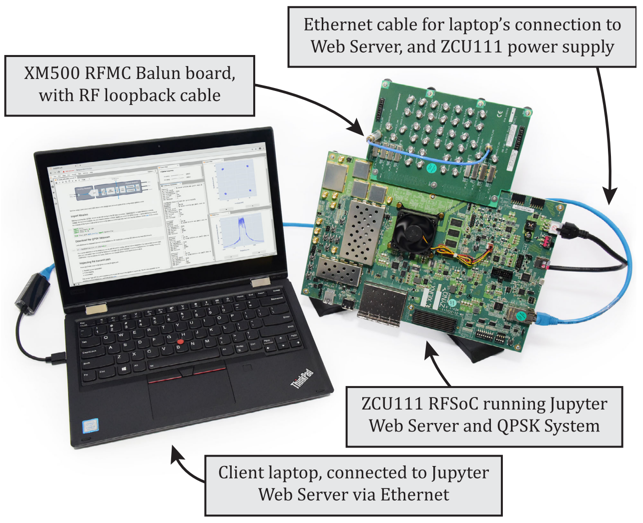

Ethernet cable for laptop's connection to Web Server, and ZCU111 power supply
XM500 RFMC Balun board, with RF loopback cable
ZCU111 RFSoC running Jupyter Web Server and QPSK System
Client laptop, connected to Jupyter Web Server via Ethernet

FIGURE 6. Setup of our SDR system on the RFSoC with PYNQ. Note the use of a loop-back cable to connect the transmitter and receiver.

The PYNQ operating system is an Ubuntu for Arm-based Linux distribution, running Xilinx's own version of the Linux kernel, containing a collection of both off-the-shelf and custom software packages. This collection of software enables the control and management of PS and PL interfaces, memory, GPIO, interrupts, and more—the core of which is the PYNQ package, written in the Python language—giving users the ability to interact with this low-level functionality within an easy-to-use framework.

One immediate benefit of this type of framework is the ability to easily write drivers for custom hardware on the PL—facilitated by the use of the Overlay system within PYNQ. The Overlay not only contains the FPGA bitstream, but also incorporates the hardware description file for a design, which is parsed at run-time and exposes the register map for all available IPs. This register map can then be used to control IP functionality, with code written entirely in Python.

At the top of the PYNQ software stack is the Jupyter-Lab environment, which provides a Graphical User Interface (GUI) for programming and interaction. We exploit the interactive capabilities of Jupyter to provide control and visualisation of our radio system. The Jupyter environment is also responsible for configuring the RFSoC data converters and other hardware elements. JupyterLab resides on a web server on the PS, and is accessed over an Ethernet link via a standard web browser. The system setup is shown in Fig. 6.

Note that, beyond a standard web browser, no software packages or libraries need to be installed on the client laptop. This is a direct consequence of PYNQ's web-server based interface. All software packages that run on the ZCU111 board are included in the PYNQ v2.5 image as standard (the contents of which were made by Xilinx in collaboration with the authors, and include the work discussed in this paper), and we make particular use of JupyterLab as an interface, Plotly for interactive graphs, and ipywidgets for graphical controls.

As shown by Fig. 6, the client laptop can display visualisations of signals from the QPSK design running on the ZCU111 board. It is important to stress that the transfer of signal data from ZCU111 to client laptop occurs only when requested by the user. These visualisations are for introspection purposes (i.e. debugging or educational) — the whole DSP chain operates continuously on the PL of the RFSoC and does not rely on any processing on the client laptop. Section IV-C3 discusses the communication flow and data format for these transfers.

III. HARDWARE DESIGN

This section outlines the design of the system hardware, including the IP blocks that perform the main DSP functionality of the transmitter and receiver, and the overall hardware system design. Further, we highlight the additional hardware infrastructure that was incorporated to permit data capture from the signal processing chains for subsequent visualisation using PYNQ.

A. RADIO TRANSMITTER AND RECEIVER IP CORES

The IP blocks for the transmitter and receiver were each developed as block-based designs using the Xilinx System Generator tool, which resides inside Simulink. Simulations were conducted to verify the correct operation of the blocks, and Hardware Description Language (HDL) IP cores were generated for each system. Two approaches were taken when designing these cores in order to compare driver development in PYNQ: the transmitter was designed as a monolithic system, whereas the receiver was separated into more discrete functionality—this is further discussed in Section IV. A block diagram representation of the transmit and receive logic is shown in Fig. 5, and is explained in detail in this section.

The first stage in the Tx IP block is a LFSR, which generates random binary data at a rate of 1 kb/s. This is followed by Gray encoding to form QPSK symbols at 500 Sym/s. The In-Phase (I) and Quadrature-Phase (Q) symbols thereafter (separately) pass through interpolation chains comprised of: (i) an RRC filter, interpolating by 4; (ii) a halfband (HB) filter interpolating by 2; (iii) a CIC compensation filter (CFIR) interpolating by 2; and (iv) a 5th order CIC filter, interpolating by 3200. The overall interpolation ratio is therefore 51,200, upsampling to a rate of 25.6 Msps. These interpolated I and Q signals are then concatenated to form the Tx IP outputs required by the RF-DAC IP input.

The Rx system was designed by creating separate IPs for each stage of the receiver that include:

- A decimation chain acting on the I and Q signals separately. This performs an overall rate reduction from 25.6 Msps to 4 ksps, using two pairs of 3rd order CIC filters and CFIRs (decimating by 40 and 2, respectively).
- Coarse frequency synchronisation, which is performed using an FFT-based method.
- Frequency correction is performed by passing the signal through single rate RRC matched filters, and then interpolating by a factor of 4 using two cascaded HBs.

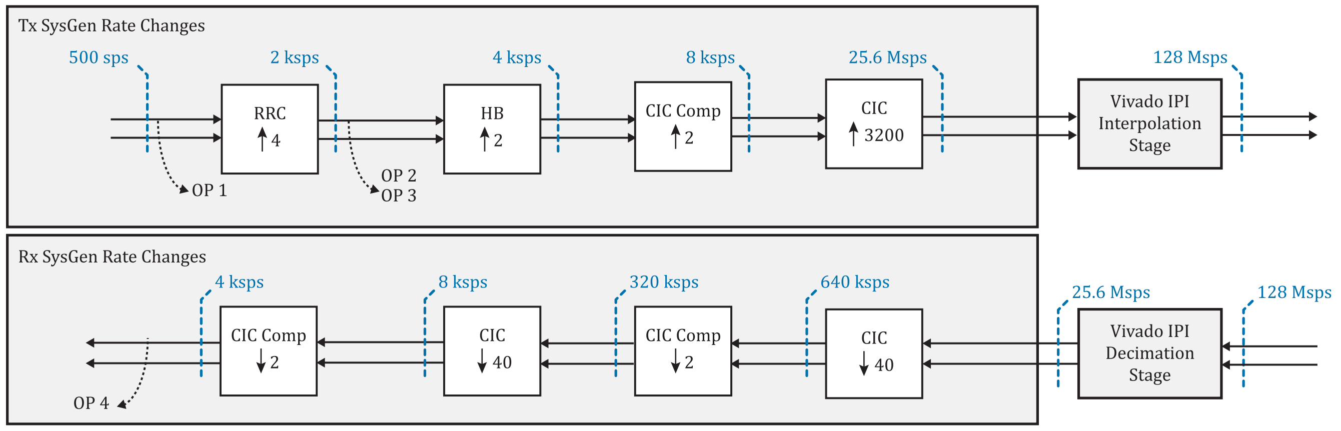


FIGURE 7. Block diagram of the sample rate changes in the Tx interpolation stage (top) and Rx decimation stage (bottom).

At this point, the I and Q signals are oversampled by a factor of 32 with respect to symbol rate.

- Timing and fine frequency synchronisation is performed using an architecture similar to that in [40] and [41].

The resulting symbol samples, along with a qualifying valid strobe, provide the final, AXI-Stream output. Note that the sample rate changes in the receiver were not chosen with efficiency in mind, but rather to enable the observation of a desired set of signals. The interpolation employed in the receiver permits an oversampled method to be adopted for the timing synchroniser. The requirements for interpolation at this stage could be removed by incorporating a ≈ 2x symbol rate timing synchroniser, and this is an intended item for future work.

CIC filters are implemented using the CIC Compiler 4.0 block, and all others as polyphase Finite Impulse Response (FIR) filters using the FIR Compiler 7.2 block. A block diagram illustrating the rate changes in the interpolation and decimation stages is shown in Fig. 7. Wordlengths are generally reduced to 16 bits at each stage, with appropriate gains inserted to maximise use of the available dynamic range.

In addition to this primary communications functionality, each IP includes additional circuitry for data capture and visualisation, which is described in Section III-C.

B. VIVADO HARDWARE SYSTEM DESIGN

The PL design was developed using Xilinx Vivado IP Integrator (IPI), building around the custom Tx and Rx IPs created in System Generator. Separate, hierarchical subsystems for the Tx and Rx IPs were used in order to lower the complexity of the top-level design, as well as to simplify reference to the IPs within the PYNQ framework—discussed in Section IV. Each hierarchy contains multiple AXI-Stream Direct Memory Access (DMA) cores pertaining to the OPs of the Tx and Rx IPs, allowing data to be streamed from the PL to PS via the off-chip PS Dynamic Random Access Memory (DRAM).

At the top level, the Zynq UltraScale+ MPSoC IP block is configured with two slave and two master

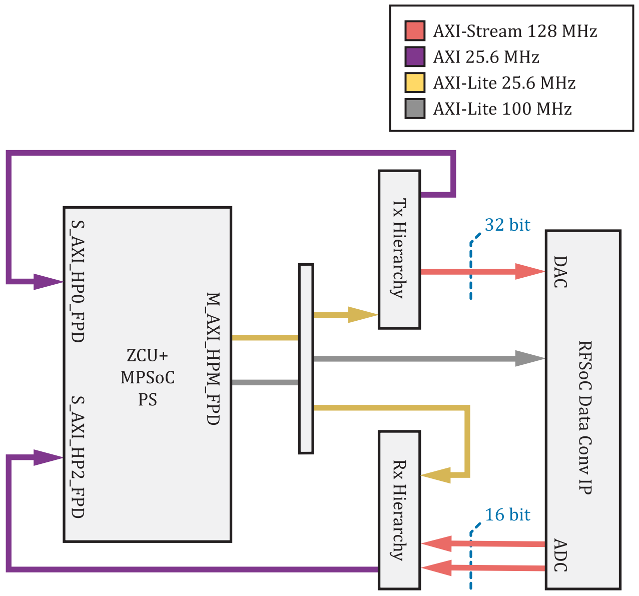


FIGURE 8. High-level Vivado IPI top-level diagram.

High-Performance (HP) ports. The HP slave ports serve the Tx and Rx IP DMAs routed through an AXI SmartConnect IP, while the HP master ports serve the AXI-Lite connections throughout the design. A block diagram of the top-level design is provided in Fig. 8.

The RF data converters have a minimum sample rate of 1000 Msps, therefore additional sample rate conversion was required between the connections of the Tx/Rx IPs and the respective DAC/ADC. Within the Tx hierarchy, an additional stage of interpolation was performed, upsampling by a factor of 5 to a rate of 128 Msps using a FIR Compiler and First-In First-Out (FIFO) buffer pair. A further 8x is achieved by making use of the data converter's built-in interpolation mode, upsampling to a rate of 1024 Msps. A reciprocal approach was taken for the Rx IP/ADC logic.

Three distinct clock domains are used: a 100 MHz PL clock and two, 409.6 MHz off-chip reference clocks generating a

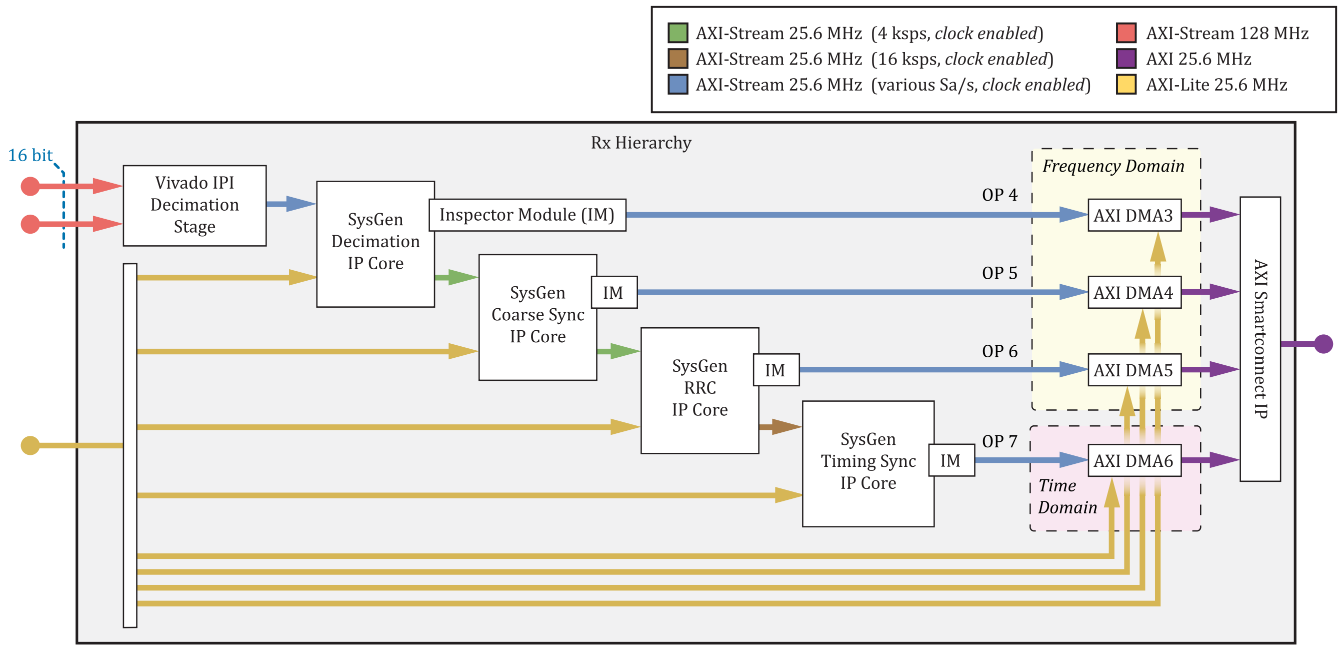

🟩 AXI-Stream 25.6 MHz (4 ksps, clock enabled)
🟫 AXI-Stream 25.6 MHz (16 ksps, clock enabled)
🟦 AXI-Stream 25.6 MHz (various Sa/s, clock enabled)
🟥 AXI-Stream 128 MHz
🟪 AXI 25.6 MHz
🟨 AXI-Lite 25.6 MHz
Rx Hierarchy
16 bit
Vivado IPI Decimation Stage
SysGen Decimation IP Core
Inspector Module (IM)
SysGen Coarse Sync IP Core
IM
SysGen RRC IP Core
IM
SysGen Timing Sync IP Core
IM
OP 4
OP 5
OP 6
OP 7
Frequency Domain
Time Domain
AXI DMA3
AXI DMA4
AXI DMA5
AXI DMA6
AXI Smartconnect IP

FIGURE 9. High-level Vivado IPI Rx hierarchy diagram.

64 MHz clock for the RF-ADC and a 128 MHz clock for the RF-DAC. A 128 MHz clock is derived from the 64 MHz clock in order to serve the AXI-Stream outputs of the RF-ADC, as well as the initial-stage Rx decimation. A 25.6 MHz clock is also derived to serve the AXI-Stream IPs in the Rx hierarchy. A similar approach is taken for the clocks derived for the transmit path—although the step to convert from 64 MHz to 128 MHz is not required, allowing us to free up Mixed-Mode Clock Manager (MMCM) resources, by using a Phase Locked Loop (PLL) for the clock down-sampling to 25.6 Msps instead. The PLL has a minimum input clock frequency of 70 MHz and therefore is not viable for the RF-ADC data path [42]. The rationale behind the mixture of clock resources is discussed in Section V.

Simplified block diagrams of the Tx and Rx hierarchy subsystems are provided in Figs. 9 and 10 respectively, containing the elements discussed in this section. For full details of the IPI design, the reader is encouraged to view the source code available at [37].

C. DATA CAPTURE AND VISUALISATION INFRASTRUCTURE

An important aspect of our SDR demonstration system is to visualise and inspect signals in the transmit and receive paths of our FPGA architecture. As previously illustrated in Fig. 5, OPs are created at sections of our system that are to be inspected. Similarly, a CP was also included in the transmit path to control the output gain of the transmitted signal.

Each OP in our system uses a custom inspection architecture that moves data from the PL, to the off-chip DRAM. JupyterLab operating in the PS can then retrieve the data from DRAM for visualisation and analysis. We name the custom inspection architecture the Data Inspection Module, and reuse the architecture at each OP shown in Fig. 5.

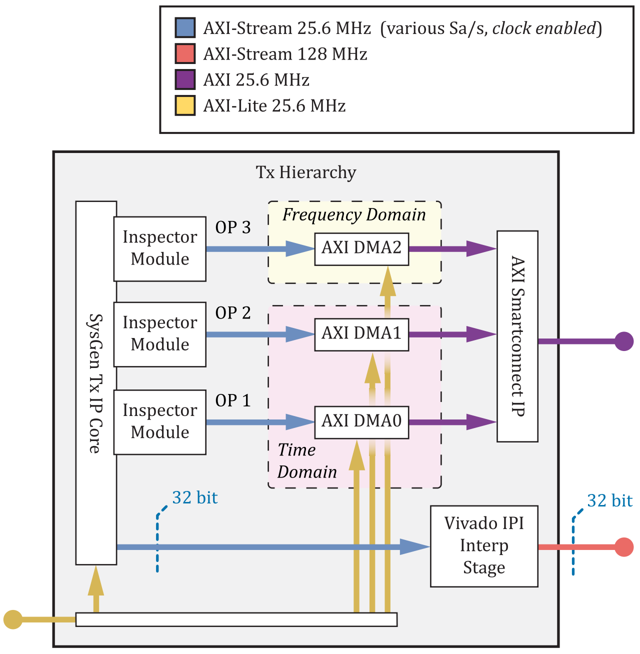


FIGURE 10. High-level Vivado IPI Tx hierarchy diagram.

The Data Inspection Module is an efficient FPGA data mover for visualising and inspecting PL data in JupyterLab. Fig. 11 contains the architecture of the Data Inspection Module, designed to move RF time domain samples or FFT frames between the PL and DRAM in the form of packets.

The Data Inspection Module design contains a synchronous FIFO that buffers valid RF data until it is ready to

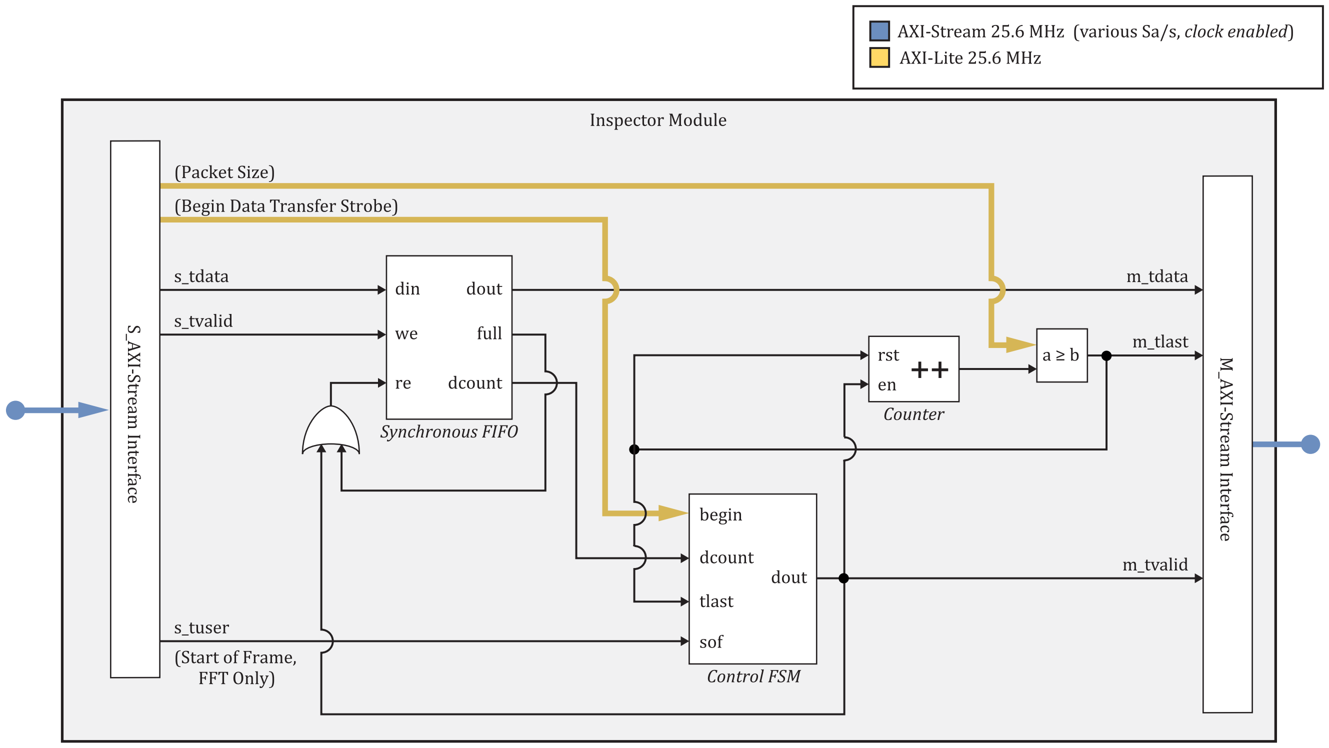


FIGURE 11. **Hardware architecture of the data inspector module.**

be transferred. The FIFO is consistently updated with new data samples and ejects old samples if they are not required. If there is a lack of data in the FIFO for data movement, the system will wait until there is enough data available, equal to a predetermined packet size.

A counter maintains the number of samples that have been passed to the DRAM by the data inspector module. When the packet size has been reached, a relational operator is used to produce the end of packet signal, `TLAST`. A Finite State Machine (FSM) ensures that data transfers to the DRAM are moderated and that all AXI-Stream sideband signals are correctly produced, including the `TVALID` signal that indicates the transfer of valid data.

The FSM also contains a Start of Frame (SoF) port, indicating when a frame of FFT data is to be moved to the DRAM. When the start of the FFT frame is pushed into the FIFO, the FSM is alerted using the SoF port. The FSM will then synchronise to the start of the FFT frame and begin transfer to the DRAM. This method ensures that the FFT frame is moved to the DRAM in the correct order.

The Master AXI-Stream interface, shown to the right of Fig. 11, is connected to an optimised Xilinx AXI DMA IP Core. This IP Core is used to efficiently move data between the PL and DRAM, via the AXI HP ports. The AXI DMA is an integral part of the Data Inspection Module architecture and is essential for achieving efficient data movement.

The Data Inspection Module also uses the AXI-Lite interface to receive the required packet size and data transfer requests from JupyterLab. As data is transferred using software issued commands from JupyterLab, no underlying knowledge of the hardware architecture is required to use this module.

IV. SOFTWARE DESIGN

This section looks at the methods used to develop the system software. We highlight the PYNQ platform, its use as an embedded front-end, and explain how we use it to control our custom IP and the RFSoC's hardened IPs.

A. THE PYNQ FRAMEWORK

PYNQ is an open-source project from Xilinx that targets its SoC family of devices, running the Ubuntu for Arm operating system, powered by Xilinx's own version of the Linux kernel. It challenges the traditional embedded C development process by offering a compelling alternative—an interactive prototyping environment facilitated by Python and a Jupyter web interface.

The foundation of the PYNQ project is formed by reuse of existing open-source projects, combining ideas from disparate fields and applying them to on-chip, embedded programming. Alongside this foundation, PYNQ supplies a Python library to expose the more embedded aspects of SoC development to Python developers. This includes fundamentals such as Memory-Mapped Input/Output (MMIO) for accessing the physical address space, drivers for DMA transfers, and plumbing for custom IP drivers. This Python library is enhanced by the run-time parsing of metadata accompanying the bitstream, namely the `.hwh` file generated by Vivado. This allows PYNQ to perform some run-time introspection on a user's hardware design, including the IPs, their configurations, and the address map. PYNQ uses this introspection to automatically bind drivers to any known IP cores and bind default drivers (with named register maps!) to any other IPs — providing a friendly prototyping environment for a hardware design with very little effort.

More information about the PYNQ project can be found at [3] and [43].

B. PYTHON DRIVERS FOR RFSoC HARDWARE

Two different types of drivers are needed for this demonstrator—drivers to control our QPSK IPs in the programmable logic, and drivers for the hardened RFSoC IP (including the data converters and clocking infrastructure).

Drivers for the QPSK IPs can be written entirely in Python, as explored in Section IV-B3. The PYNQ framework makes writing custom IP drivers a fairly rapid process, largely thanks to the MMIO and DMA libraries. An interactive approach can be taken to driver prototyping too—the user is free to peek and poke memory-mapped registers or perform DMA transfers in an ad hoc way at first. This can later be refined into a more formal driver class and automatically bound to its target IP core or hierarchy of IP cores by PYNQ [44].

Although we implement drivers for the QPSK IPs entirely in Python, a different approach was chosen for interacting with the RFSoC's hardened silicon IPs:

- On-board Clock Synthesizers: because the configuration is performed over I^2C (Inter-IC) and it is convenient to reuse Linux's support in C for this. The existing example code is also written in C.
- RF Data Converters: because there is no documented register map, we must reuse the vendor's baremetal C-code driver.

During this project, PYNQ drivers for the RFSoC's clock synthesizers and RF data converters were developed in collaboration with Xilinx. The source code is available at [45] and is included in the v2.5 PYNQ release, available at [46]. In both cases, an existing C driver is compiled as a shared library and is then used with a Python wrapper. The Python wrapper communicates with the shared library via the C Foreign Function Interface (CFFI) module [47].

1) ZCU111 CLOCK SYNTHESIZERS

The ZCU111 board has a set of clock synthesizers that drive all RF logic—one LMK04208 for a reference clock, then a bank of 3x LMX2594s. Having some control over the generation of these clocks is essential. We use an existing C example [48] with a very simple interface. With this example, custom clock frequencies can be realised by adding new register sets to a look-up table, and the user can reference these register sets to specify a target frequency. A very thin Python wrapper can then be written to use CFFI to call a single function in the C code to reconfigure all of the clock synthesizers. Our implementation can be found at [45].

The clock synthesis driver could be extended to allow for different generated clock frequencies for each of the domains (`RF{1,2,3}_CLK{A,B}`), or enable the generation of register values for new target frequencies on demand.

2) RF DATA CONVERTERS

There is a more formal existing C driver for the RF data converters that can be used as a foundation for the Python driver [49]. The data converters have a rich set of functionality, which can easily result in the API (Application Programming Interface) appearing a little intimidating. Instead of simply exposing a 1:1 mapping of the available C functions to Python equivalents, as is done with the clock synthesizers, some extra structure can be introduced to make the API more manageable.

There is a clear hierarchy to the data converters (it is composed of tiles, where each tile is composed of blocks, etc.) and this is also heavily suggested by the functions in the API. The flat list of $\approx$ 50 functions can then be split into an object-oriented hierarchy, also with a distinction between *attributes* (used just to get/set values), and *methods* (that perform some action). Fig. 12 shows a UML (Unified Modelling Language) interpretation of this structure.

The Python implementation of this wrapper can be made particularly concise and extensible by exploiting Python's support for reflection. This allows for modification of an object's structure at run-time. With these tools, namely `setattr()`, getter and setter functions for all of the attributes shown in Fig. 12 can be generated from a simple data structure—a list of tuples, each describing the C function name, C data type, and a read-only flag. This approach is demonstrated in Listing 1.

To appreciate some of the legibility improvements made by this idiomatic Python implementation, let's first consider a small example of using the existing C driver. Listing 2 shows an excerpt of C code required to update the mixer frequency of the first RF-ADC block.

Note in particular how all of the driver calls for the ADC block are made with reference to an IP-level instance pointer and a set of tile type, tile index, and block index parameters. With our Python version of the driver, this accessor plumbing is translated into an object-orientated hierarchy, as shown in Listing 3. Here we can store references to particular RF-ADC/RF-DAC blocks and access their properties as native Python structures (such as dictionaries for `MixerSettings` and other records).

Although we have introduced the benefits of an object-orientated approach to the driver API, there is still scope for making a more idiomatic Python interface for the data converters. This could, for example, also handle the streaming data transfer as part of a template hardware design (a "base overlay" in PYNQ nomenclature) for the ZCU111.

3) QPSK TX/RX DRIVERS

As a point of interest, we decided to try different approaches for building the Tx and Rx IPs. The Tx side was kept

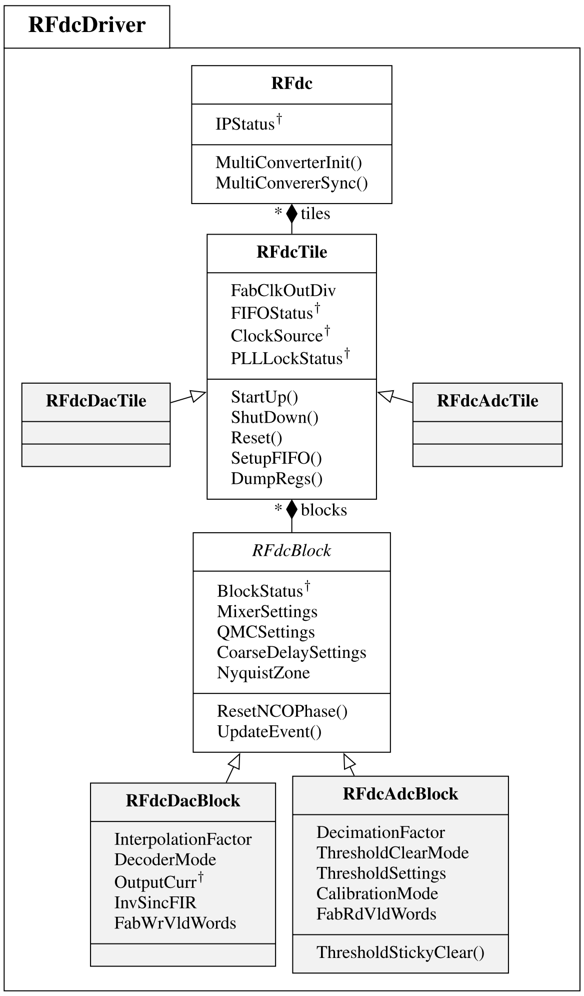


FIGURE 12. A class diagram for the Python data converter API (†read-only).

```
# We define all tile attributes in a list...
_tile_props = [("FabClkOutDiv",   "u16", False),
               ("FIFOStatus",     "u8",  True),
               ("ClockSource",    "u32", True),
               ("PLLLockStatus",  "u32", True)]
# ...

# Stub class for tile containing only methods
class RFdcTile:
    def StartUp():
    # ...

# Iterate through list of attributes descriptions
    and add these attributes to the Tile class at
    run-time
for (name, typename, readonly) in _tile_props:
    setattr(RFdcTile, name, _create_c_property(
        name, typename, readonly))
```

Listing 1. Example of data driven properties used in the Python wrapper.

as a monolithic System Generator design, while the Rx side was split into separate System Generator projects; then composed into a complete system design within Vivado IP Integrator. This choice also affected how the drivers are structured.

```
int Status;
u16 Tile = 0;
u16 Block = 0;
XRFdc_Mixer_Settings MixerSettings = {0};
XRFdc *RFdcInstPtr = ...

...

Status = XRFdc_GetMixerSettings(RFdcInstPtr,
    XRFDC_DAC_TILE, Tile, Block, &Mixersettings);
if (Status != XRFDC_SUCCESS) {
  return XRFDC_FAILURE;
}

MixerSettings.Freq = 1600;

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

Listing 2. Example of updating a mixer frequency with the RFDC C driver.

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

Listing 3. Example of updating a mixer frequency with the RFDC Python driver.

Because PYNQ can bind a Python class to a specific IP core (or a hierarchy of IP cores) the developer is naturally encouraged to write drivers per IP core. So, when presented with a monolithic IP core it is easy (and somewhat tempting) to let this translate to a monolithic Python class. As demonstrated in the Tx side driver (`qpsk_tx.py` in [37]), this temptation can lead towards some questionable practices. For example, there are, in retrospect, clear opportunities for further abstraction (code for each OP is essentially copied and pasted with minor changes). While this can be dismissed as poor developer practice, the psychology of it is interesting. In this case, the effort required to abstract over different base addresses, programatically accommodating all OPs, can be just enough to encourage this copy-and-paste "code smell" (a characteristic of software that hints at structural issues).

The converse is true for the Rx driver (`qpsk_rx.py` in [37]). Here, the full Rx signal path is split into separate IPs, each with a single OP. Now when approaching the PYNQ drivers, it becomes very natural to spot the commonality between them and implement one generic OP class. In this case (as shown in Listing 4), the whole set of Rx IPs can simply inherit from the generic OP class, overriding VLNV (Vendor, Library, Name, and Version) strings and adding any extra functionality (e.g. automatic reset timer in the timing synchronisation).

In this section we have observed that the granularity (and regularity) of IP cores used in a design can very naturally impact the structure of PYNQ drivers written for them, largely because of the 1:1 mapping of classes to IP cores. Of course, there is nothing precluding developers from making their own abstractions for large IP cores, but this effect was evident when following the pragmatic "path of least resistance" development.

```
class ObservationPoint(DefaultIP):
        # ...

# ...

class RxRRC(ObservationPoint):
    def __init__(self, description):
        super().__init__(description, 512)

    bindto = ['UoS:SysGen:axi_qpsk_rx_rrc:1.0']


class RxTSync(ObservationPoint):
    def __init__(self, description):
        super().__init__(description, 16)
        # Set loop filter reset counter to 1
            second (16kHz)
        self.sync_reset=16000

    bindto = ['UoS:SysGen:axi_qpsk_rx_tsync:1.0']
setattr(RxTSync, 'sync_reset',
    _create_mmio_property(20))
```

Listing 4. **Rx IP driver snippet with class hierarchy.**

C. INTERACTIVE VISUALISATIONS WITH JupyterLab

The other goal for our software design is to reuse existing technologies to quickly build an environment for prototyping RF applications. This should facilitate:

- Real-time introspection of the signal path, without any (expensive) external instrumentation
- Control of the system through GUI elements

Fortunately, this can be realised readily by combining some existing open-source efforts—namely JupyterLab as a web interface, ipywidgets for interactive graphical controls, and Plotly for interactive plotting. With these three projects, we can link graphical widgets to configurable aspects of our design, and provide some quick on-chip instrumentation for real-time visualisations in the frequency domain, time domain, and constellation plots. This goes a long way towards avoiding additional expensive instrumentation (oscilloscopes and spectrum analyzers) during the prototyping phase of SDR development. Also, the control aspects provide a good basis for an interactive dashboard in a deployed design. So, enter, JupyterLab.

1) JupyterLab

JupyterLab provides a development environment that is structured as a client/server system; running Python code on the RFSoC board and rendering an interface on the client PC's web browser. Every document produced with Jupyter (a "notebook") can intermingle markdown formatted documentation, blocks of Python code, and the output of any code (which can include anything from numerical or text-based results to interactive JavaScript plots). One of the original motivations for the Jupyter project was to give the scientific community a means of sharing methods and results in a reproducible fashion [50].

For this project, JupyterLab enables us to produce a demonstrator (developed with only a web browser), mixing diagrams and documentation of the design with the top-level code. It also lets us use the client's web browser to perform the rendering of our visualisations via JavaScript delivered by a JupyterLab extension (Plotly).

The last feature of JupyterLab that we exploit is its windowing system. Fig. 13 shows a typical setup where the user can open multiple windows within a single browser tab, arranging different combinations of the main notebook, ipywidgets controls, terminal sessions, and streaming plots—highlighted in boxes A, B, C, and D respectively.

2) BASIC INTERACTIVITY WITH IPYWIDGETS

Python libraries such as `ipywidgets` can be used to make GUI elements that control some of the more commonly used attributes. For example, we can readily create controls for the transmitter's centre frequency and gain (or technically digital attenuation in our case). An example of slider widgets can be seen in Fig. 13. Already with these widgets and the plots from section IV-C3, we could begin to replicate the web-based dashboards found on commercial radio platforms—for example the Amarisoft LTE stack which provides controls, QAM plots and channel response plots [51].

These widgets use callback functions that may occur asynchronously to the main thread. This deserves some extra thought during development, especially when working with hardware, to ensure that the design cannot fall into an invalid state. Our solution ensures atomic access to the RF data converter functions through use of a simple mutex (or "mutual exclusion") lock for synchronisation.

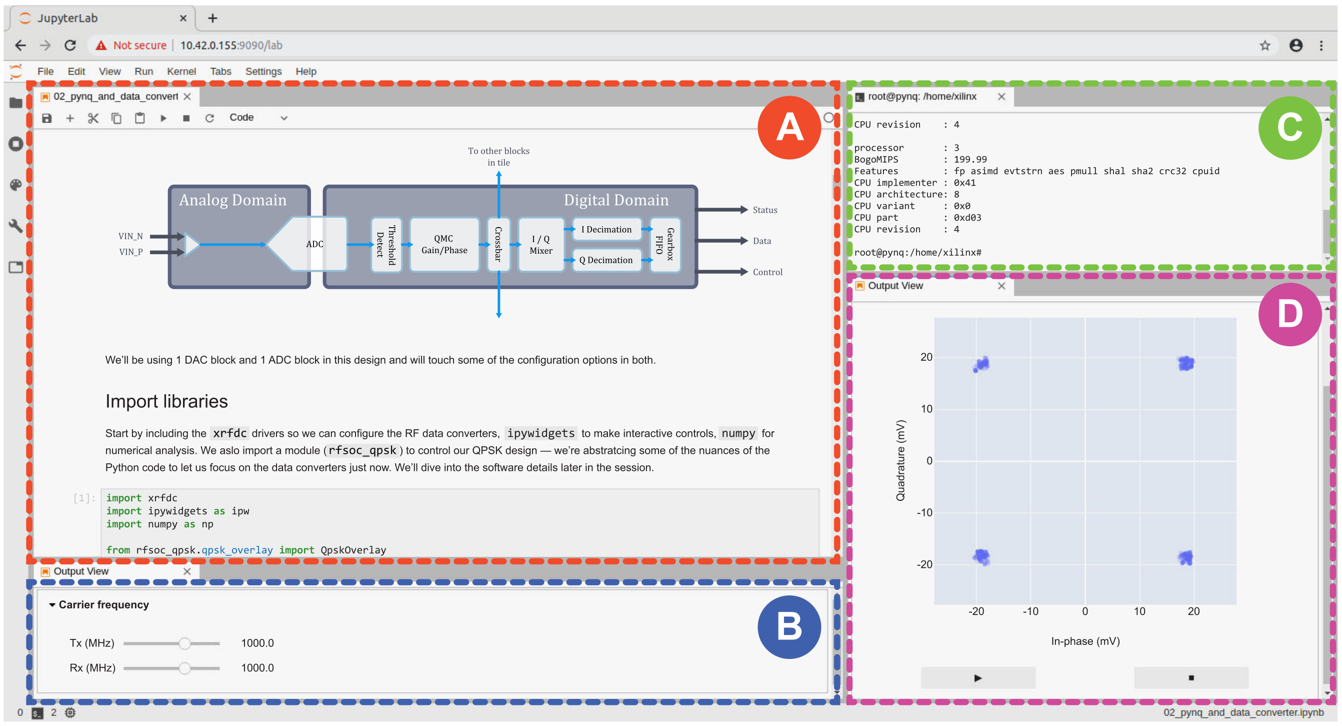
3) INTERACTIVE PLOTTING WITH PLOTLY

The plotting functionality is based on Plotly—a Javascript plotting library with good interactivity, some ability to update plot data dynamically, and JupyterLab integration [52]. We add a thin wrapper around Plotly to provide three main plot classes: a time domain I/Q plot, a constellation plot, and a frequency domain plot.

We use Python multithreading to request plot updates based on real-time data gathered from the PL, without stalling the main Python thread. This allows users to execute other cells in the notebook without disrupting any streaming plots. Even with a fairly naive polling timer approach, we can achieve $\approx$ 20 Frames Per Second (FPS) for time domain plots—as discussed in Section V-B.

It is important to understand the distinction between the server side (the ZCU111) and client side (the computer's web browser) processing here. All Python code is executed on the server side, but the core Plotly library is actually JavaScript (delivered via a JupyterLab extension) and is executed by the web browser of the client device. The general flow is:

- The server side will:
  - Perform a DMA transfer, bringing a frame of data into the PS DRAM
  - Optionally perform pre-processing such as resampling or a (software) FFT



We'll be using 1 DAC block and 1 ADC block in this design and will touch some of the configuration options in both.

Import libraries

Start by including the `xrfdc` drivers so we can configure the RF data converters, `ipywidgets` to make interactive controls, `numpy` for numerical analysis. We aslo import a module (`rfsoc_qpsk`) to control our QPSK design — we're abstratcing some of the nuances of the Python code to let us focus on the data converters just now. We'll dive into the software details later in the session.

```
[1]: import xrfdc
     import ipywidgets as ipw
     import numpy as np

     from rfsoc_qpsk.qpsk_overlay import QpskOverlay
```

```
CPU revision    : 4

processor       : 3
BogoMIPS        : 199.99
Features        : fp asimd evtstrn aes pmull sha1 sha2 crc32 cpuid
CPU implementer : 0x41
CPU architecture: 8
CPU variant     : 0x0
CPU part        : 0xd03
CPU revision    : 4

root@pynq:/home/xilinx#
```

FIGURE 13. Example of JupyterLab session running our QPSK design, providing real-time control and visualisation including A) the main notebook view, B) a window with ipywidget controls, C) a terminal session, and D) a window with a streaming constellation plot.

  – Generate a JSON (JavaScript Object Notation) description of a Plotly plot, using Plotly's python wrapper library
- JupyterLab sends this JSON description to the client browser
- The client side will parse the JSON plot description and render it using JavaScript

Because of this flow, plot performance will depend on the RFSoC, the client web browser performance, and the communication link between the two.

We provide infrastructure to lessen the demand on the link and the client device—although this is not used in the QPSK example because the data rates are not high enough to merit it. For any plots with a fixed plot width, the raw data set can be resampled on the server to reduce the number of points to, at most, one per pixel. This lessens the demands on plot rendering and the size of JSON data sent between server and client. The resampling algorithm can be simple downsampling, or decimation (including anti-alias filtering).

Resampling can also be triggered by redraw events, so as a user zooms into a smaller range of time, the full resolution of the raw data set becomes visible again. This helps produce large time domain plots that can remain responsive when viewed at any scale—looking at features spanning a number of seconds, or a number of microseconds.

V. DEMONSTRATION SYSTEM AND RESULTS

In this section we discuss the effectiveness of the demonstrator as a learning tool and highlight the benefits of using the PYNQ framework for embedded development. This is followed by a quantitative performance analysis of the software and hardware systems. All results were obtained from the ZCU111 REV-1.0 development board.

A. PERFORMANCE AS A DEMONSTRATOR

The purpose of the demonstrator is to guide the user through each step of the QPSK transceiver DSP path, while providing an interactive environment in order to both educate the user in software defined radio, and demonstrate the features of the RFSoC. Each step includes written explanations of the signal processing involved, while displaying live plots of the captured data to the user.

The use of PYNQ allows a level of documentation and user interaction that is difficult to obtain from other methods, facilitating interaction with driver-level functionality such as carrier frequency and transmitter gain. The user can manipulate these settings by either editing and running code on-the-fly or, with the use of ipywidgets, change settings with GUI sliders—all while receiving live visual feedback with the use of Python plotting libraries. The ability to use Python libraries alongside embedded platform code also helps reduce the code-base, avoiding the need to develop custom GUIs, which in turn reduces overall embedded development time and increases code readability. For example, Listing 5 shows a program for a single DMA transfer using the PYNQ framework in 11 lines of code, where random data is sent to a FIR filter and the output is displayed on a plot; an equivalent program in C would be many times the size.

The demonstrator has been in development since the latter part of 2018 and various stages of the design have been

```
# create a random array of 16-bit data
data = np.random.randint(2**15, size=100, dtype=np
    .int16)
# create DMA object
dma  = ol.axi_dma

# allocate contiguous memory for size of data
xlnk = Xlnk()
in_buffer  = xlnk.cma_array(shape=(len(data),)
    dtype=np.int16)
out_buffer = xlnk.cma_array(shape=(len(data),)
    dtype=np.int16)

# copy data to input buffer
np.copyto(in_buffer, data)

# initiate data transfer
dma.sendchannel.transfer(in_buffer)
# receive result
dma.recvchannel.transfer(out_buffer)
# wait for response
dma.sendchannel.wait()
dma.recvchannel.wait()

# plot results
plt.plot(out_buffer)
```

Lsiting 5. PYNQ code for DMA transfers to/from an IP, and plotting the result.

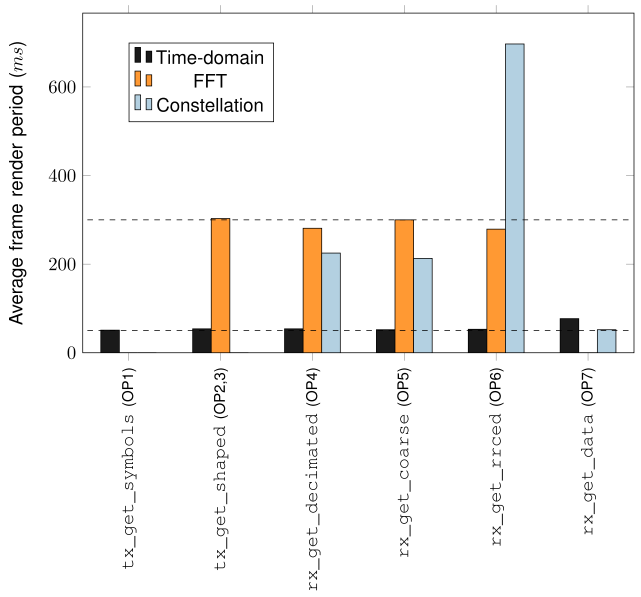


Figure 14. Average rendering periods for each plot type and Observation Point, with the dashed lines representing the target frame rates.

exhibited at conferences. Furthermore, the entire project has been released as open-source software, allowing anyone with access to a RFSoC ZCU111 development board to evaluate the design, or use it as a template for new projects [37].

An early version of the design, which consisted of the transmit part of the transceiver only, was exhibited at the 28th International Conference on Field Programmable Logic & Applications (FPL), Dublin, in August 2018. The transmitted signal was verified using a RTL-SDR QPSK receiver as described in [41]. An initial version of the full transceiver was then exhibited at the Xilinx Developer Forum (XDF), San Jose, in October of that year.

In 2019 the demonstrator was included in the "PYNQ on RFSoC" workshop at the XSight20: Xilinx Worldwide Sales Conference, Los Angeles, where over 100 participants over three days were trained on using the PYNQ framework on the RFSoC.

The majority of feedback from attendees was that the demonstrator provided a user-friendly interface that was able to explain the complexities of RFSoC system design in an easy-to-understand manner.

B. SOFTWARE PERFORMANCE

1) BROWSER PLOTTING PERFORMANCE

One of the most visually immediate metrics for software performance is the refresh rate of the real-time plots. Investigating this can give other engineers a grasp of how well the web-browser introspection can perform without needing a ZCU111 board. While any deficiencies in such a plot are easy for a human to observe, it is not a trivial metric to analyse due to the number of different technologies, languages, and hosts at play. For the raw rendering performance it is sufficient to consider only the JavaScript code running in the client PC's browser, if we can assume that the thin layer of Python-side code is not a performance bottleneck.

It is possible to analyse the client-side plot performance using the built-in Chrome DevTools. We record a browser performance profile over ≈ 10 seconds while streaming each plot in turn. It is important to note that the DevTools concept of a frame does not correspond to a full Plotly frame redraw, so we must obtain frame timing by filtering the event log by a function name we know to only happen once per redraw operation—such as `o.prepareFn`. Fig. 14 shows the average frame rendering periods, while Table 1 formulates this data alongside the number of data points and standard deviation for each plot.

The time domain and FFT plots are rendered nearly with their respective prescribed timer periods of 50 *ms* (20 FPS) and 300 *ms* (3.3 FPS), while the constellation plot performance has a clear dependence on the number of points per plot. Because time in the constellation plot is implied via opacity rather than an explicit axis (i.e. samples fade out as they get older), the resampling technique cannot sensibly help mitigate poor performance with larger data sets. We can also note that the anomalous "Rx data" time domain plot average is explained by the extra preprocessing step required to classify I/Q values as recovered bits, without adjusting frame sizes to compensate for this extra step.

Inspecting the averages and distributions of FFT frame periods from Table 1, we can notice that the timer for the software FFTs (in the Rx side), inherited from the hardware FFT (in the Tx side), is not pushing the design to any limit—indicated by the unusually small deviation from the timer period. Consequently, either a higher resolution FFT could be shown at the same rate, or similar FFT could be shown with a higher refresh rate.

TABLE 1. Rendering periods for each plot type and Observation Point.

	Time domain			FFT			Constellation		
Observation Point	Points	Avg (ms)	σ (ms)	Points	Avg (ms)	σ (ms)	Points	Avg (ms)	σ (ms)
									Tx Sources
`tx_get_symbols` (OP1)	128	51	25.7	—	—	—	—	—	—
`tx_get_shaped` (OP2&3)	1024	54	20.2	1024	303	95.7	—	—	—
									Rx Sources
`rx_get_decimated` (OP4)	1024	54	24.3	1024	281	72.4	1024	225	61.0
`rx_get_coarse` (OP5)	1024	52	22.5	1024	300	1.7	1024	213	31.5
`rx_get_rrced` (OP6)	4096	53	19.1	4096	279	77.0	4096	679	56.8
`rx_get_data` (OP7)	128	77	27.3	—	—	—	128	52	22.6

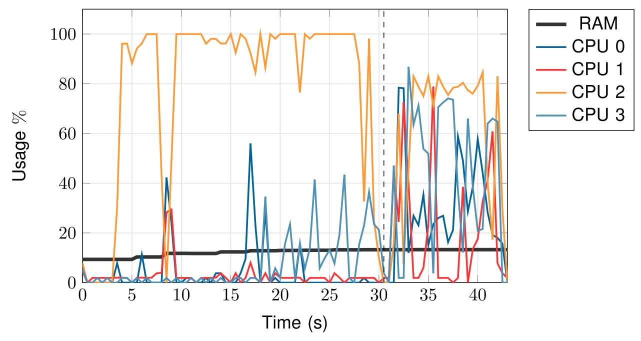


Figure 15. ZCU111 memory and CPU usages while loading the QPSK design, generating several one-off plots (before 30 *s*), and then streaming live-updates (after 30 *s*).

These performance results only affect the plotting visualisations running on the client laptop and not the DSP on the ZCU111. The DSP data path runs entirely independently on the RFSOC's PL and its performance cannot be impacted by the software running on either device.

2) ZCU111 CONTROL LATENCY WITH PYNQ

Another software performance metric is the latency of our controls. As measured by the IPython `timeit` command, it takes 4.23 *ms* to alter either of the RF-DC mixer frequencies. This is done via a CFFI call from Python to C, and executes a non-trivial C function.

A simple function to set Tx gain via a single MMIO write, entirely implemented in Python, takes only 11.5 μs according to `timeit`. Not only is this function written in Python, but the PYNQ implementation for MMIO is also written purely in Python—only relying on standard libraries such as `numpy`. This encouraging performance may come as a surprise to those who are new to using Python in embedded environments, given the stigma historically associated with interpreted languages, especially in the embedded domain.

Rapid response times like these are particularly encouraging when considering operation within DSA and shared spectrum environments, where standards, such as IEEE 802.22, require spectrum sharing radio nodes to reconfigure themselves and vacate an RF channel within 2 seconds of identifying a new licensed incumbent [53].

3) ZCU111 MEMORY AND CPU USAGE

Finally, let's consider the memory and CPU usage on the ZCU111 while running the demonstrator. Fig. 15 shows a trace of percentage use of RAM (4 GB in total) and each of the 4 Arm Cortex-A53 processors. At time 0 *s*, the system is idle and uses 9% of the available RAM to host Ubuntu, Jupyter, and any other background PYNQ processes. From here until time 30 *s*, the demonstrator design is initialised and we generate time-domain visualisations for OP1, OP2, and OP4. There are two insights for this period:

1) The memory usage is only slightly increased by our QPSK demonstrator, reaching a maximum of 13% at its plateau. This is possible because we only allocate one buffer for each OP and try to reuse it as much as possible, without relying on Python's garbage collection.
2) There is a single CPU that is nearly at 100% utilisation for most of this period. This is not indicative of the workload, but rather a consequence of Python's interpreter design and our use of PYNQ's blocking DMA calls (instead of their *asyncio* counterparts).

To expand on point 2), CPython's well known Global Interpreter Lock (GIL) ensures that even multi-threaded Python programs are not interpreted concurrently, largely because CPython's memory management is not thread-safe. This is why almost all activity is seen on only one processor — we have made no efforts to "break out of" the GIL. The 100% utilisation, however, is somewhat misleading. We use PYNQ's blocking calls to the DMA engine which will sit in a busy-waiting loop until the DMA transfer completes, ensuring very high CPU utilisation for any DMA-bound tasks. Therefore, Fig. 15 does not imply that the demonstrator has reached the computational ceiling of the A53 processors! PYNQ does expose a non-blocking interface to the DMA engine that makes use of the asyncio library, and this would greatly impact the CPU utilisation traces.

Beyond 30 *s*, the time-domain plot for OP4 is left to stream new data at 50 *ms* intervals. The same busy-wait DMA effects can be observed here. The CPU utilisation is not 100% however, because we request 33 *ms* of samples (128 samples at 4 *kHz*) for every 50 *ms* period, leaving some CPU time truly idle. Note that the time resolution of Fig. 15 is 0.5 *s*, so this busy/idle repetition averages out to ≈ 80% CPU usage.

In summary, this demonstrator shows that complex PYNQ designs can run on the ZCU111 with low memory usage. Processor usage is harder to analyse due to our use of busy-wait DMA transfers, although the results for plot streaming are encouraging. For future designs, we encourage designers to opt for the existing non-blocking DMA calls. Also, if the designer can escape CPython's GIL, then there are also three more processors that could be exploited for DSP tasks.

4) A NOTE ON BOTTLENECKS

With the current system, the main bottleneck has been identified as the client-side rendering of constellation plots — part of the only processing performed outside of the RFSoC's single-chip solution.

The constellation plots, as shown back in Fig. 14, start to perform substantially worse than our 20 FPS target when given large datasets. This effect is likely seen due to the use of opacity for each point, indicating a sample time. The effect could be mitigated for time-domain plots with our dynamic server-side resampling of signals, but this method is unsuitable for constellation plots. It also does not affect the FFT plots, as the number of samples relates to resolution in frequency and not directly to recording length/sample frequency.

However, our system is producing data at orders of magnitude below the full capability of the RFSoC. Consider the potential bottlenecks for a system with higher data rates.

As the data rate on the PL increases, it becomes more difficult to plot time-domain data on the PS due to the amount of data that must be either resampled or converted to JSON and sent to the client in full. Moreover, as the data rate reaches the data converter limits, memory storage becomes an issue where throughput can be measured in GB/s. In this scenario, any downsampling or decimation would be best suited to take place on the PL before being stored in memory.

For systems with any substantial DSP tasks running in software, CPython's GIL may also become a bottleneck, restricting all application execution to a single processor core. This could be fairly easily overcome by partitioning the system into different processes and using some form of inter-process communication, breaking out of the GIL and exploiting all available cores.

C. FPGA UTILISATION

The FPGA hardware design includes a full QPSK transceiver, including synchronisation that only utilises <9% of CLBs (configurable logic blocks), <7% BRAMs (block RAMs), and ≈ 5% of DSPs. Although this low utilisation is due to the large size of the FPGA available on the XCZU28DR-2FFVG1517E, this leaves a large amount of resources available for multiple instances of the transceiver to be implemented if a multi-channel system is desired. Table 2 displays an abridged Vivado utilisation report of the Tx and Rx hierarchies, the data converter, as well as an overall total of the resources used.

TABLE 2. FPGA resource utilisation for the Tx and Rx hierarchies, and RF-DCs.

	CLB (LUTs)	BRAM	DSP	MMCM	PLL
Tx	14602 (3.43%)	43.5 (4.03%)	134 (3.14%)	0 (0%)	1 (6.25%)
Rx	18433 (4.33%)	31.5 2.92%)	79 (1.85%)	1 (12.5%)	0 (0%)
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%)

The most notable result here is the MMCM where 12.5% of resources are used. This is due to the use of a clocking wizard (requiring 12.5% MMCM resources per instance) for the Rx logic, which is required to upsample the respective AXI-Stream clock from 64 MHz to 128 MHz, as well as downsampling to 25.6 MHz. The use of a MMCM resource was not required for the Tx logic as the DAC produced exactly the clock frequency required to support its AXI-Stream channel. This allowed the clocking wizard to be controlled by a PLL instead (using only 6.25% resources per instance) to create the 64 MHz and 25.6 MHz clocks needed for the Tx logic. This mixture of MMCM and PLLs allows the maximum Tx/Rx channel pairs (8 each, 16 total on the ZCU111) to fit within the available PL resources if a multi-channel system were to be implemented.

An overall total of 2.24% of CLBs and 4.05% of BRAMs are used by DMAs to accommodate the Observation Points provided for demonstration purposes. If the transceiver were to be used in a deployment environment, these could be removed—further increasing available resources on the FPGA.

Using the default Vivado implementation strategy, all timing constraints were met.

VI. CONCLUSION

This paper has introduced an SDR demonstration system based on the Xilinx RFSoC platform, and the Python-based PYNQ software framework. The system is the first to demonstrate successful combination of these two technologies for SDR, and includes support for software-based control of radio functionality, as well as the capture and visualisation of 'live' data from the chip.

The paper details the composition of the system, including the development of the hardware and software portions of the design. In particular, we highlight the additional infrastructure that is incorporated within the hardware system to support data capture, and the control and visualisation software elements implemented using PYNQ. The associated set of design files have been shared with the community on an open-source basis, hence there is an opportunity to directly build upon our work.

The developed SDR system was evaluated in several respects. We exposed early versions at events, and collected informal feedback that led to subsequent improvement of the interactive aspects of the system. Design reuse opportunities were considered, particularly in terms of leveraging existing open source software, as discussed extensively throughout the paper. Software/hardware co-design was investigated by implementing the hardware IPs for the transmit and receive sides with two different levels of granularity, and assessing the design from methodology and documentation perspectives. It was found that smaller IP blocks mapped more naturally to object-oriented driver software, leading to elegant, intuitive code and enhanced design reuse opportunities.

Further, we quantitatively investigated the responsiveness of the system, in terms of the latencies for applying software control parameters to hardware, and for data capture, and assessed achievable plot refresh rates. It was found that suitable rates could be achieved for all plot types. We also provide an analysis of memory and processor utilisation on the ZCU111 board, demonstrating that there is little pressure on the PS memory. There are, however, some interesting (but, in our case, superficial) patterns in the processor usage that are attributed to our own software design choices. We also suggest how to avoid such usage patterns in future designs. Less than 10% of the resources of a XCZU28DR RFSoC device were consumed by the entire system, of which Block RAM (two thirds of that used) was the main cost for PYNQ-related infrastructure. There is therefore considerable potential to incorporate enhanced radio functionality, further Observation Points, or to extend the design to multiple channels.

ACKNOWLEDGMENTS

The authors would like to thank Patrick Lysaght, Cathal McCabe, Peter Ogden, Graham Schelle (all of Xilinx), and Enno Leubbers (formerly of Xilinx) for their support of this work. They also extend their appreciation to two of our research colleagues at Strathclyde, Andrew Maclellan, and Marius Šiaučiulis, who have provided useful feedback during the production of this paper. They also thank Xilinx for the use of a ZCU111 RFSoC development board, and software donations.

REFERENCES

[1] *Zynq UltraScale + RFSoC Data Sheet: Overview DS889*, Xilinx, San Jose, CA, USA, Jul. 2018.

[2] *Zynq UltraScale + RFSoC RF Data Converter 2.0 PG269*, Xilinx, San Jose, CA, USA, Apr. 2018.

[3] Xilinx. *PYNQ: Homepage*. Accessed: Jun. 17, 2020. [Online]. Available: https://pynq.io

[4] Xilinx. *Xilinx Discusses Fronthaul Challenges for the 5G Optical Network*. Accessed: Jun. 17, 2020. [Online]. Available: https://www.youtube.com/watch?v=UPucJcDAfSk

[5] Medium. *Cloud RAN and eCPRI Fronthaul in 5G Networks*. Accessed: Jun. 17, 2020. [Online]. Available: https://medium.com/5g-nr/cloud-ran-and-ecpri-fronthaul-in-5g-networks-a1f63d13df67

[6] Xilinx. *RF Solutions With Zynq UltraScale + RFSoC*. Accessed: Jun. 17, 2020. [Online]. Available: https://www.xilinx.com/publications/events/developer-forum/2018-frankfurt/rf-solutions-with-zynq-ultraScale-plus-rfsoc.pdf

[7] Ettus Research Inc. *Products Webpage*. Accessed: Jun. 17, 2020. [Online]. Available: https://www.ettus.com/products

[8] Nutaq. *PicoSDR Series for Wireless Multi-Standard Prototyping*. Accessed: Jun. 17, 2020. [Online]. Available: https://www.nutaq.com/products/picosdr

[9] Lime Microsystems Ltd. *SDR and Companion Board Products*. Accessed: Jun. 17, 2020. [Online]. Available: https://limemicro.com/products/boards

[10] Analog Devices Inc. *ADI AD9361 System on Module (SOM) SDR*. Accessed: Jun. 17, 2020. [Online]. Available: https://wiki.analog.com/resources/eval/user-guides/adrv936x_rfsom

[11] Analog Devices Inc. *AD-FMCOMMS3-EBZ User Guide*. Accessed: Jun. 17, 2020. [Online]. Available: https://wiki.analog.com/resources/eval/user-guides/ad-fmcomms3-ebz

[12] Avnet Inc. *Zynq-7000 AP SoC / AD9361 Software-Defined Radio Evaluation Kit*. Accessed: Jun. 17, 2020. [Online]. Available: http://zedboard.org/sites/default/files/product_briefs/PB-AES-ZSDR2-ADI-G-V1.pdf

[13] Xilinx. *Avnet Zynq-7000 SoC / AD9361 Software-Defined Radio Systems Development Kit*. Accessed: Jun. 17, 2020. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/1-45sl7b.html#overview

[14] Mathworks Inc. *Design and Prototype SDR Systems With MATLAB and Simulink*. Accessed: Jun. 17, 2020. [Online]. Available: https://uk.mathworks.com/discovery/sdr.html

[15] *GNURadio Homepage*. Accessed: Jun. 17, 2020. [Online]. Available: https://www.gnuradio.org/

[16] J. Blum. *Pothos SDR Dev Environment Wiki*. Accessed: Jun. 17, 2020. [Online]. Available: https://github.com/pothosware/PothosSDR/wiki

[17] National Instruments. *Software Defined Radio*. Accessed: Jun. 17, 2020. [Online]. Available: https://www.ni.com/en-gb/innovations/wireless/software-defined-radio.html

[18] Mathworks Inc. *QPSK Receiver Using Analog Devices AD9361/AD9364*. Accessed: Jun. 17, 2020. [Online]. Available: https://uk.mathworks.com/help/supportpkg/xilinxzynqbasedradio/examples/qpsk-receiver-using-analog-devices-ad9361-ad9364.html

[19] Mathworks Inc. *QPSK Transmitter Using Analog Devices AD9361/AD9364*. Accessed: Jun. 17, 2020. [Online]. Available: https://uk.mathworks.com/help/supportpkg/xilinxzynqbasedradio/examples/qpsk-transmitter-using-analog-devices-ad9361-ad9364.html

[20] ORCA. *ORCA Functionalities*. Accessed: Jun. 17, 2020. [Online]. Available: https://www.orca-project.eu/orca-functionalities/

[21] M. Danneberg, R. Bomfin, S. Ehsanfar, A. Nimr, Z. Lin, M. Chafii, and G. Fettweis, "Online wireless lab testbed," in *Proc. IEEE Wireless Commun. Netw. Conf. Workshop (WCNCW)*, Marrakech, Morocco, Apr. 2019, pp. 1–5.

[22] J. Mitola, "The software radio architecture," *IEEE Commun. Mag.*, vol. 33, no. 5, pp. 26–38, May 1995.

[23] SDR Forum. (Nov. 2007). *SDRF Cognitive Radio Definitions*. Accessed: Jun. 17, 2020. [Online]. Available: http://www.sdrforum.org/pages/documentLibrary/documents/SDRF-06-R-0011-V1_0_0.pdf

[24] B. Drozdenko, M. Zimmermann, T. Dao, K. Chowdhury, and M. Leeser, "Modeling considerations for the hardware-software co-design of flexible modern wireless transceivers," in *Proc. 26th Int. Conf. Field Program. Log. Appl. (FPL)*, Lausanne, Switzerland, Aug. 2016, pp. 1–4.

[25] S. Hosny, E. Elnader, M. Gamal, A. Hussien, A. H. Khalil, and H. Mostafa, "A software defined radio transceiver based on dynamic partial reconfiguration," in *Proc. New Gener. CAS (NGCAS)*, Valletta, Malta, Nov. 2018, pp. 158–161.

[26] J. Ralston and C. Hargrave, "Software defined radar: An open source platform for prototype GPR development," in *Proc. 14th Int. Conf. Ground Penetrating Radar (GPR)*, Shanghai, China, Jun. 2012, pp. 172–177.

[27] B. Wang, J. Saniie, S. Bakhtiari, and A. Heifetz, "Software defined ultrasonic system for communication through solid structures," in *Proc. IEEE Int. Conf. Electro/Inf. Technol. (EIT)*, Rochester, MI, USA, May 2018, pp. 267–270.

[28] S. Gokceli, T. Levanen, J. Yli-Kaakinen, M. Turunen, M. Allen, T. Riihonen, A. Palin, M. Renfors, and M. Valkama, "Software-defined radio prototype for fast-convolution-based filtered OFDM in 5G NR," in *Proc. Eur. Conf. Netw. Commun. (EuCNC)*, Valencia, Spain, Jun. 2019, pp. 235–240.

[29] J. Budroweit and A. Koelpin, "Design challenges of a highly integrated SDR platform for multiband spacecraft applications in radiation environments," in *Proc. IEEE Top. Workshop Internet Space (TWIOS)*, Anaheim, CA, USA, Jan. 2018, pp. 9–12.

[30] 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 Commun. Mag.*, vol. 53, no. 9, pp. 64–71, Sep. 2015.
[31] S. Tridgell, D. Boland, P. H. W. Leong, and S. Siddhartha, "Real-time automatic modulation classification," in *Proc. Int. Conf. Field-Program. Technol. (ICFPT)*, Tianjin, China, Dec. 2019, pp. 299–302.
[32] M. Fosberry and M. Livadaru, "Digital synthetic receive beamforming with the Xilinx ZC1275 evaluation board," in *Proc. IEEE Int. Symp. Phased Array Syst. Technol. (PAST)*, Waltham, MA, USA, Oct. 2019, pp. 1–2.
[33] M. Livadaru, M. Fosberry, N. Campbell, K. Bassett, S. Speck, J. Fung, P. Schibly, S. Blackwell, A. Kraemer, A. Redenbaugh, B. Mcmahon, and R. Lapierre, "On the integration of wideband adaptable hardware technologies to enable RF machine learning," in *Proc. IEEE Int. Symp. Phased Array Syst. Technol. (PAST)*, Waltham, MA, USA, Oct. 2019, pp. 1–2.
[34] Red Pitaya. *Red Pitaya's SDR Transceiver*. Accessed: Jun. 17, 2020. [Online]. Available: https://www.redpitaya.com/n30/red-pitayas-sdr-transceiver
[35] K. Haeublein, W. Brueckner, S. Vaas, S. Rachuj, M. Reichenbach, and D. Fey, "Utilizing PYNQ for accelerating image processing functions in ADAS applications," in *Proc. 32nd Int. Conf. Archit. Comput. Syst. (ARCS Workshop)*, Copenhagen, Denmark, May 2019, pp. 1–8.
[36] E. Wang, J. J. Davis, and P. Y. K. Cheung, "A PYNQ-based framework for rapid CNN prototyping," in *Proc. IEEE 26th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Boulder, CO, USA, May 2018, p. 223.
[37] *University of Strathclyde—Software Defined Radio Research Laboratory*. Accessed: Jun. 17, 2020. [Online]. Available: https://github.com/strath-sdr/rfsoc_qpsk
[38] Xilinx. *ZCU111—Product Page*. Accessed: Jun. 17, 2020. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/zcu111.html
[39] *ZCU111 Evaluation Board User Guide UG1271*, Xilinx, San Jose, CA, USA, Oct. 2018.
[40] M. Rice, *Digital Communications: A Discrete-Time Approach*. Upper Saddle River, NJ, USA: Prentice-Hall, 2009.
[41] D. S. W. Atkinson, K. W. Barlee, and R. W. Stewart, *Software Defined Radio Using MATLAB & Simulink and the RTL-SDR*. Glasgow, U.K.: Strathclyde Academic Media, 2015.
[42] *Zynq UltraScale + MPSoC Data Sheet: DC and AC Switching Characteristics DS925*, Xilinx, San Jose, CA, USA, Jul. 2019.
[43] C. Ramsay, L. H. Crockett, and N. David, *Exploring Zynq MPSoC: With PYNQ and Machine Learning Applications*. Glasgow, U.K.: Strathclyde Academic Media, 2019.
[44] Xilinx. *Python Overlay API*. Accessed: Jun. 17, 2020. [Online]. Available: https://pynq.readthedocs.io/en/v2.3/index.html
[45] Xilinx. *ZCU111-PYNQ*. Accessed: Jun. 17, 2020. [Online]. Available: https://github.com/Xilinx/ZCU111-PYNQ
[46] Xilinx. *PYNQ: Development Boards*. Accessed: Jun. 17, 2020. [Online]. Available: https://pynq.io/board.html
[47] A. Rigo and M. Fijalkowski. *CFFI Reference*. Accessed: Jun. 17, 2020. [Online]. Available: https://cffi.readthedocs.io/en/latest/ref.html
[48] Xilinx. *Xilinx Embedded Software Development*. Accessed: Jun. 17, 2020. [Online]. Available: https://github.com/Xilinx/embeddedsw/blob/23eb39df101391b896adf20fa9d6c5aee27b0adc/XilinxProcessorIPLib/drivers/rfdc/examples/xrfdc_clk.c
[49] Xilinx. *Xilinx Embedded Software Development: RFDC*. Accessed: Jun. 17, 2020. [Online]. Available: https://github.com/Xilinx/embeddedsw/tree/23eb39df101391b896adf20fa9d6c5aee27b0adc/XilinxProcessorIPLib/drivers/rfdc
[50] F. Perez and B. E. Granger. *An Open Source Framework for Interactive, Collaborative and Reproducible Scientific Computing and Education*. Accessed: Jun. 17, 2020. [Online]. Available: https://ipython.org/_static/sloangrant/sloan-grant.html
[51] *Amarisoft eNodeB: Full Specification*. Accessed: Jun. 17, 2020. [Online]. Available: https://www.amarisoft.com/technology/enodeb/
[52] Plotly. *Plotly Python Open Source Graphing Library*. Accessed: Jun. 17, 2020. [Online]. Available: https://plot.ly/python/
[53] *ISO/IEC/IEEE, Information Technology—Telecommunications and Information Exchange Between Systems—Local and Metropolitan Area Networks—Specific Requirements—Part 22: Cognitive Wireless RAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications: Policies and Procedures for Operation in the TV Bands*, IEEE Standard 802-22, May 2015.

JOSH GOLDSMITH (Graduate Student Member, IEEE) received the B.Eng. degree (Hons.) from the University of Strathclyde, in 2017, where he is currently pursuing the Ph.D. degree with the Software Defined Radio Research Laboratory. He completed a six month internship with Xilinx, Longmont, developing hardware systems and training material for the RFSoC within the PYNQ Team, in 2019. He is a contributing author of *Exploring Zynq MPSoC* Book [43]. His research interests include run-time reconfigurable hardware, specifically for FPGA radio applications, signal processing, and embedded systems.

CRAIG RAMSAY received the B.Eng. degree (Hons.) from the University of Strathclyde, in 2017, where he is currently pursuing the Ph.D. degree with the Software Defined Radio Laboratory. He is a co-author of the Book *Exploring Zynq MPSoC*. His current research interests include functional programming techniques for hardware description, including formal verification and using dependent types for DSP applications.

DAVID NORTHCOTE (Member, IEEE) received the B.Eng. degree (Hons.) in electronic and electrical engineering, in 2015. He is currently a Researcher with the Department of Electronic and Electrical Engineering (EEE), University of Strathclyde. His Ph.D. research was on the efficient implementation of the Hough Transform for embedded vision systems using Zynq MPSoC. He is the coauthor of the technical book *Exploring Zynq MPSoC* [43]. His research interests include efficient implementation of wireless communication and computer vision applications on Zynq.

KENNETH W. BARLEE (Member, IEEE) received the B.Eng. degree (Hons.), in 2014.

He is currently a Researcher with the Software Defined Radio Laboratory, University of Strathclyde. During his Ph.D. degree, he developed filter bank multicarrier (FBMC)-based cognitive SDR transceivers for use in secondary user dynamic spectrum applications. He is one of the authors of the SDR textbook *Software Defined Radio using MATLAB & Simulink and the RTL-SDR* [41]. From 2018 to 2019, he was a part of the team that designed, developed, and with the help of partners, deployed a full shared spectrum 4G/5G mobile network across 360 km^2 of the Scottish Orkney Islands, as part of the 5G RuralFirst Project. He carried out the programming of software defined (radio) basestations, interoperability testing with third party packet cores, RF planning and network design, and developed bespoke security and management solutions for basestation and user equipment.

LOUISE H. CROCKETT received the master's and Ph.D. degrees in electronic and electrical engineering from the University of Strathclyde. She is currently a Senior Teaching Fellow with the University of Strathclyde. Her teaching focuses on digital systems design using hardware description language (HDL), simulink block-based design, and FPGA/SoC technology, and builds practical skills to equip graduates for roles in industry. Her research interests include hardware implementation of digital signal processing (DSP) systems, in particular for communications and SDR. She is the principal author of *The Zynq Book*.

ROBERT W. STEWART received the bachelor's and Ph.D. degrees from the University of Strathclyde. From 2014 to 2017, he was the Head of the Department of Electronic and Electrical Engineering, University of Strathclyde. He is currently a Professor of signal processing and manages a research group working on DSP, FPGAs, whitespace radio, and low-cost SDR implementation. He leads the Strathclyde cohort of the Scotland 5G Centre. He is building on the work of 5G RuralFirst, the team has a focus on spectrum sharing, SDR eNodeB deployments, and developing innovative solutions to combat the connectivity issues faced in rural areas of the U.K.