Hindawi
International Journal of Reconfigurable Computing
Volume 2017, Article ID 3925961, 28 pages
https://doi.org/10.1155/2017/3925961

Research Article

Software-Defined Radio FPGA Cores: Building towards a Domain-Specific Language

Lekhobola Tsoeunyane,[1] Simon Winberg,[1] and Michael Inggs[2]

[1]*Department of Electrical Engineering, University of Cape Town, Software Defined Radio Group, Rondebosch, Cape Town 7701, South Africa*
[2]*Department of Electrical Engineering, University of Cape Town, Radar Remote Sensing Group, Rondebosch, Cape Town 7701, South Africa*

Correspondence should be addressed to Lekhobola Tsoeunyane; lekhobola@gmail.com

Received 9 December 2016; Revised 7 April 2017; Accepted 16 April 2017; Published 17 July 2017

Academic Editor: Michael Hübner

This paper reports on the design and implementation of an open-source library of parameterizable and reusable Hardware Description Language (HDL) Intellectual Property (IP) cores designed for the development of Software-Defined Radio (SDR) applications that are deployed on FPGA-based reconfigurable computing platforms. The library comprises a set of cores that were chosen, together with their parameters and interfacing schemas, based on recommendations from industry and academic SDR experts. The operation of the SDR cores is first validated and then benchmarked against two other cores libraries of a similar type to show that our cores do not take much more logic elements than existing cores and that they support a comparable maximum clock speed. Finally, we propose our design for a Domain-Specific Language (DSL) and supporting tool-flow, which we are in the process of building using our SDR library and the Delite DSL framework. We intend to take this DSL and supporting framework further to provide a rapid prototyping system for SDR application development to programmers not experienced in HDL coding. We conclude with a summary of the main characteristics of our SDR library and reflect on how our DSL tool-flow could assist other developers working in SDR field.

1. Introduction

Software-Defined Radio (SDR) approaches for rapid prototyping of radio systems using reconfigurable hardware platforms offer significant advantages over traditional analog and hardware-centered methods. In particular, time and cost savings can be achieved by reusing tested design artefacts. For example, a reconfigurable computer coupled to a commercial off-the-shelf (COTS) Radio Frequency (RF) daughterboard can reduce development time and lower costs in comparison to a custom-built PCB approach. A broad variety of SDR prototyping platforms is available, such as the USRP and Microsoft Sora [1], and rapid prototyping tools such as National Instruments LabView and GnuRadio [2]. The choice of SDR platform and components used to develop a complex SDR system is typically based on a variety of many interrelated selection decisions [3]. Important selection decisions during development are influenced by the developers' familiarity with processor and platform architectures, coding languages, design approaches, support for legacy systems, and familiar design tools, among other factors. Development tool-chains and programming languages are highly influential in terms of developer productivity. Similarly, familiarity with the tools can impact the quality and reusability of the designs [3]. In this paper we consider a standard VHSIC Hardware Description Language (VHDL) approach to SDR prototyping, for which we develop a reusable library of Intellectual Property (IP) cores for use in Software-Defined Radio (SDR) application prototyping. We test the system using a legacy FPGA-based platform and recent add-on COTS daughterboard as a case study around which we build and test this library.

The aim of our investigation is to establish an effective selection of IP cores for baseline SDR applications, which can improve developer productivity using cores that can be further built upon to realize more specialized application needs. The Reconfigurable Hardware Interface for computatioN and radiO (RHINO) platform [4] is used as a case study for testing the library. In our effort to improve SDR productivity and reduce design complexity, we also propose a tool-flow that uses a Domain-Specific Language (DSL) as its entry-point to describe algorithms and automate the generation of HDL code. It exploits the parameters of SDR cores in order to integrate these existing library cores in the compilation flow hence enabling rapid prototyping of FPGA-based SDR at high-level of design abstraction.

The SDR IP core library comprises Digital Signal Processing (DSP) cores and input/output (I/O) interface cores. The library, which we are referring to as "SDR IP cores" library, is available for free under the General Public License (GPL) on https://github.com/lekhobola/Rhino-Processing-Blocks. It is designed around use by both the novice and experienced low-level HDL developers, providing novice users with experience of using IP cores that support open bus interfaces in order to exploit System-On-Chip (SoC) design without commercial, parameter, and bus compatibility limitations. The provided modules will be of particular benefit to the novice developers in providing ready-made examples of processing blocks, as well as parameterization settings for the interfacing cores and associated RF receiver side configuration settings. DSP cores can be used in any FPGA platforms whereas porting the I/O interface cores requires replacing Spartan-6 clock management and interconnecting libraries with new target-specific platform libraries.

The DSP cores are realized with fundamental DSP algorithms: Finite Impulse Response (FIR), Infinite Impulse Response (IIR), Fast Fourier Transform/Inverse Fast Fourier Transform (FFT/IFFT), and Digital Down Converter (DDC) algorithms. These DSP cores are accompanied by a description of how they can be integrated into a common Open Standard Interconnection Bus, namely, Wishbone. Furthermore, the I/O interface cores realize the interface control logic for Gigabit Ethernet (Gbe) and 4DSP FMC150 Analog-to-Digital Converter/Digital-to-Analog Converter (ADC/DAC) daughter board, both being part of RHINO. The Gbe interface core uses UDP protocol to enable high-speed data transfer between RHINO and external devices while FMC150 ADC/DAC provides an air interface for RHINO at high sampling rates. A Frequency Modulation (FM) receiver is then built from the IP cores to demonstrate the importance and reusability of the library of IP cores in the real world context of SDR.

The remainder of this paper is organized as follows. In Section 2, we provide a background to the SDR, FPGAs, and reconfigurable computing (RC). Next, the existing IP libraries are reviewed in Section 3. This is followed by the methodology used in developing and evaluating the library of SDR IP cores in Section 4. We continue with the design and implementation of a library of SDR IP cores in Section 5. In Section 6 we perform validation testing for SDR IP cores and results are compared to ideal Matlab simulation results. As a case study for SDR IP cores, we continue the testing by building an FM receiver in Section 7. This is followed by benchmarking all the SDR cores including an FM receiver in Section 8. Finally, we propose a tool-flow that enables high-level design using SDR IP cores in Section 9 while Section 10 covers conclusion.

2. Background

The ever increasing popularity and evolution of wireless communication technologies and standards are changing the manner in which wireless services and applications are used [6]. The demand and usage of these services by users are growing rapidly and are constantly pushing designs to their limits. Wireless devices are becoming more common and users are demanding the convergence of multiple services and technologies [7] in a single device. These lead to potential challenges in areas of equipment design, wireless service provision, security, and regulation [8].

2.1. SDR Systems. Configurable technologies are a solution to today's increasing user needs for wireless services and applications. These types of technologies are upgradable, reconfigurable, and adaptable to changes in technology standards and need [9]. One such technology that offers all these features is SDR. SDR is defined as radio in which hardware components or physical layer functions of a wireless communications system are all implemented in software [10].

SDR prototyping has opened doors to many possibilities in the field of radio communications. Owing to its rapid growth in recent years, it has gained popularity and has also found wide adoption in the analysis and implementation of many wireless communications systems. Traditional systems are now replaced by SDR systems because of their high reconfigurability and increased capabilities which suit modern wireless communications technology [9, 10].

SDR relies on a general purpose hardware that is easy to program and configure in software to enable a radio platform to adapt to multiple forms of operation such as multiband, multistandard, multimode, multiservice, and multicarrier [6, 10]. A typical SDR transceiver is depicted in Figure 1. The analog RF front-end converts RF signals to Intermediate Frequency (IF) signals in the receiver chain while the transmitter converts IF signals to RF signals. This is also where signal preconditioning and postconditioning using analog functions such as amplification and heterodyne mixing prior to ADC and after the DAC take place [6, 8]. The DSP performance largely depends on the digital computing hardware device used. Furthermore, improved and higher sampling ADCs and DACs are pushing the tasks traditionally performed in analog closer towards the antenna, hence allowing them to be processed digitally using processors or reconfigurable devices [11]. However, a drawback is that the ADCs and DACs are usually costly, and achieving high sampling rates (over millions of samples per second) remains a limitation in SDR [6]; this is a motivating factor for reusable SDR platforms for prototyping to share the cost of the same platform across multiple projects.

2.2. Overview of FPGAs and Reconfigurable Computing. The emergence of FPGA technology more than two decades ago

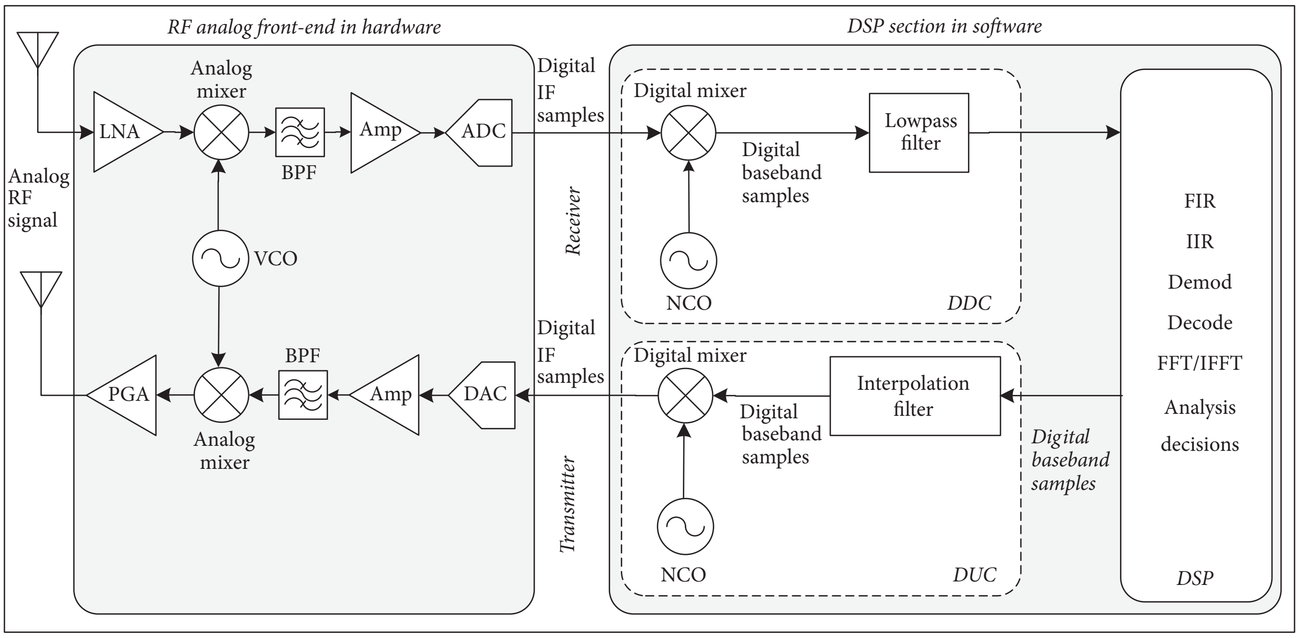


Figure 1: Radio transceiver architecture.

has revolutionized the field of SDR. FPGAs are made of highly reconfigurable and multiple logic blocks and cells together with switch matrix to route signals between them [12]. Their flexibility and speed have made them popular and are preferred to lay a general purpose hardware platform for SDR. The reconfigurable and parallel characteristics of FPGAs enable computationally intensive and complex tasks to be processed in real time with better performance and flexibility. These features have seen them gaining popularity over traditional general purpose processors (GPPs) and DSP processors [10]. For these reasons, they are used in RC as their structure can be reconfigured during start-up or runtime to perform advanced computations [13].

2.3. Design for Reuse. FPGAs have led to the concept of design for reuse which is a driving factor in enhancing the productivity and improving the system-level design in SDR applications. A library of parameterizable FPGA cores makes a design for reuse effective [14]. The timing, area, and power configurations are the key to SoC success as they allow mix-and-match of different IP cores so that the designer can apply the trade-offs that best suit the needs of the target application [15].

3. Review of Existing IP Libraries

The continuous design and implementation of a library of HDL cores, called IP cores in this paper, is increasingly driven by the desire to meet shortest possible time-to-market. This has led to greater demands of minimal development and debugging time [14, 16]. Many of the IP libraries have one or more of the characteristics listed below [14, 16–19]:

(i) Modularity

(ii) Parameterizability

(iii) Portability

(iv) Reusability

(v) Upgradability

(vi) Specific Technology Independency

(vii) Ability to consume fewer FPGA resources

Hardware designers are relying on predesigned IP cores from the IP libraries to increase productivity and reduce design time. However, many of the FPGA vendors and third-party IP libraries are static [18]. A static IP does not allow high performance to be achieved even when hardware resources or power budget is available nor does it achieve better performance to save both size and power consumption [18]. Integrating the third-party IPs can also be a challenge. It is often time-consuming and error-prone [19]. The IP libraries developed by private vendors are expensive and prohibitive to low-cost prototyping [20].

All the above shortcomings of private vendor IP libraries have led to new open-source hardware development models where reusable IPs are developed and made freely available to the public. Two examples of communities supporting open IP cores are OpenCores and GRLIB. OpenCores has the considerable number of IPs as well as Wishbone bus and its cores are accessible for free; however, OpenCores IPs are not parameterizable [20]. Likewise, GRLIB has many IP cores, interconnected by AMBA-2.0 AHB/APB bus on a SoC design. But a drawback of using GRLIB is that not all the IP cores are free [19].

4. Methodology

This objective of our investigation concerns making an effective selection of SDR cores needed to develop essential parts of an FPGA-based SDR application and from this to propose a DSL and initial selection of programming constructs that can facilitate the development of FPGA-based SDR applications without the programmer needing to have experience with HDL-based coding. The methodology we followed in this project comprises the following four aspects:

(1) Establish the design for an SDR IP core library that provides an essential collection of SDR building blocks that can serve as initial building blocks for FPGA-based SDR applications. Decide a suitable design and interfacing scheme for this set of SDR cores based on input from expert consultants.

(2) Testing the SDR cores independently on the physical FPGA-based SDR platform to confirm effective functionality of the cores.

(3) Developing a comprehensive SDR application from the SDR cores and testing this application on hardware to validate the operation of the cores working together.

(4) Benchmarking the SDR core library to ensure these cores are of an adequate standard in comparison to similar IP cores available in other libraries.

(5) Proposing a DSL-based tool-flow as an effective strategy for rapidly developing SDR applications for FPGA-based reconfigurable computing platforms using the SDR IP cores library.

The first step involved consultation with SDR experts. We interviewed and corresponded with these experts involved in FPGA-based design and implementation, both from industry and academia. Members of the engineering team at the Square Kilometre Array (SKA) were corresponded with in order to gain suggestions and feedback related to designs, processing requirements, and interfacing techniques; a total of three staff members contributed insights. We also met and corresponded with researchers involved with FPGA work at the University of Cape Town, including research scientists, postgraduate researchers, research officers, and academic staff; insights from four senior researcher staff and three postgraduate researchers were obtained at the university. The insights gained from this process were then used to prepare the design for the SDR core library and to decide the parameters that the cores should provide. The SDR cores were then coded using VHDL. Section 5 presents the subsequent design of the SDR IP cores library, starting by explaining the generic architecture that the cores fit into, then discussing the choice of SDR cores and how these were divided into DSP cores and I/O interface cores, and then explaining structure, parameters, and interface for each of the cores provided.

During the second step, testing of the individual cores was done to validate their operation; this was done using simulation and test vectors and by running the cores on a hardware platform. As part of this step, a reconfigurable hardware platform was chosen on which to test the cores. The chosen platform contained an FPGA that connected directly to a high-speed sampling card and to an Ethernet port for sending data. The results of this testing are shown in Section 6.

In the third step, we developed a representative SDR application that used the cores to confirm their operation and adequate performance when integrated as part of a complete SDR system. This application involved the development of an FM receiver for which digitally downconverted data was transferred over Ethernet to a host computer for demodulation and playback.

In the fourth step, our SDR cores were benchmarked against alternate cores available from other libraries. This benchmarking was done to confirm that the cores did not utilize an excessive number of logic elements compared to alternate solutions and that the operational clock rates of our cores were at adequate speed. Section 8 reports the benchmarking results.

In the final step, we propose a DSL to support rapid integration of the SDR cores for prototyping FPGA-based SDR applications without the programmer needing to have experience with the use of HDL coding. The proposed DSL and supporting tool-chain is presented in Section 9.

5. Design of SDR IP Cores

This section discusses the design of a library of SDR IP cores which is divided into DSP cores and I/O interface cores. For the design of DSP cores, we follow a modular coding approach using technology-independent logic elements which result in simple and reusable functional blocks. Likewise, for the design of I/O interface cores, the coding style is still modular but it comprises both technology-independent and technology-dependent functional blocks. Many commercial cores are closed source, licensed modules provided as monolithic hardware routing implementations optimized for and compatible with specific FPGA chips. A benefit of our cores is the availability of the underlying source code and therefore can be customized and be optimized further to meet the design requirements.

Although the SDR cores were tested on RHINO platform, the generic design of DSP cores using modular FPGA elements makes their portability possible without any changes or optimizations in the design, whereas porting of I/O interface cores to a wider range of platforms would still need an additional logic description or replacement of platform-specific elements in modules composed of such elements. Novice developers wanting to reuse these processing cores would consequently be advised to review the theoretical operation of the cores, possibly trying them in Octave or Matlab to gain a practical understanding of their behavior or limits, whereafter they would be familiar with the parameters concerned and be well prepared for moving to the FPGA, RHINO-based context of application of making these processing operations work in real time.

5.1. Design of DSP Cores. The design of the DSP cores presented in this paper is influenced by previous work performed for the design of hardware architectures to implement DSP algorithms. Wishbone bus slave interfaces were added to these designs to accommodate reusability, considering that the Wishbone standard is commonly used by developers making use of open-source or open-hardware IP. Some of

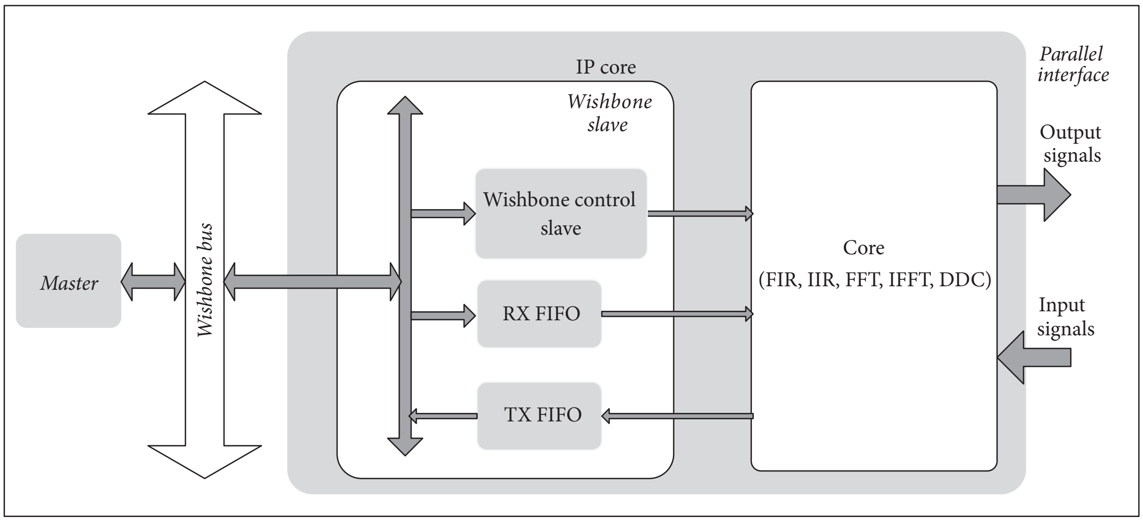


Figure 2: An overall architecture of a DSP IP core.

these DSP algorithms have been used by both commercial and open-source IP designers to implement their IP cores; however obtaining optimal results depends on the RTL coding style at a low level of design abstraction; thus the commercial solutions, in particular, are likely to have significant optimization performed for their proprietary implementations. Commercial IP cores are typically optimized for deployment on specific platforms whereas the open-source community hardly follows similar levels of consistency and standardization needed to implement such high-quality designs. In this work, we pay much attention to good RTL design conventions and practices typically obtained from consultation with experts and our own experience in FPGA design. The examples of technical recommendations that help in optimization of the cores during synthesis include (1) using modular coding style, (2) using a synchronous design approach, (3) avoiding latches, long combinatorial loops, and paths, (4) synchronizing resets at each clock domain, (5) using compatible platform-specific hardware resources such as memory, clock, and I/O management IP libraries, (6) isolating clock gating and switching, and (7) using clock PLLs and management blocks. All these design practices and many others not mentioned above make the optimization by low-level synthesis tools easier and effective. We also parameterize the cores to make it possible for future integration in the high-level synthesis tools.

The general structure of the design of the DSP cores is shown in Figure 2. These cores are implemented using fixed-point arithmetic [21] and are designed to operate with up to 130 MHz clock frequency. The configuration of core parameters such as data width, a number of filter coefficients, and FFT/IFFT length is performed through VHDL generics, hence providing a user with a wide range of options during the design process.

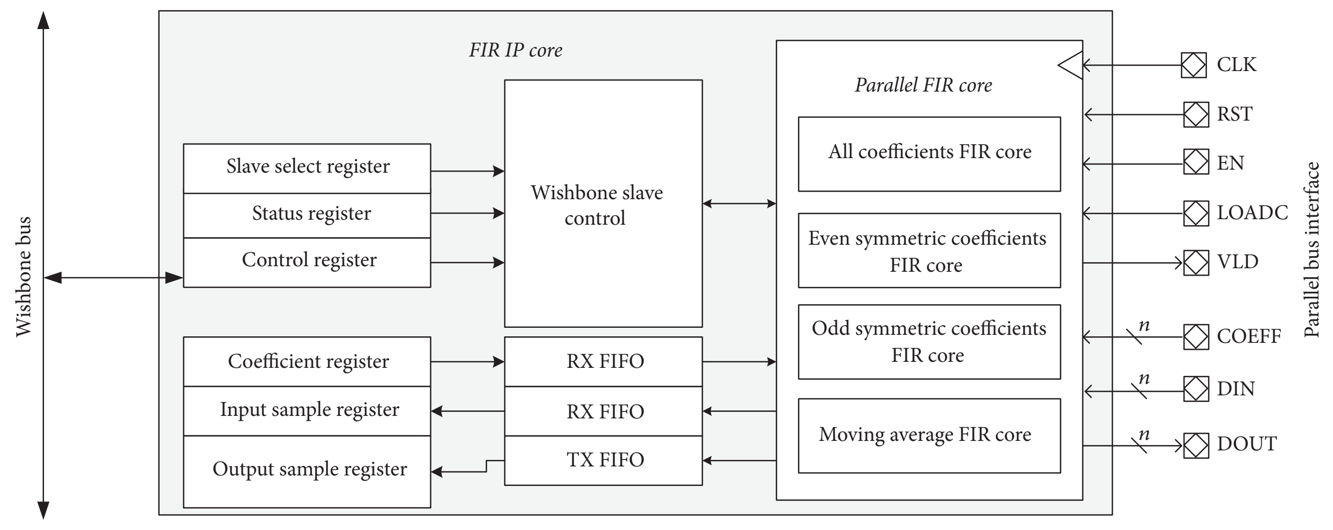
5.2. Selection of the DSP Cores. In this project, only restricted selection of IP cores was developed due to time constraint. These were chosen specially in consideration of common SDR processing needs and was also based on a thorough literature review as well as establishing priorities on what was needed for RHINO platform. The following processing DSP IP cores were chosen for inclusion into the library: FIR filter, IIR filter, FFT/IFFT modules, and a parameterized and highly scalable DDC core. Furthermore, I/O interface cores, namely, FMC150 core and UDP/IP core, were developed for their importance in providing high-speed data communication with external devices.

5.3. Connecting the DSP Cores. As shown in Figure 2, the DSP cores can be interconnected with other cores using high-speed parallel interface at operating frequency of up to 130 MHz. The cores are also designed around a common SoC interface, namely, Wishbone, whose purpose is to further improve the reusability of these cores on a SoC design. The Wishbone slave control logic manages read-write operations of the slave registers while the first in first out (FIFO) memory stores incoming input data and outgoing processed data.

5.3.1. FIR IP Core. The FIR IP core is designed to enable modularity and scalability of SDR applications with the assurance of maximum attainable clock speed. With the support of five different FIR structures, the user has a wide range of choices to synthesize efficient FIR filter that meets the design needs under consideration. The top-level block diagram of the FIR IP core is depicted in Figure 3.

The proposed FIR core realizes a number of structures which include transposed parallel FIR structure, averaging FIR filter, and two optimized realizations, namely, even and odd symmetric parallel FIR filters [22, 23]. Parallel FIR architectures are designed around low-order, high-performance applications while optimized architectures are to be used in high-order, applications and where resources are limited.

The FIR core operation depends mainly on the structure chosen by the designer and the diagram that summarizes the operational flow based on the selected FIR structure is shown in Figure 4. Except for a moving average FIR filter, all



FIR IP core parameters

Name	Description	Valid range
DIN_WIDTH	Width of data input	8
DOUT_WIDTH	Width of data output	8
COEFF_WIDTH	Width of a coefficient	8
NUM_OF_TAPS	Number of taps	2
COEFFS	Filter coefficients	Array size = taps size
LATENCY	FIR filter structure	0 = transpose 1 = odd symmetric 2 = even symmetric 3 = moving average

FIGURE 3: Architecture of FIR IP core.

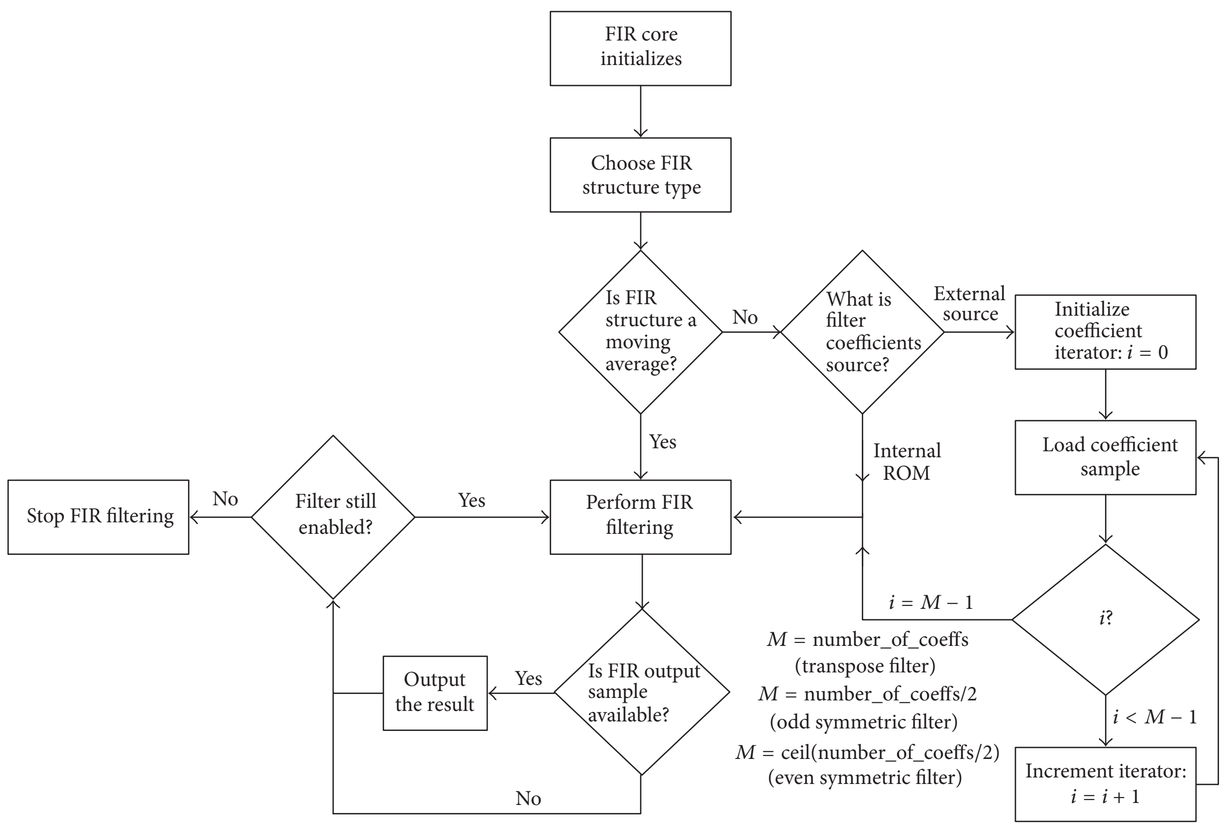
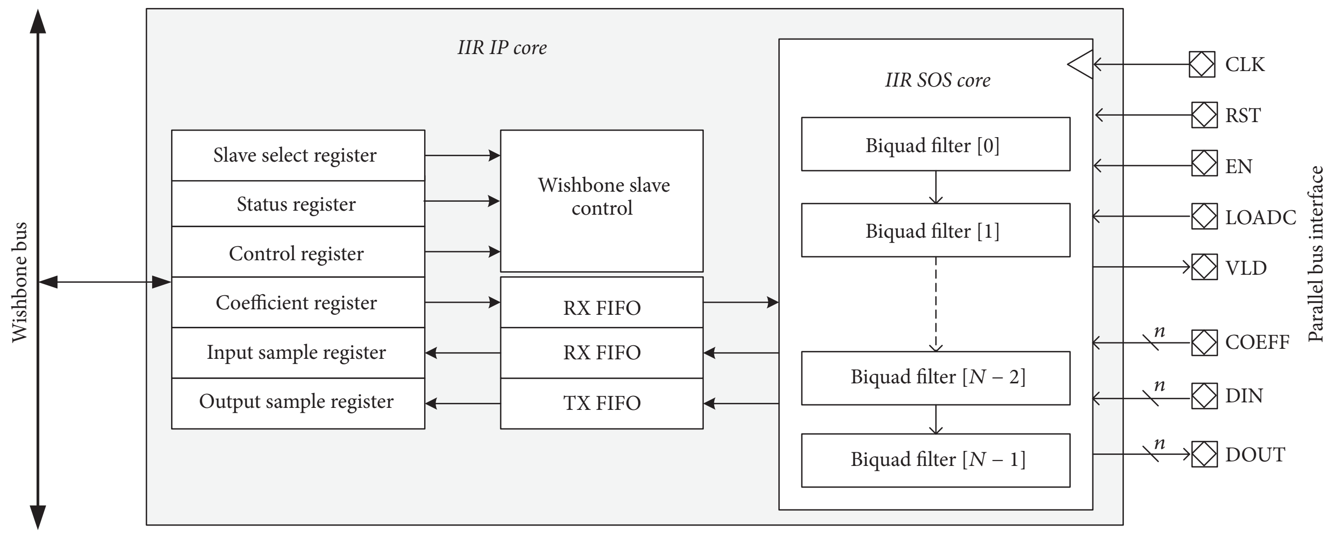

FIR core initializes
Choose FIR structure type
Is FIR structure a moving average?
No
Yes
What is filter coefficients source?
External source
Internal ROM
Initialize coefficient iterator: $i = 0$
Load coefficient sample
i?
$i = M - 1$
$i < M - 1$
Increment iterator: $i = i + 1$
M = number_of_coeffs (transpose filter)
M = number_of_coeffs/2 (odd symmetric filter)
M = ceil(number_of_coeffs/2) (even symmetric filter)
Perform FIR filtering
Filter still enabled?
Yes
No
Stop FIR filtering
Is FIR output sample available?
Yes
No
Output the result

FIGURE 4: FIR core data flow diagram.



IIR IP core parameters

Name	Description	Valid range
DIN_WIDTH	Width of data input	8
DOUT_WIDTH	Width of data output	8
COEFF_WIDTH	Width of a coefficient	8
STAGES	Number of biquad stages	1
a	Number of recursive coefficients	2
b	Number of recursive noncoefficients	3

Figure 5: An architecture of the IIR IP core.

other filter structures use coefficients stored in the distributed RAM or rather load coefficients from an external source. The user decides whether to use distributed RAM coefficients or to load them from an external memory. The FIR core does not begin filtering process until the coefficients loading is finished. If internal coefficients are used, filtering occurs immediately without waiting for loading to happen.

5.3.2. IIR IP Core. The core is built from a basic structure of a second-order IIR filter also known as biquad of Direct Form I [22]. IIR core allows cascading of the biquads to build higher order IIR filters without experiencing coefficient-sensitivity problems. This IIR structure with a cascade of biquads is called Second-Order Sections (SOS). The block diagram of IIR core designed is shown in Figure 5. The IIR core recursive and nonrecursive coefficients for each biquad are configured by the user.

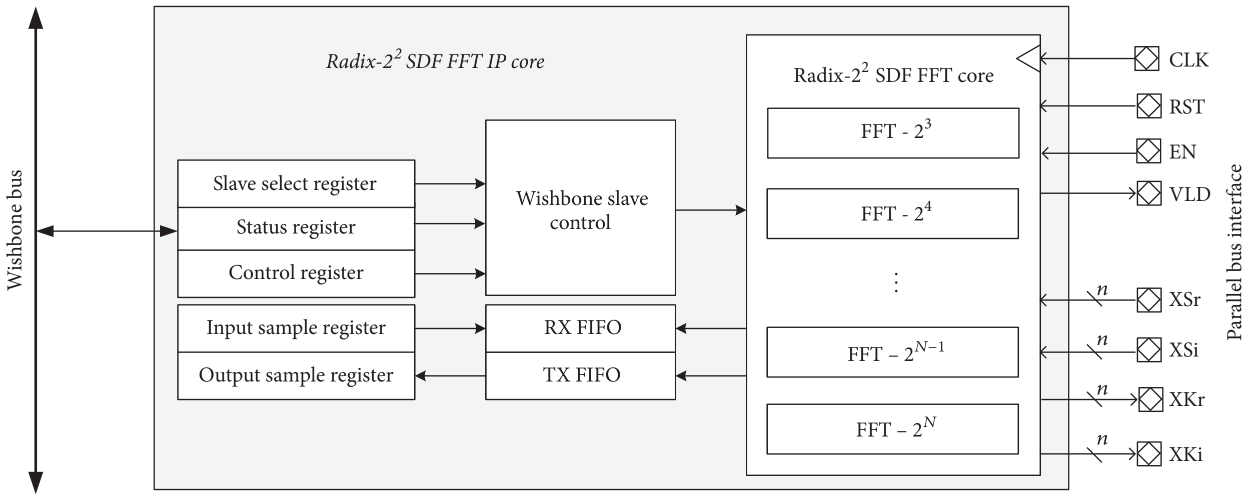
5.3.3. FFT/IFFT IP Core. Radix-2^2 Single-Path Delay Feedback (R-2^2 SDF) algorithm [24] is exploited to implement a complex pipelined R-2^2 SDF architecture of the FFT on an FPGA. The high-level block diagram of the designed FFT IP core is shown in Figure 6. The implemented FFT core is further used to implement an IFFT core. The procedure is straightforward as the IFFT is computed by conjugating the twiddle factors of the corresponding forward FFT output [25]. Some benefits of using R-2^2 to design the FFT core are that its FFT architecture has simple pipeline control and reduced multipliers by a factor of $(N - 1)/2$ compared to Radix-2 and Radix-4 which are used to design an FFT for Xilinx IP Cores Library [26]. The designed FFT/IFFT core length can be configured to 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, and 4096 by the user. However, larger size FFTs can be implemented using a Matlab script accompanying a core.

An example of 32-point R-2^2 SDF FFT is illustrated in Figure 7. Data arrives at the input in a sequential order and output data leaves the core in a bit-reversed order. For an FFT with N points, a complete stage consists of two butterflies, namely, BFI and BFII, delay feedback shift register, and a twiddle factor complex multiplier. On the other hand, half a stage only has a single butterfly which is BFI.

5.3.4. DDC IP Core. The developed DDC core is highly configurable and can be tailored easily to meet many SDR multirate applications needs. Figure 8 illustrates the top-level block diagram of the DDC IP core. This can be used in SDR applications to perform the first processing after ADC. The DDC performs the tasks such as frequency downconversion, sample rate reduction, and high-speed filtering [27].

The DDC structure is realized as shown in Figure 9. The structure is composed of Numerically Controlled Oscillator (NCO), digital mixer; Cascaded Integrator Comb (CIC) and FIR filter were all designed to complete the structure of the DDC.

5.4. Design of I/O Interface Cores. This section presents a development of FMC150 ADC/DAC interface core and a UDP/IP core for Gbe which are designed to be operational on RHINO.



FFT/IFFT IP core parameters

Name	Description	Valid range
N	Number of FFT points	$= 2^N$, $N = 3$
DOUT_WIDTH	Width of data output	8–32
COEFF_WIDTH	Width of a coefficient	8–32
TF_WIDTH	Bit width of twiddle factors	8

Figure 6: An architecture of the FFT IP core.

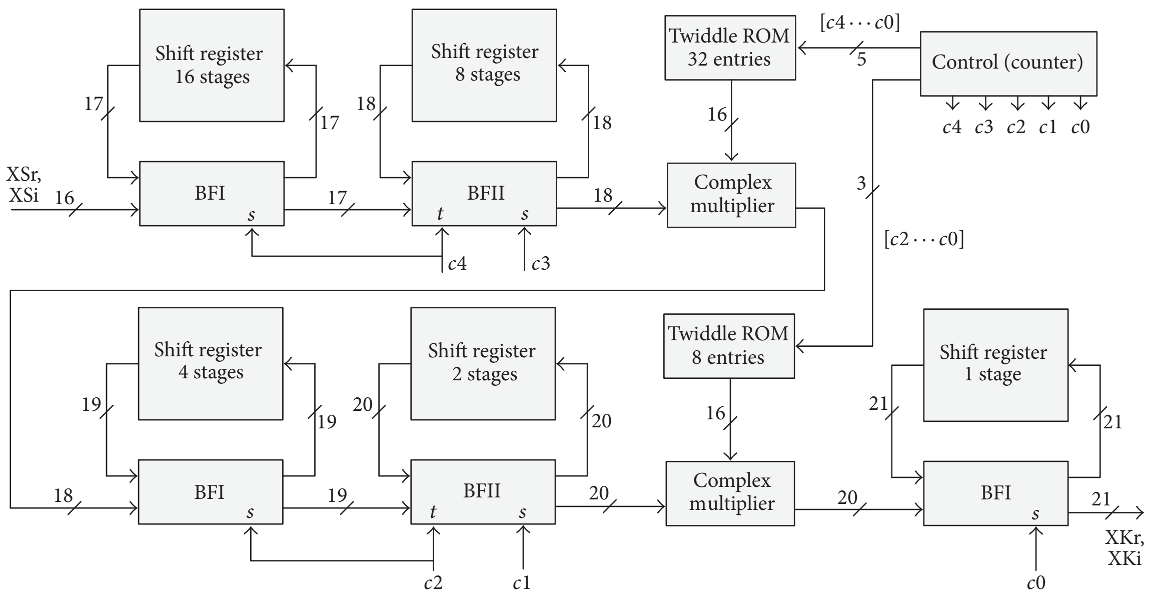
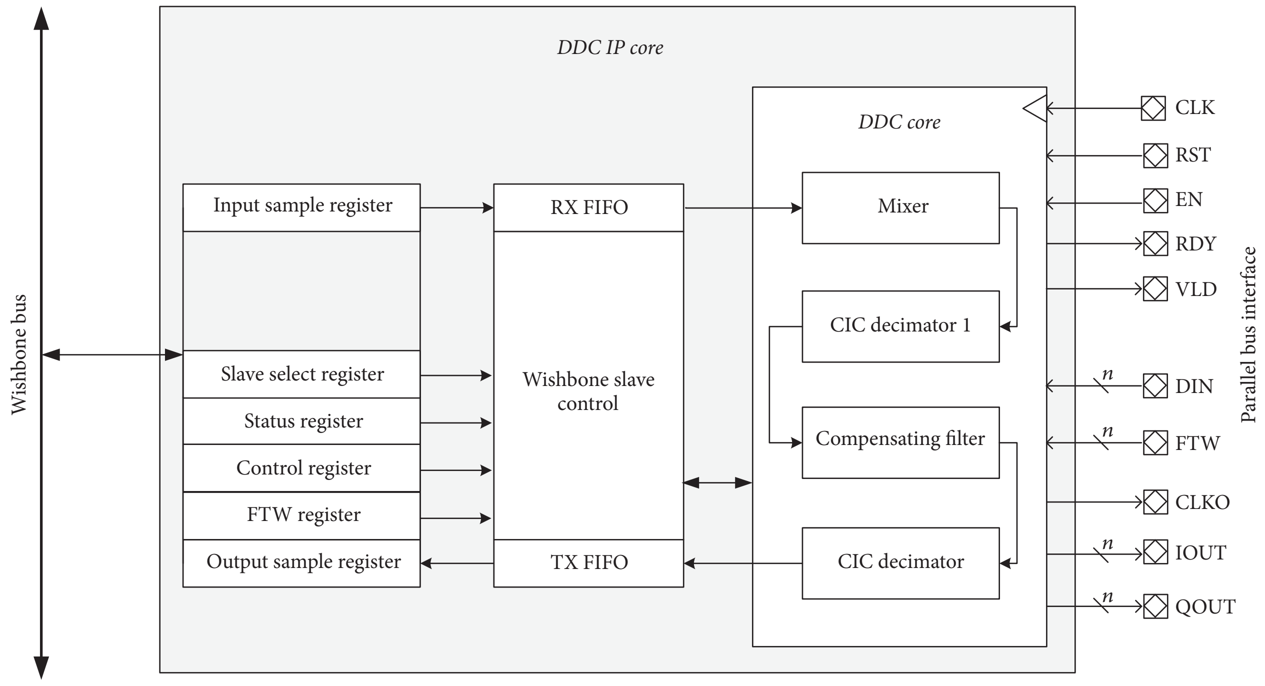


Figure 7: 32-point FFT structure using Radix-2^2 Single-Path Delay Feedback algorithm.

5.4.1. FMC150 Interface Core. The FMC150 is designed with TI's ADS62P49/ADS4249 dual-channel 14-bit 250 MSPS ADC and TI's DAC3283 dual-channel 16-bit 800 MSPS DAC. The TI's CDCE72010 PLL is the clock distribution device that provides a clock to drive the DAC and ADC. The internal clock source can optionally be locked to onboard 100 MHz or external reference clock [28].

The FMC150 core presented in this section provides Low-Voltage Differential Signaling (LVDS) interface to a 4DSP FMC150 daughter card as depicted in Figure 10. A design example in Figure 10 is configured on ADC sampling rate of 61.44 MSPS and sends digital samples to DAC at 61.44 MSPS rate. The maximum sampling rates that FMC150 core were tested on using RHINO are 163.84 MSPS and 245.76 MSPS for ADC and DAC, respectively.

5.4.2. UDP/IP Core. In order to make RHINO Gbe operational, an FPGA-based Gbe core is needed to configure,



DDC IP core parameters

Name	Description	Valid range
N	Number of FFT points	8
DOUT_WIDTH	Width of data output	8
PHASE_WIDTH	NCO phase width	8
PHASE_DITHER_WIDTH	Phase dither width	PHASE_WIDTH
SELECT_CIC1	Activate CIC1 of a DDC	0 or 1
NUMBER_OF_STAGES1	Number of CIC1 Stages	>0
DIFFERENTIAL_DELAY1	Differential delay of CIC1	1 or 2
SAMPLE RATE_CHANGE1	Decimation factor of CIC1	>0
SELECT_CFIR	Use a compensating FIR filter of DDC	0 or 1
NUMBER_OF_TAPS	Number of coefficients	>0
FIR_LATENCY	Type of FIR filter structure	0, 1, 2, 3
COEFF_WIDTH	Coefficient bit width	8
COEFFS	Quantized integer filter coefficients	Array size = taps size
SELECT_CIC2	Activate CIC2 of a DDC	0 or 1
NUMBER_OF_STAGES2	Number of CIC2 stages	>0
DIFFERENTIAL_DELAY2	Differential delay of CIC2	1 or 2
SAMPLE RATE_CHANGE2	Decimation factor of CIC2	>0

Figure 8: An architecture of the DDC IP core.

monitor, and control the Ethernet interface. This section presents a design of a UDP/IP core based on the combination of Internet Protocol version-4 (IPv4) and User Datagram Protocol (UDP) in order to provide a high-speed and efficient solution for communication over a Gbe.

FPGA devices require Ethernet Media Access Controller (EMAC) to interface with the physical layer (PHY) chip on the board [29]. RHINO uses an integrated Marvell 88E111 PHY chip. The PHY is needed for the FPGA to connect with external devices. The user logic can be deployed to configure the EMAC physical interface [29] in a form of wrapper files. In our case, the wrapper files configure the OpenCores Trimode MAC [5] which is published under the GNU Lesser General Public License (LGPL). This is a very cost-effective and nonrestrictive solution in comparison with proprietary Media Access Controllers (MACs) such as Xilinx's Trimode Ethernet Media Access Controller (TEMAC) [30] which is costly. Furthermore, the OpenCores Trimode MAC IP core supports data rates of 10, 100, and 1000 Mbps and is compliant with IEEE 802.3 specification [5]. Our UDP/IP core is only configured on 1000 Mbps speed.

The architecture of the UDP/IP core is illustrated in Figure 11. Address Resolution Protocol (ARP) is used to resolve the sender and receiver MAC addresses before packet data communication. An OpenCores Trimode MAC [5] is responsible for delivering data over a shared physical channel. The MAC consists of two user interfaces that simplify the connection to a PHY. It encodes/decodes packet data

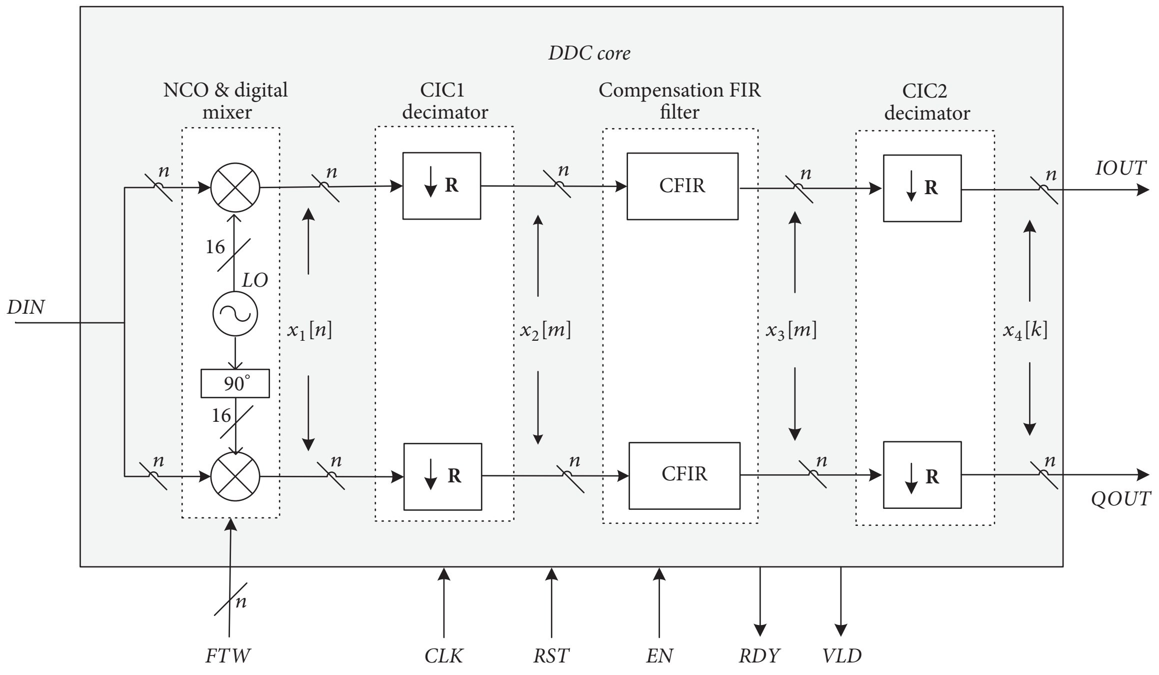


Figure 9: A structure of the Digital Down Converter.

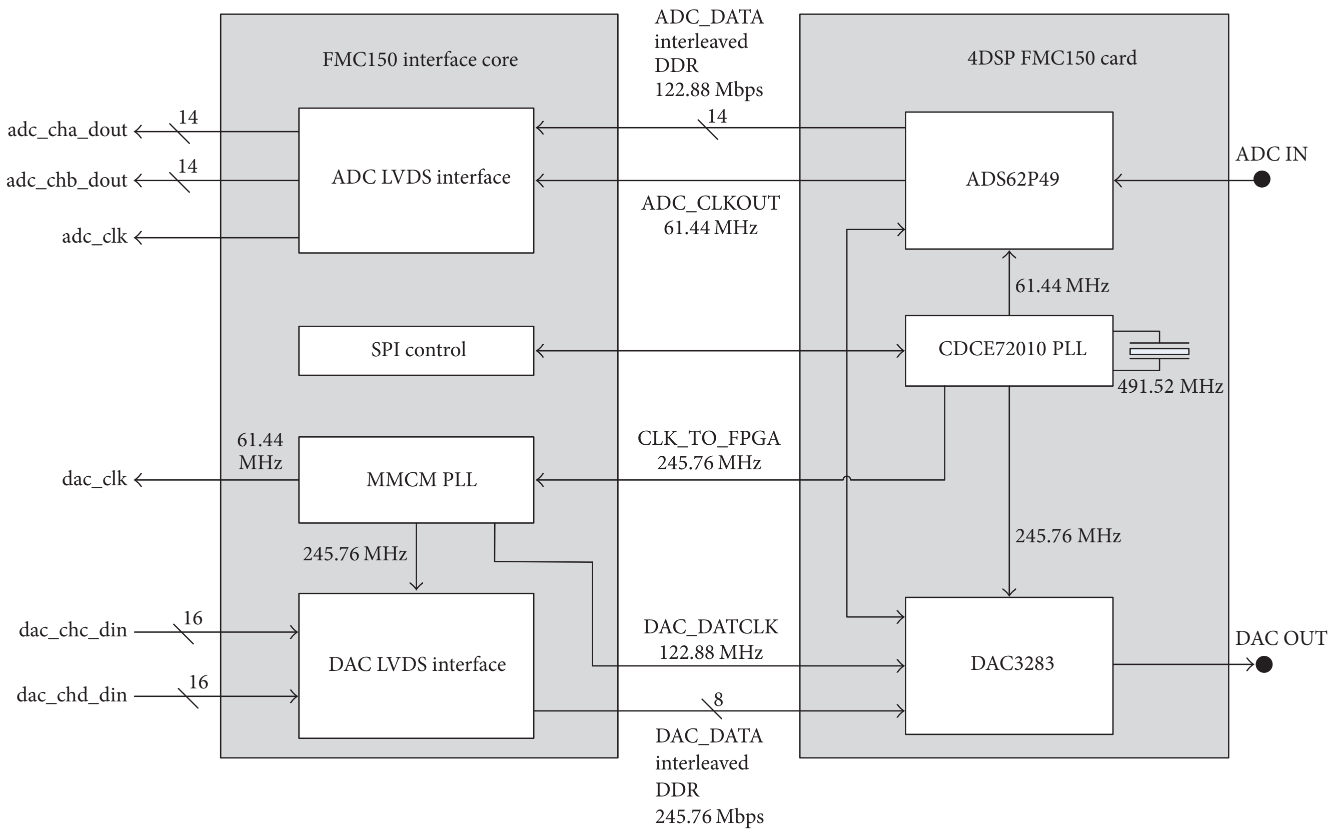
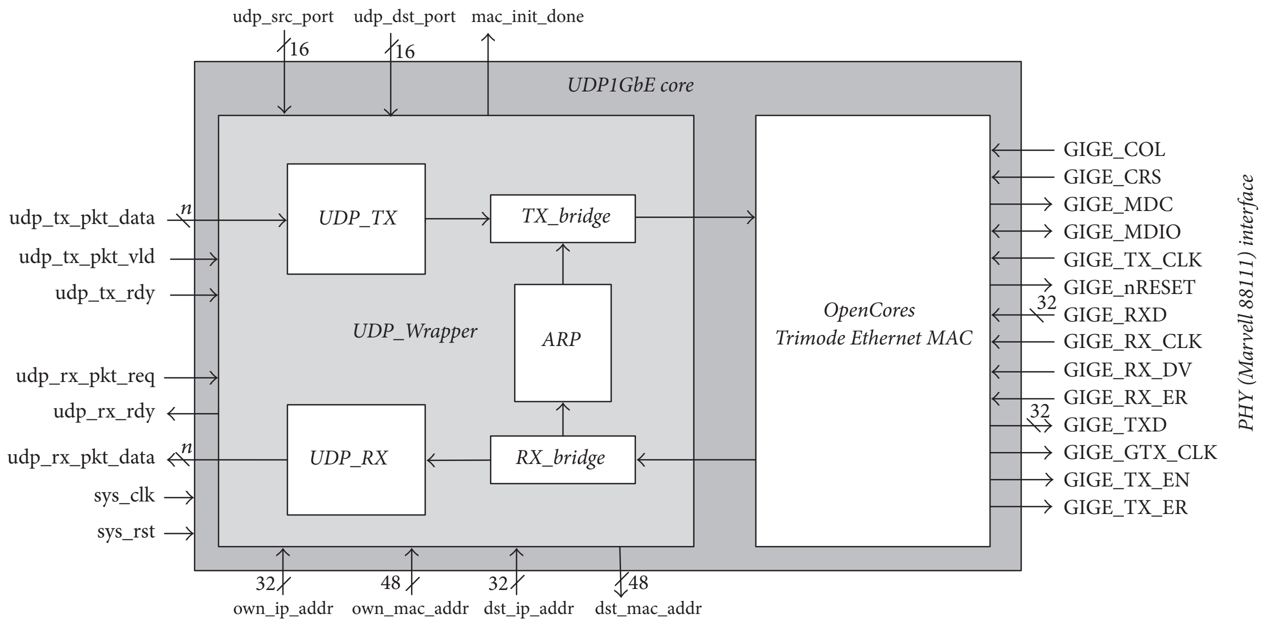


Figure 10: Structure of FMC150 core interfacing with the 4DSP FMC150 card.



UDP1GbE core parameters

Name	Description	Valid range
UDP_TX_DATA_BYTE_LENGTH	Transmitted payload size in bytes	1–1474
UDP_RX_DATA_BYTE_LENGTH	Received payload size in bytes	1–1474

FIGURE 11: Structure of UDP/IP core for interfacing and control of a Gbe (uses OpenCores Trimode Ethernet MAC [5] to interface with a PHY).

to/from PHY during transmission and reception of data using Gigabit Media-Independent Interface (GMII) between a MAC and a PHY. The second interface is serial and it is called Management Data Input/Output (MDIO) bus. It transfers configuration data to a PHY and it is also used to read PHY status registers. The IPv4 is used by the designed UDP/IP core to deliver messages between the RHINO and a destination device. The IP addresses are configured statically and they must be in the same subnetwork for successful communication to happen. UDP is chosen as a transport layer protocol. It is used in this design for its simplicity and the fact that it supports high-speed and real-time data transfers [31, 32].

6. Testing and Results

In order to verify the functionality and correctness of these cores, testing which involved behavioral and functional simulation was performed. Each DSP core was successfully synthesized on Spartan-6 of RHINO and tested from input data generated using Matlab. After the core had processed the data, the results were stored in an output file as a vector of samples. Matlab scripts were used to plot graphs and perform further signal processing of the results for analysis. The general experimental setup for DSP cores is shown in Figure 13. The operating frequencies of 100 MHz were used when testing the FIR, IIR, and FFT cores. For a DDC core, 122.88 MHz of clock frequency was used.

6.1. Overview of RHINO Platform. Reconfigurable Hardware Interface for ComputatioN and RadiO (RHINO) is a standalone FPGA processing board and has commonalities with the better known Reconfigurable Open Architecture Computing Hardware (ROACH); however, it is a significantly cutdown and lower-cost alternative which has similarities in the interfacing and FPGA or processor interconnects of ROACH. RHINO was designed at the University Of Cape Town and is largely aimed around a lower-cost, totally open-source FPGA board which provides a good platform for the development of software-defined radio applications [4]. The RHINO platform was designed to be a combination of an education and training platform for learning about reconfigurable computing and as a research and prototyping platform for studies related to SDR [4, 33].

The two main processing elements of RHINO include ARM processor and Spartan-6 FPGA as shown in Figure 12. The computationally intensive functions are processed by the FPGA while the ARM processor provides configuration, control, and interface function with FPGA through Berkeley Operating System for Reprogrammable Hardware (BORPH) [4, 34]. BORPH is an extended Linux kernel that allows control of FPGA resources as if they were native computational resource [34]. This, as a result, allows users to program the FPGA with a given design or configuration and run it as software process within Linux.

Other building blocks of RHINO include FMC connectors which enable interface with ADC, DAC, and mixed

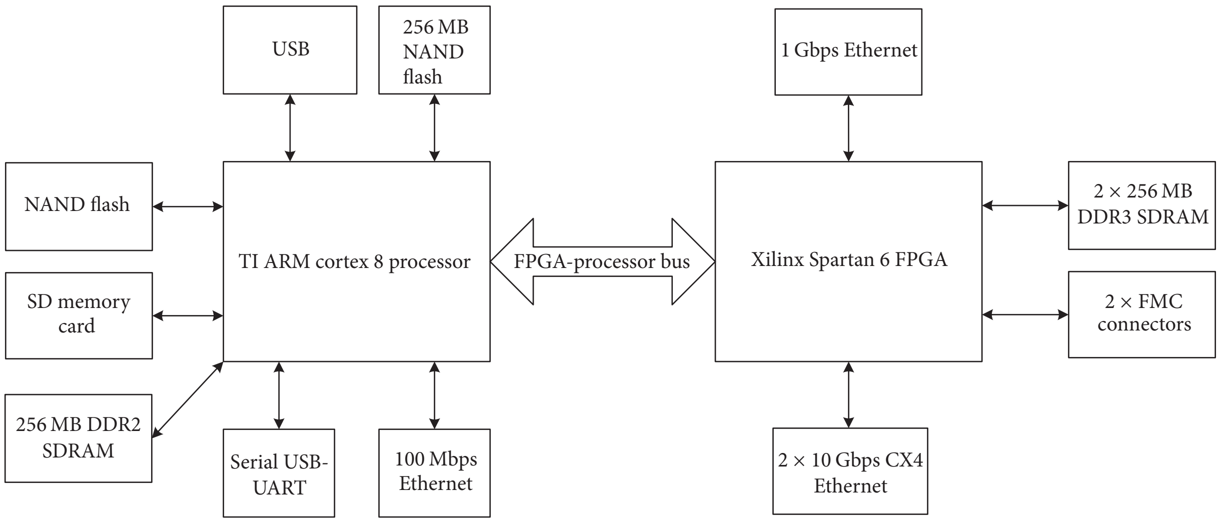


Figure 12: An architecture of RHINO platform building blocks.

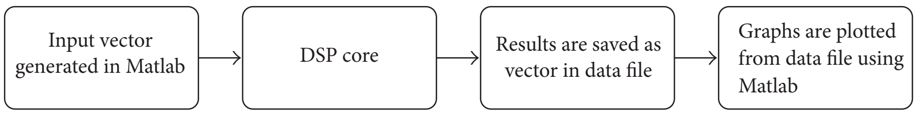

Input vector generated in Matlab
DSP core
Results are saved as vector in data file
Graphs are plotted from data file using Matlab

Figure 13: Process of configuring a DSP core.

signal daughter cards, supporting sample rates over 1GS/s [35]. The 1/10 Gbe connectors provide a high-speed network connection between the FPGA and remote devices using standard TCP or UDP transport layer protocols to convey packets of data.

6.2. Testing FIR Core. The FIR core was verified by designing length $L = 95$ FIR bandpass filter to specifications shown in Figure 14. The resulting Parks-McClellan optimal FIR coefficients of 16-bit width are illustrated in Figure 14(b) in a form of filter frequency response. The filter was tested on an input signal consisting of a sum of sinusoids at frequencies 440, 800, 2200, and 2500 Hz as shown in Figure 14(a) and only a 2200 Hz was isolated by a bandpass filter. The results shown in Figure 14(d) closely match with the results of the ideal filter in Matlab shown in Figure 14(c); however, the Signal-to-Noise Ratio (SNR) has slightly decreased due to quantization and round-off errors.

6.3. Testing IIR Core. Testing the IIR filter was similar to FIR testing in Section 6.2 except for filter response shown in Figure 15(b) which was designed with Chebyshev Type I filter to specifications shown in Figure 15. The results of the IIR core obtained are shown in Figure 15(d) which closely match the ideal Matlab results shown in Figure 15(c). In comparison to the FIR core results, the IIR core is highly selective and uses fewer coefficients leading to better results.

6.4. Testing FFT/IFFT Core. This testing involved generating an input vector (length = 1024) of a rectangular pulse as shown in Figure 16(a) and processing it with a 1024-point FFT/IFFT core. This was used at the input of the core operating in FFT mode. The output of the FFT core as shown in Figure 16(c) was later used as an input data to the IFFT core. The FFT core yielded the sinc waveform in Figure 16(c) which was the expected Fourier transform of a pulse waveform. This also matched with the Matlab generated FFT of the pulse wave shown in Figure 16(b). As expected, the IFFT core produced the original rectangular pulse waveform which is illustrated in Figure 16(d).

6.5. Testing DDC Core. Using Matlab and 122.88 MSPS sample rate, an FM signal vector was created by modulating 94.5 MHz sine wave with 15 kHz baseband signal. This vector was used as an input to a DDC core. Due to bandpass sampling used, the FM signal was centered at 28.38 MHz after sampling. Similarly, the carrier was also located at 28.38 MHz. In order to convert it to baseband, NCO signals are multiplied with input FM signal using the mixer. The product then becomes the desired signal component centered at DC and a spurious harmonic located at 56.76 MHz as shown in Figure 17(b). This undesired signal component was removed by a CIC filter which decimated the 122.88 MSPS ADC sample rate by a factor of 1 : 128 resulting in 960 kSPS sample rate as shown in Figure 17(c). The nonideal response of the CIC filter was corrected by introducing a compensation FIR filter in the final stage of the DDC and its output is shown in Figure 17(d).

After the digital downconversion, the FM demodulator was used to demodulate the FM signal. The magnitude

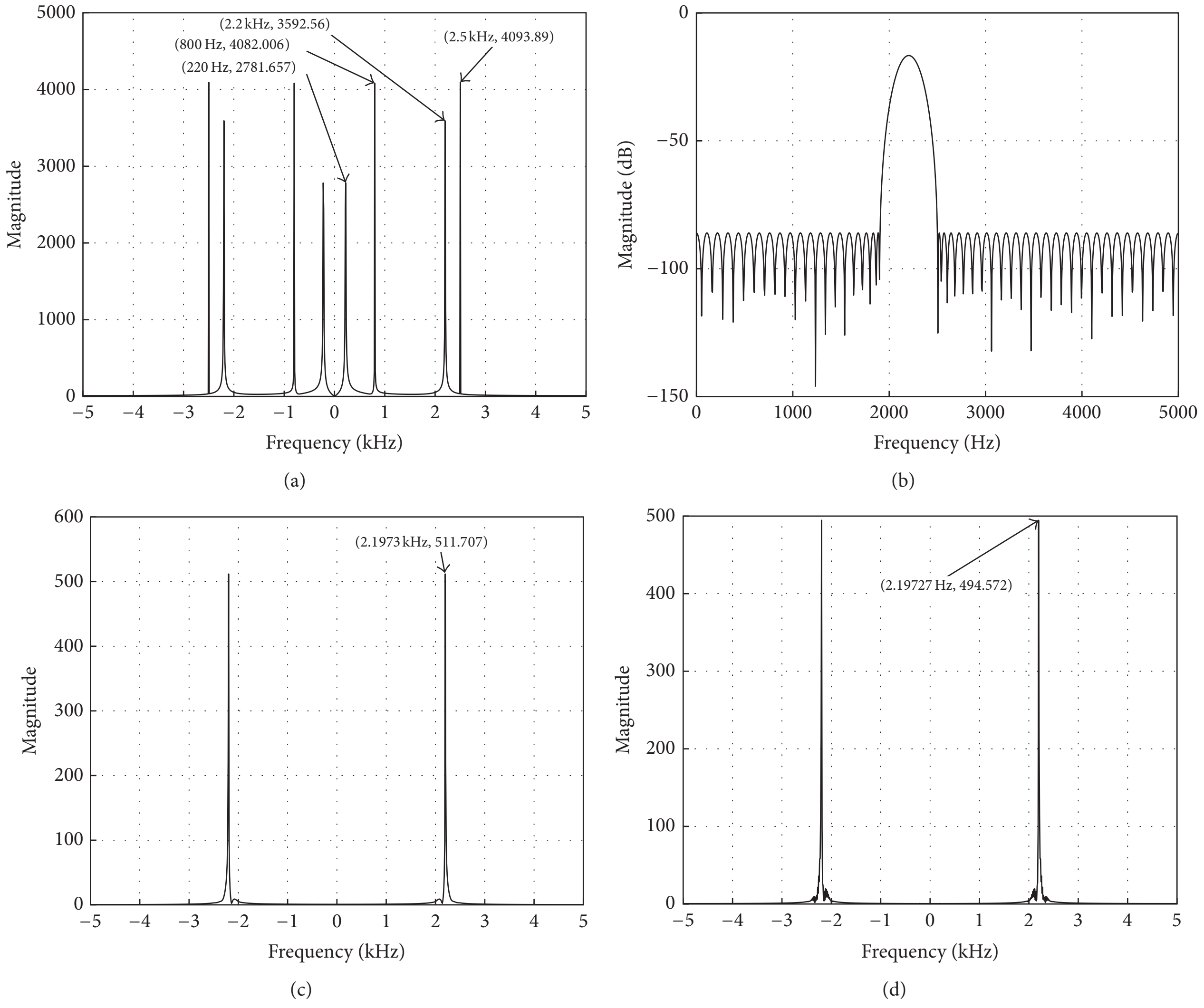


Figure 14: The results of FIR filter testbench. (a) Magnitude spectrum of input signal. (b) FIR filter frequency response (filter response parameters: *sampling frequency = 10 kHz*, *lower cutoff frequency = 2.190 kHz*, *higher cutoff frequency = 22.15 kHz*, *passband ripple = 3 dB*, *stopband attenuation = 80 dB*, and *number of coefficients = 95*). (c) Magnitude spectrum of Matlab FIR filter output. (d) Magnitude spectrum of FPGA FIR filter output.

spectrum and amplitude versus time graphs of the signal after demodulation are shown in Figures 18(a) and 18(b). This output has a transient response which is the effect of the FM demodulator. When the transient was removed, this resulted in steady-state response whose magnitude spectrum and time domain graphs are shown in Figures 18(c) and 18(d), and they represent the recovered 15 kHz baseband signal.

6.6. Testing FMC150 ADC Core and UDP/IP Core. The experimental setup is shown in Figure 19 and it involved stream-based processing that incorporated ADC and Gbe cores. The 20 MHz input tone to a 49.152 MSPS ADC was generated with a function generator as illustrated in Figure 20(a). The FMC150 interface core was used to capture the ADC samples and these samples were sent to a Desktop Personal Computer (PC) via a Gbe using UDP/IP core. At the PC end, the received samples were plotted as shown in Figure 20. The Spurious Frequency Dynamic Range (SFDR) of the ADC signal was measured as 44 dBc, about 55% of the vendor specified ADC figure. The pronounced spurious harmonics are due to the high-level of distortion in the 10 dBm input signal from a function generator. A better function generator with a low distortion effect would improve results. The throughput speed recorded on a Gbe using Wireshark was 98.62 MB/s which is 89.65% of the theoretical figure of 110 MB/s (as specified by Huang et al. [36]).

6.7. Testing FMC150 DAC Core. The block diagram of the experimental setup is illustrated in Figure 21. The experiment used the NCO core designed in Section 5.3.4 to synthesize two different sine waveforms of frequencies 17.23 MHz and 28.38 MHz. The digital samples were sent to the DAC at 61.44 MSPS sampling rate. The DAC, in turn, converted

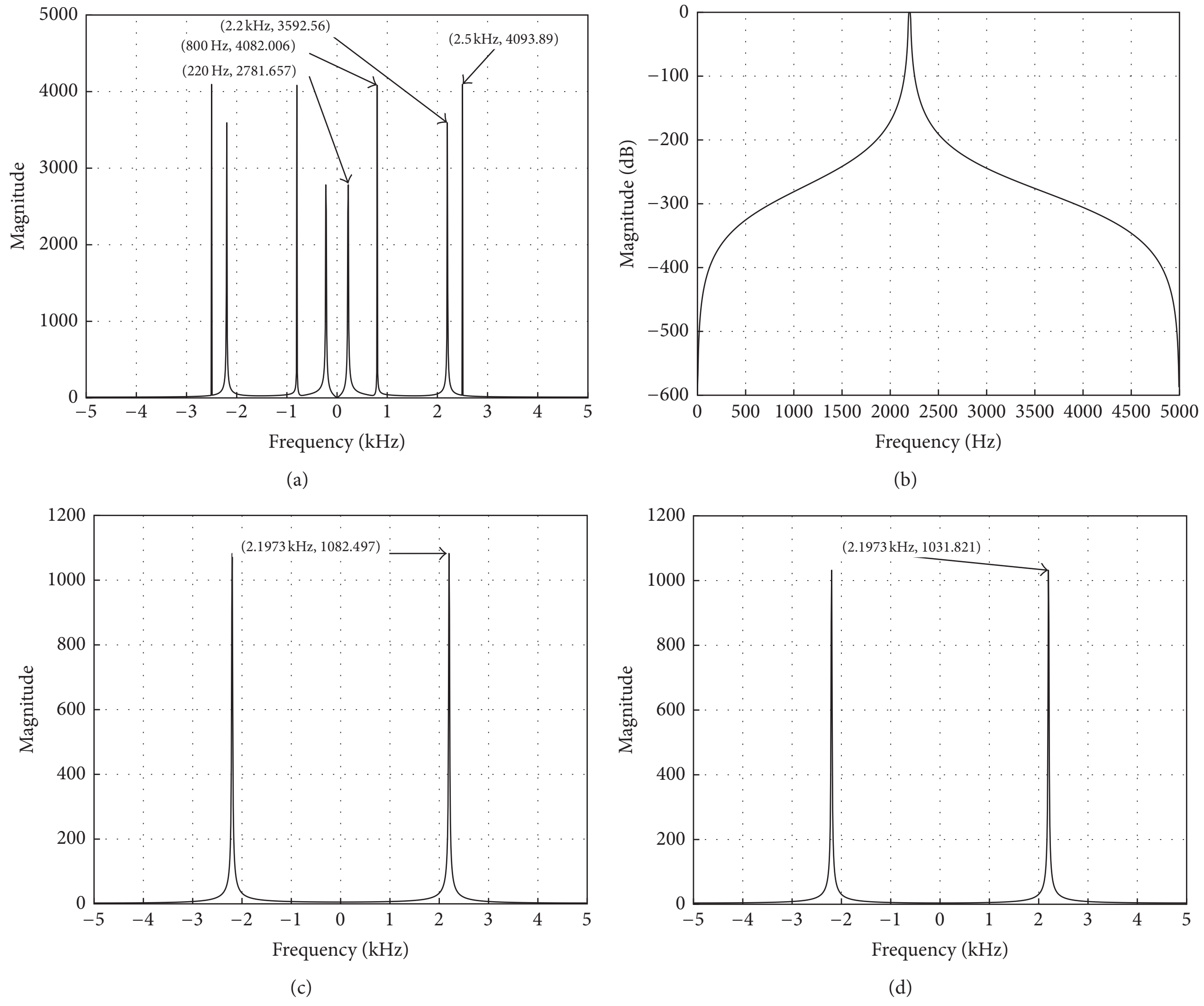


Figure 15: The results of IIR filter testbench. (a) Magnitude spectrum of input signal. (b) IIR filter frequency response (filter response parameters: *sampling frequency = 10 kHz*, *lower cutoff frequency = 2.190 kHz*, *higher cutoff frequency = 22.10 kHz*, *passband ripple = 0.1 dB*, *stopband attenuation = 200 dB*, *number of coefficients = 6*, and *number of sections = 6*). (c) Magnitude spectrum of Matlab IIR filter output. (d) Magnitude spectrum of FPGA IIR filter output.

digital data into analog signals which were measured on a spectrum analyzer and the results are shown in Figure 22.

7. Validation: Development of FM Receiver

This section reports on the development of a wideband digital FM receiver. This is used for validation of the proposed SDR IP cores library. Using the SDR IP cores which incorporate DSP cores and I/O interface cores, this prototype serves as a proof of concept that the cores can be used not only in this FM receiver design but also to prototype other real-time SDR applications. The complete design of FM receiver comprises an analog RF front-end circuitry and digital receiver which forms the largest part of the FM receiver processing.

7.1. Analog Front-End. The block diagram of the analog RF front-end design is shown in Figure 23 and is sensitive to −65 dBm FM signal. The indoor FM antenna with a variable gain of 36 dB receives the FM signal in the frequency range of 88–108 MHz. The front-end also provides a bandpass filtering of FM band and a total gain of 75 dB to make the FM signal compatible with the input swing of the ADC. The overall gain is determined as

$$\begin{aligned}\text{Total Front-End Gain (dB)} &= P_{\text{fm-signal}} + P_{\text{fmc150-ADC}} \\ &= 10 + 65 = 75\,\text{dB},\end{aligned} \quad (1)$$

where $P_{\text{fm-signal}}$ is FM signal power in dBm and $P_{\text{fmc150-ADC}}$ is ADC input signal power in dBm.

7.2. Digital Receiver. The digital receiver processing is implemented with a DDC core and the FM demodulator. The block diagram showing the processing blocks is shown in Figure 24. The 20 MHz bandwidth RF signal is digitized with a 14-bit

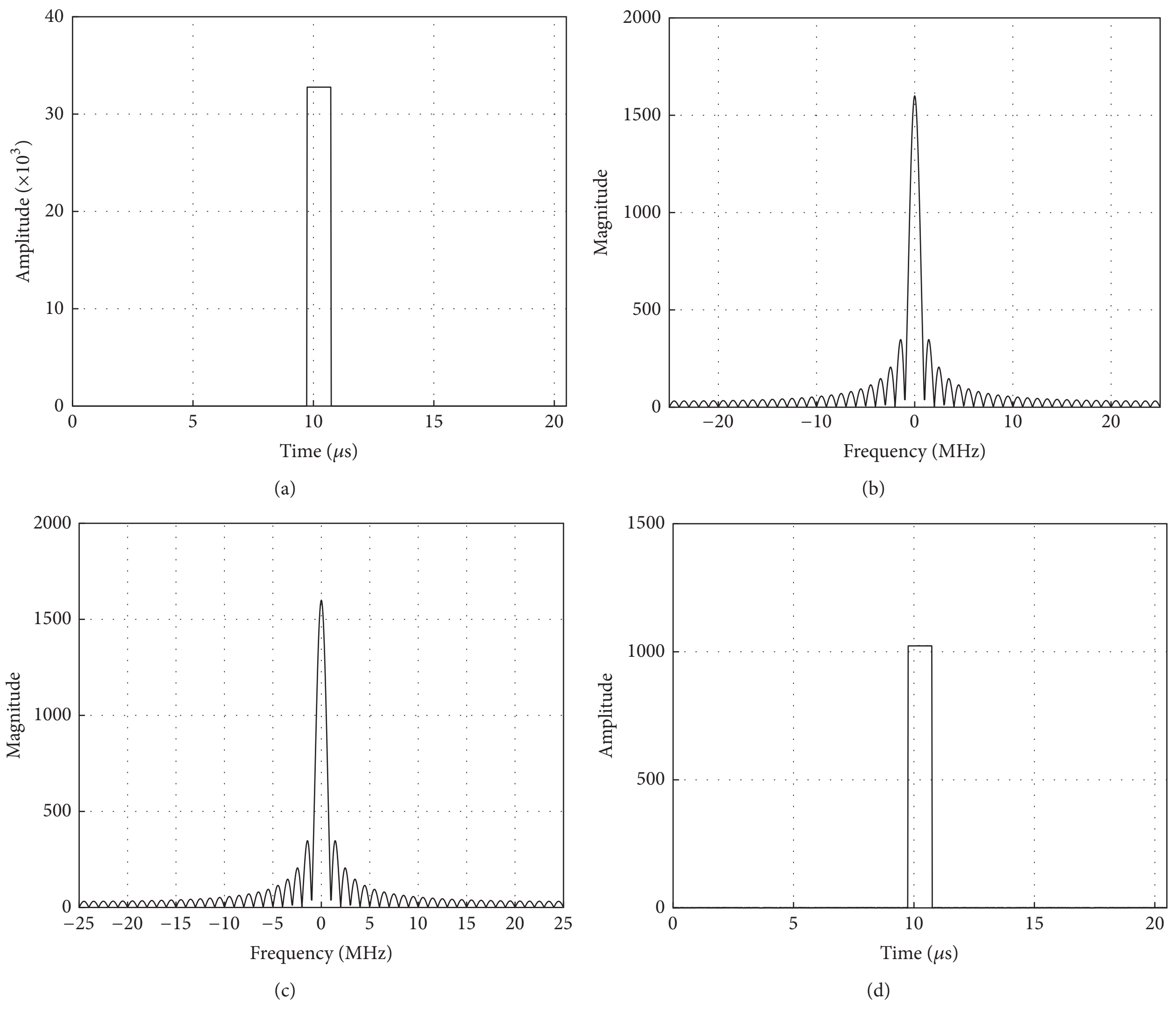


Figure 16: Matlab and FPGA results of a 1024-point FFT and IFFT core tested with rectangular pulse input waveform. (a) Rectangular pulse waveform, (b) 1024-point FFT core output using Matlab, (c) 1024-point FFT core output using FPGA, and (d) 1024-point IFFT core output using FPGA.

precision ADC. The ADC sampling rate f_s is determined by using a bandpass sampling criterion [37] in (2) where n falls in a range 1 to 5. We choose $n = 2$ and this results in a wide frequency range of 108 MHz–176 MHz valid for ADC bandpass sampling. We choose 122.88 MSPS as the ADC bandpass sampling speed. The chosen ADC sample rate results in the digitized FM signal downshifted from 88–108 MHz band to 14.88–34.88 MHz band.

$$\frac{2f_H}{n} \leq f_s \leq \frac{2f_L}{n-1}, \tag{2}$$

where n is given by $1 \leq n \leq (f_H/(f_H - f_L))$, f_H is high frequency, and f_L is low frequency.

The 14-bit samples received from the ADC are extended to 16-bit signed words which are directed into the DDC core input. To generate a complex baseband I/Q signal, the sine and cosine waveforms are generated using the NCO core and then multiplied with the FM signal using digital quadrature mixer. The frequency of the NCO output is equal to the frequency of a sampled FM signal and it ranges between 14.88 MHz and 34.88 MHz.

After mixing down the FM signal using the quadrature mixer, the image and mixer products are eliminated by the CIC filter which uses zero multipliers in its implementation. This CIC filter also decimates the 122.88 MSPS ADC rate to 960 kSPS by decimation ratio of 1 : 128. Despite its low cost and efficient and simple implementation, the CIC filter introduces undesirable droop in its filter response passband [38]. To correct this nonflat response in the passband of the CIC filter, the compensation FIR (C-FIR) filter is used.

The CIC and C-FIR filter specification parameters and their respective filter responses are shown in Figure 25. In this same figure, the total filter response is shown which results after decimation and compensation.

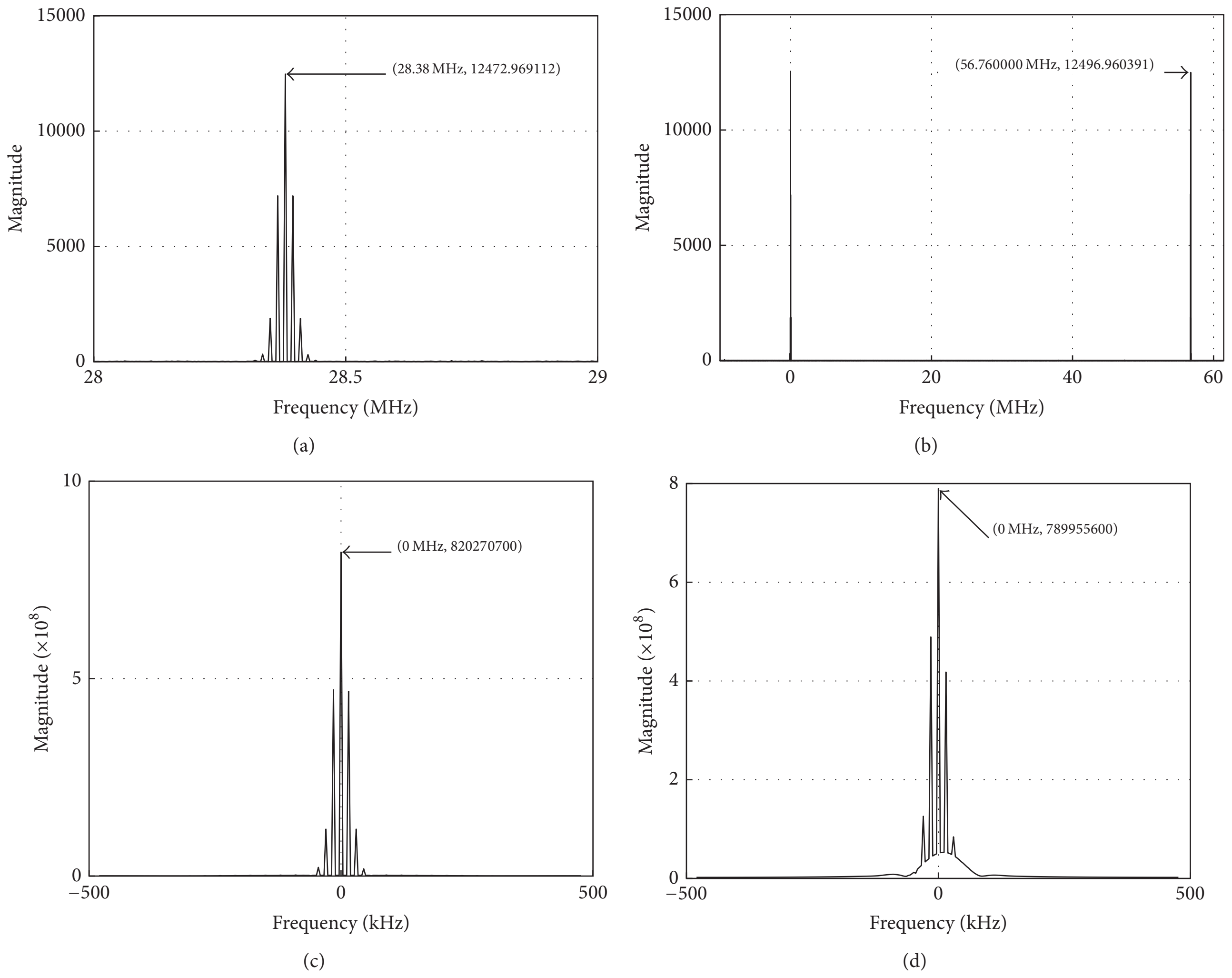


Figure 17: Results of DDC core and FM demodulator. (a) FM-modulated signal generated in Matlab, (b) magnitude spectrum of mixer, (c) magnitude spectrum of CIC-1, and (d) magnitude spectrum of a C-FIR filter.

7.3. Data Packetization. A data packet that is composed of 32-bit samples needs to be generated before the actual construction of a UDP packet. Each 32-bit is decomposed into 16-bit I/Q data samples. Generation of data packets is facilitated by a length-N double buffer that lies between a DDC core and UDP/IP core as shown in Figure 24. In this example, we choose $N = 33$. As illustrated in Figure 26, a double buffer accepts samples from a DDC core and writes them to *BUFFER 1* buffer. The UDP/IP core then reads samples from *BUFFER 2* as a unit to generate a data packet that is padded to a data field of a UDP packet. Furthermore, double buffering also solves a producer-consumer problem during concurrent read and write process by DDC core and UDP/IP core, respectively. Each data packet generated forms part of the UDP data field where each 32-bit sample in a packet is represented by 16-bit I/Q samples as shown in Figure 27.

7.4. Final Test: FM Receiver. The block diagram showing the experiment setup is shown in Figure 28. The test was performed by tuning to a local radio station at 94.5 MHz (K-FM). The final output of the FPGA processing was complex I/Q samples centered at DC as shown in Figure 29(a). These samples were then demodulated in Matlab using arctan/differentiation FM demodulator at the PC end. The output of the FM demodulator is a real-valued signal and is shown in Figure 29(b). The results clearly show the mono audio, pilot tone, stereo audio, and RBDS spectral components; however, stereo audio and RBDS are not distinguished due to a weak FM signal received by the ADC. The ADC tends not to be sensitive to signals with power way below 10 dBm. Increasing the analog RF front-end gain will improve results.

8. Benchmark Results

We benchmark our IP cores using Xilinx ISE v14.7, targeting the Spartan-6 xc6slx150t FPGA found on RHINO platform. We do the same with cores found in Xilinx DSP core library and OpenCores where they both represent commercial and open-source cores, respectively. The benchmark results

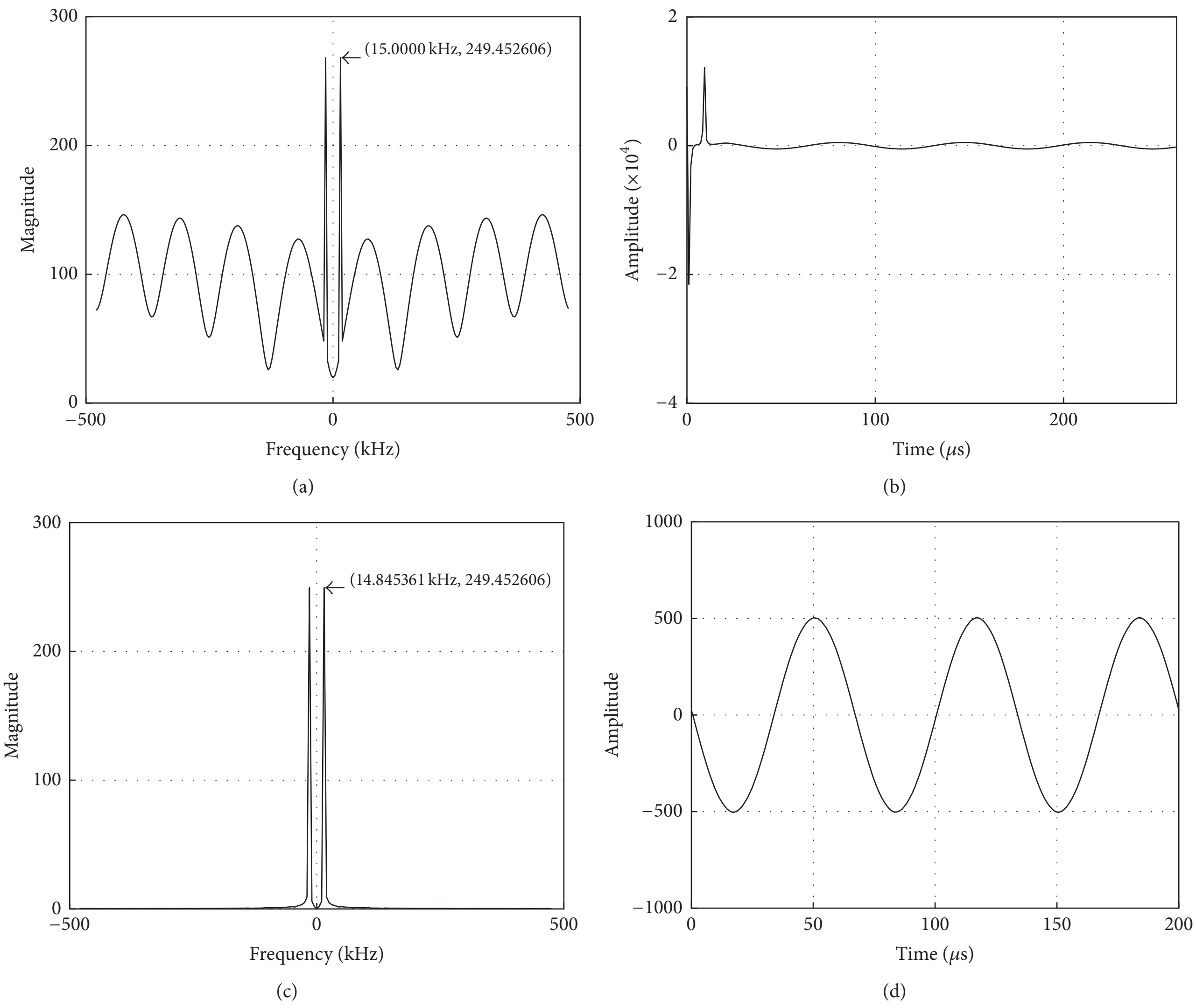


Figure 18: FM demodulator output showing the demodulated signal with transients and after removing the transients. (a) Magnitude spectrum of FM-demodulated signal, (b) FM-demodulated signal; (c) magnitude spectrum of FM-demodulated signal without transients, (d) FM-demodulated signal without transients.

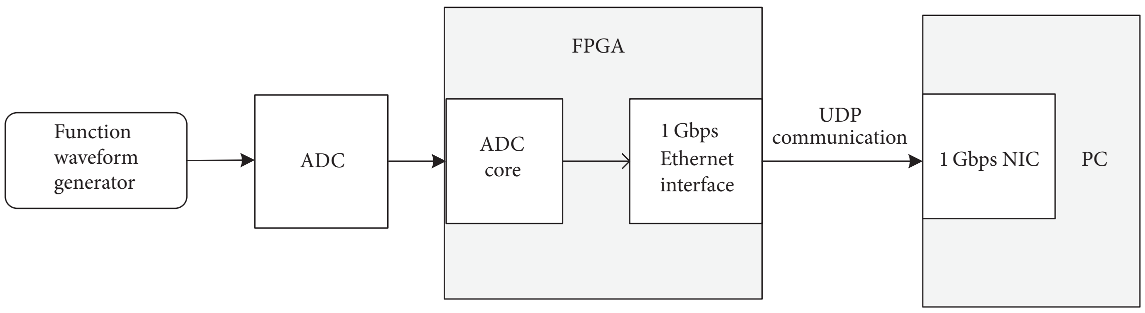
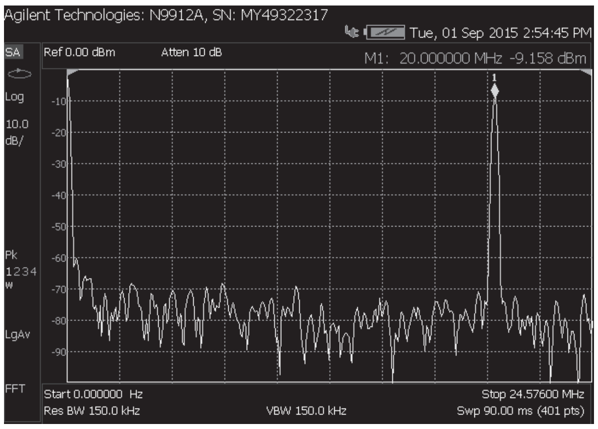


Figure 19: Experimental setup for a streaming core using FMC150 ADC core and Gbe core.

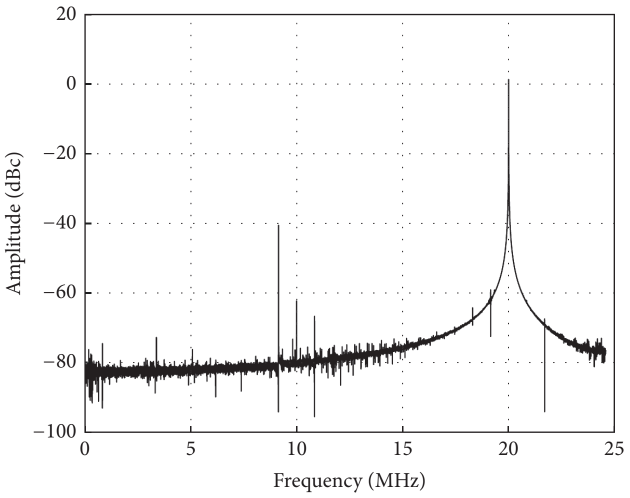
shown in Table 1 use metrics of FPGA resource utilization and the maximum clock speed that can safely be used to execute each core. The key parameter values for each of the SDR cores are as follows:

(1) The number of coefficients for the FIR core is 21 with data width set to 16 bits.

(2) The IIR core is benchmarked on 16 stages with 16-bit data.



(a) ADC input of 20 MHz tone



(b)

FIGURE 20: 20 MHz tone ADC output streamed using UDP. (a) ADC input of 20 MHz tone generated with function generator, (b) FFT for 20 MHz ADC signal captured on a PC after streaming.

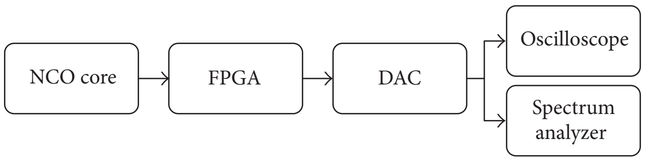
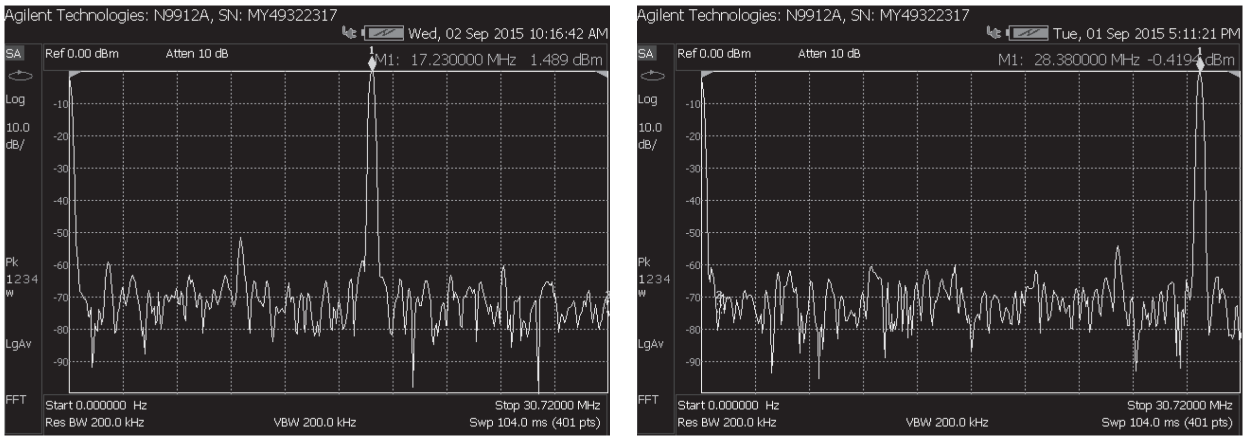


FIGURE 21: A block diagram showing experimental setup for DAC interface core.


Wed, 02 Sep 2015 10:16:42 AM
M1: 17.230000 MHz 1.489 dBm
Tue, 01 Sep 2015 5:11:21 PM
M1: 28.380000 MHz -0.4194 dBm
Stop 30.72000 MHz
Res BW 200.0 kHz
VBW 200.0 kHz
Swp 104.0 ms (401 pts)

(a) DAC output of a 17.23 MHz tone (b) DAC output of a 28.38 MHz tone

FIGURE 22: The spectra different sinusoids generated using NCO core and measured at the FMC150 DAC output.

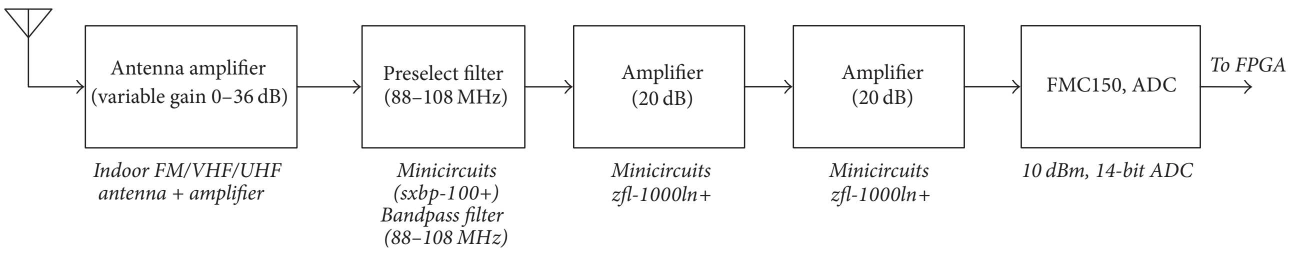

Antenna amplifier (variable gain 0–36 dB)
Preselect filter (88–108 MHz)
Amplifier (20 dB)
Amplifier (20 dB)
FMC150, ADC
To FPGA
Indoor FM/VHF/UHF antenna + amplifier
Minicircuits (sxbp-100+) Bandpass filter (88–108 MHz)
Minicircuits zfl-1000ln+
Minicircuits zfl-1000ln+
10 dBm, 14-bit ADC

FIGURE 23: A block diagram of the analog RF front-end.

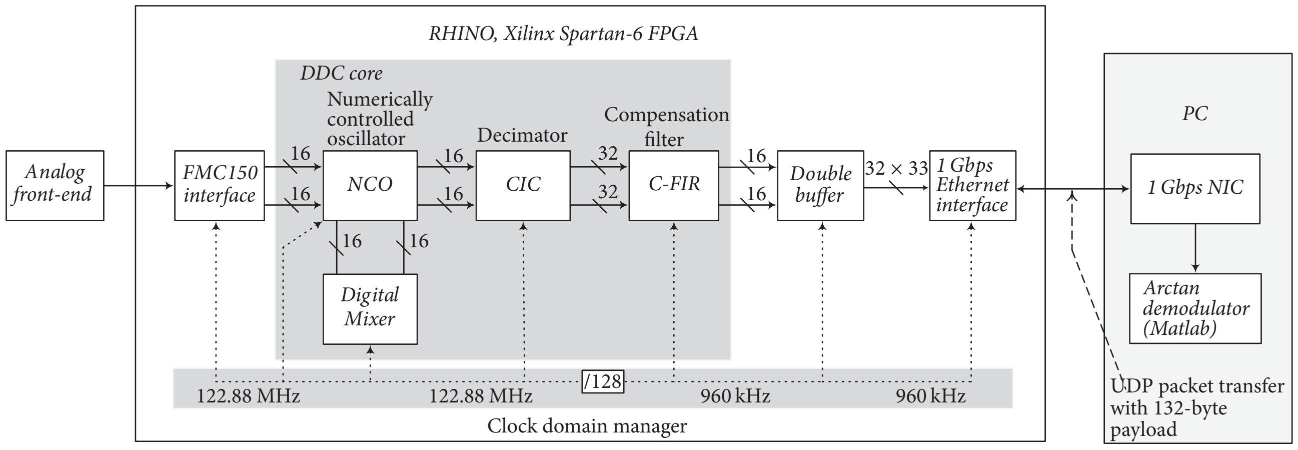


Figure 24: An architecture of the digital FM receiver.

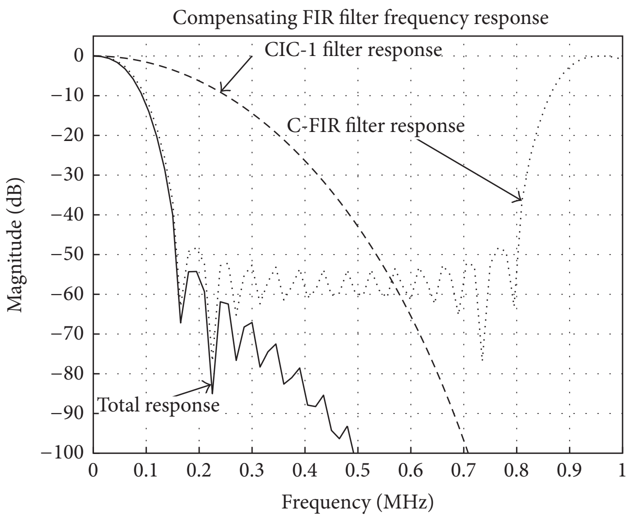


Figure 25: The total frequency response due to CIC and C-FIR frequency response. CIC parameters: *input sample = 122.88 MSPS, output sample rate = 960 kSPS, decimation factor (R) = 128, number of stages (N) = 10, differential delay (M) = 1.* C-FIR parameters: *input and output sample rate = 960 kSPS, number of coefficients = 21, cutoff frequency = 90 kHz, stopband attenuation = 10 dB.*

(3) The number of FFT core points is set to 1024 with data width set to 16 bits.

(4) The DDC core is configured to operate on 122.88 MSPS of an RF signal and output 1.28 MSPS of a baseband signal.

The Xilinx library does not have an IIR core while OpenCores does not have a DDC core needed to perform benchmarking.

The Xilinx DSP cores exhibit more performance and use the fewest FPGA resources overall. The static nature of the Xilinx cores allows easy access to generic parameters through a core generator wizard which makes it very complex to even modify the generated code. Our cores have performance that is slightly less than Xilinx cores but with comparable FPGA resource utilization. They also expose easy interface for configuration of generic parameters of the cores. The OpenCores cores have the lowest performance and the largest resource occupation on the FPGA. Furthermore, they have other constraints such as limiting the number of FIR core coefficients to less than 22 and the FFT points are limited to 1024.

The results have shown that our SDR cores achieve significant performance and use less resources while exposing design parameters to the user in the VHDL code. Further benchmark tests for a Gbe core, FMC150 interface core, and the FM receiver application are performed and results shown in Table 1 show a satisfactory performance needed for SDR.

9. Proposed DSL and Tool-Flow for SDR

We briefly introduce a new SDR high-level synthesis, namely, SdrHls which enables the FPGA design of SDR applications using a Domain-Specific Language (DSL). Instead of translating a DSL into new FPGA functionality, SdrHls maps user design specifications in DSL onto parameterizable

Table 1: IP core benchmark results for Xilinx, OpenCores, and SDR cores.

Source	Cores	Slices (23038)	%	LUTs (92152)	%	Registers (184304)	%	RAM (21680)	%	DSP48A1s (180)	%	BUFGs (16)	%	Maximum Clock Frequency (MHz)
Xilinx library	FIR	30	1	50	1	96	1	13	1	1	1	1	1	303
	FFT	885	3	2294	2	3403	1	607	2	16	8	1	1	141
	DDC	680	2	1223	1	2179	1	472	2	7	3	1	1	134
OpenCores	FIR	1556	6	3872	4	691	1	0	0	21	11	1	1	80
	IIR	247	1	857	1	864	1	0	0	72	40	1	1	66
	FFT	1209	5	2768	3	3120	1	1024	4	16	8	1	1	84
SDR cores	FIR	43	1	132	1	304	1	0	0	30	16	1	1	130
	IIR	144	1	376	1	492	1	0	0	36	20	1	1	94
	FFT	930	4	2518	2	1267	1	642	2	16	8	1	1	118
	DDC	1404	6	4024	4	5179	2	0	0	2	1	1	1	129
	Gbe	1205	5	2701	1	2928	1	411	1	0	0	7	43	165
	FMC150	631	2	1335	1	1272	1	142	1	0	0	8	50	184
	FM Rec.	1404	6	4024	4	5179	2	0	0	2	1	1	6	154

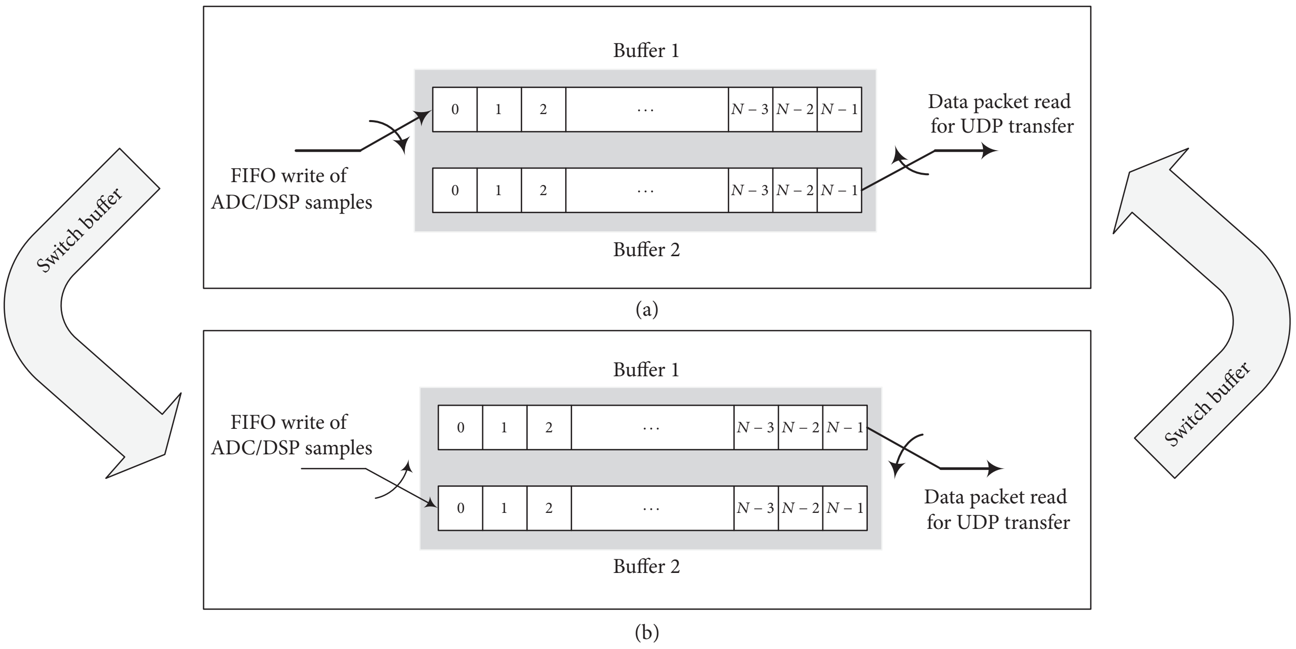

Buffer 1
0
1
2
⋯
N − 3
N − 2
N − 1
Buffer 2
FIFO write of ADC/DSP samples
Data packet read for UDP transfer
Switch buffer
(a)
(b)

Figure 26: Double buffering used between the ADC or DSP output and Gbe input. (a) Writing DSP samples to BUFFER 1 and creating a data packet by concurrent reading of BUFFER 2 samples. (b) Writing DSP samples to BUFFER 2 and creating a data packet by concurrent reading of BUFFER 1 samples.

Data field of UDP packet	Sample 0	Sample 1	Sample 2	⋯	Sample 30	Sample 31	Sample 32
Word length in bits	32	32	32	⋯	32	32	32

Sample field of UDP data field	In-phase data	Quadrature data
Sample length in bits	16	16

Figure 27: UDP frame data field format. Data field is composed of 33 samples collected from a DDC core output.

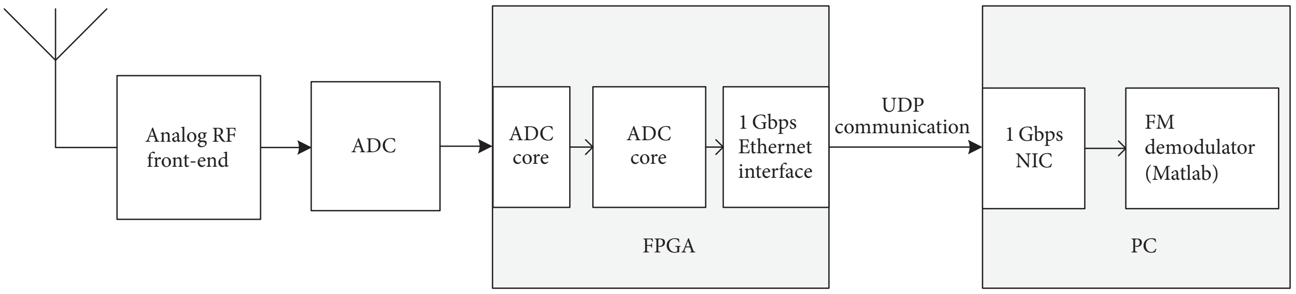


Figure 28: Experimental setup for a digital wideband FM receiver.

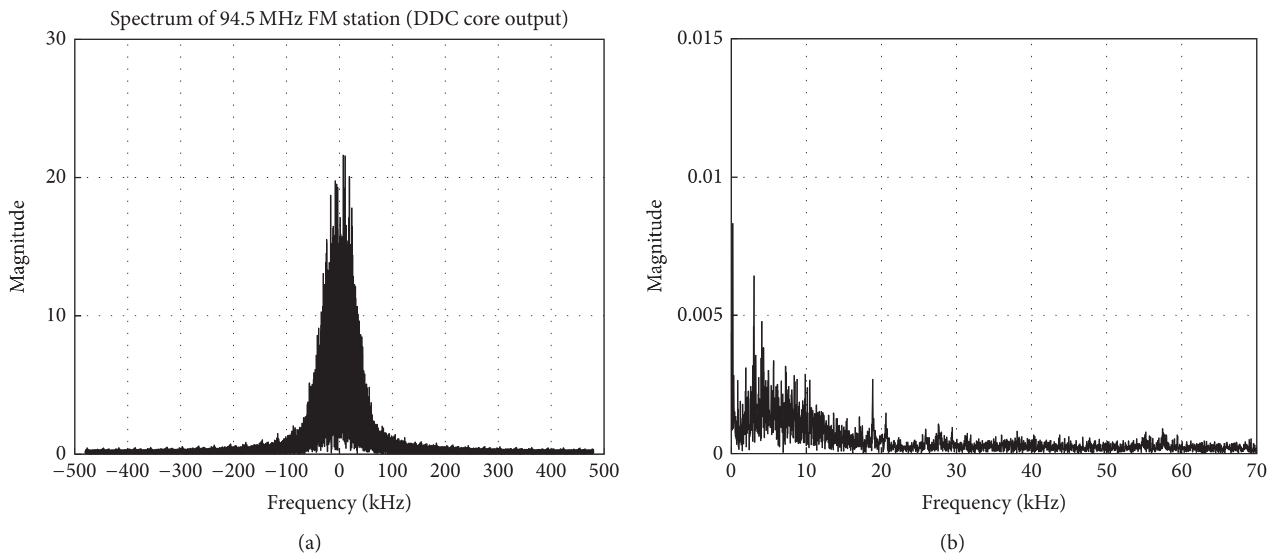


Figure 29: The results of FM receiver when tuning to 94.5 MHz radio station. (a) Spectrum of 94.5 MHz FM station baseband signal which is the output of a DDC core, (b) spectrum of FM-demodulated 94.5 MHz FM station in Matlab; the signal is real-valued after demodulation.

SDR IP cores and stitches the cores together to yield the desired FPGA design. It achieves this by adopting a model-based design approach employing a Synchronous Dataflow (SDF) [39] while also leveraging the high-level topological design patterns of a DSL that are based on a functional language called Scala. There are other dataflow-based HLS tools for fast prototyping of DSP applications such as Ptolemy [40], LabView [41], and Simulink [42]. These tools provide intuitive methodologies to specify, simulate, and/or execute DSP applications. However, they strive to provide generic solutions in all areas of DSP field thereby resulting in compromised efficiency of produced designs. SdrHls is a domain-specific tool optimized for producing efficient FPGA-based SDR designs and uses idioms and notations familiar to domain users of SDR.

An overview of our design flow for SdrHls is illustrated in Figure 30 and its description is backed by a typical SDR design example of the FM receiver discussed in Section 7.

9.1. A DSL and High-Level Compilation. We begin with an application specification that represents an SDR algorithm to be translated into an FPGA design in VHDL. This is specified using a DSL that employs a dataflow-based approach as shown Listing 1 and the corresponding SDF diagram is illustrated in Figure 31. The expressiveness and conciseness of the rich DSL syntax enable the intuitive description of a system, therefore, raising the low-level FPGA design abstraction to a higher level of design abstraction. This makes it easy for domain users with limited or no hardware design skills to generate hardware design and for skilled users to improve productivity.

Depending upon the system requirements, the user selects from the existing library of SDR cores the components needed to construct a complete system. The parameters for components can optionally be configured using a DSL and the SdrHls will assign default values for the unset parameters during high-level synthesis. Furthermore, the parameters set in a DSL are later mapped to VHDL generics in the final hardware design. Such parameters can be defined as static values in a DSL or read from data files stored in a local memory. Most importantly, a DSL allows parameter values to be dynamically generated using a DSL itself and the compiler will assign static values for the parameters. A typical example is generating the filter coefficients for a FIR core.

The first six lines in Listing 1 include the SdrHls DSL compiler library and the Delite library and define an object for running the main SdrHls main application method. The fourth line generates a list of coefficients for a compensating filter of the CIC filter in Figure 24. The compensator constructor takes in parameters set with values as follows: sample rate change = 128, a number of stages = 10, and differential delay = 1. This is followed by the configuration of the parameters which correspond to VHDL generics of the IP cores. In this example, the ADC core takes no parameters as denoted by Nil, and the DDC core parameters set include data width and filter coefficients with other parameters not shown to make the code brief. The Gbe is set with 64 bytes of transmitted payload which is calculated using (3). The *Component* object is used to define the IP core and it takes in the VHDL component name of the IP core together with its generic parameters. The *Chain* object is a topological design pattern that creates a cascade of component objects connected to each other using FIFO channels. The square brackets after each of the components define the consumption and production rates of each component or actor in an SDF dataflow. The rate of zero denotes nonexisting input or output channel while the existence of a channel is denoted by the rate greater than zero. Lastly,

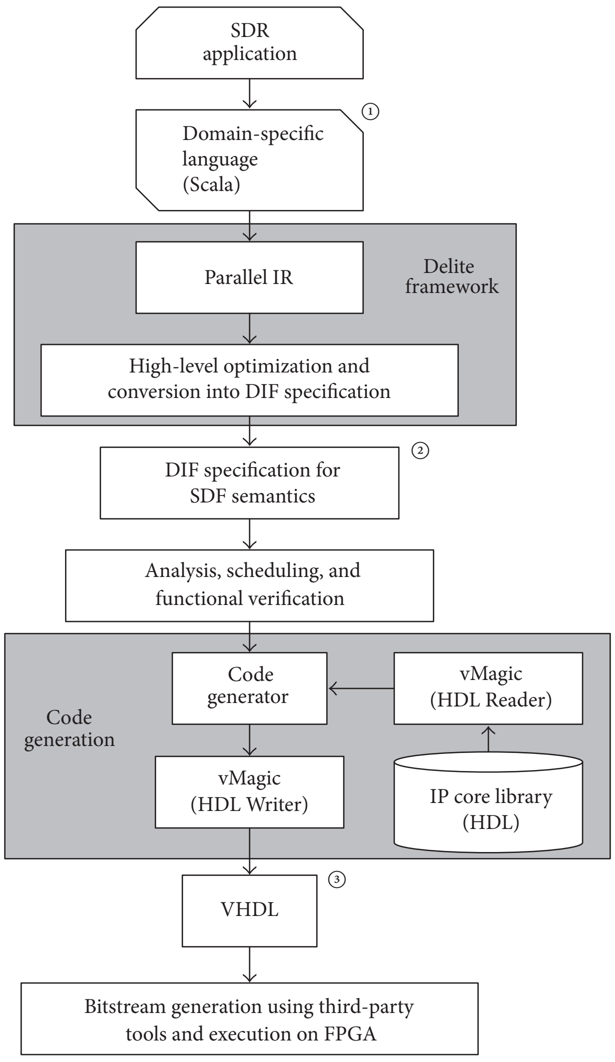

SDR application
Domain-specific language (Scala)
①
Parallel IR
Delite framework
High-level optimization and conversion into DIF specification
DIF specification for SDF semantics
②
Analysis, scheduling, and functional verification
Code generator
vMagic (HDL Reader)
Code generation
vMagic (HDL Writer)
IP core library (HDL)
VHDL
③
Bitstream generation using third-party tools and execution on FPGA

Figure 30: A high-level synthesis design flow for SDR.

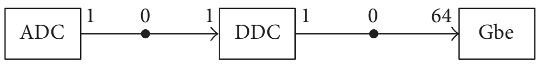


Figure 31: An SDF diagram of the FM receiver.

the VHDL code of the system is generated by a *synthesize* method.

$$\begin{aligned}\text{UDP payload size} &= \text{Gbe core consumption rate}\\ &\quad \times \text{baseband channels}\\ &\quad \times \text{data width} = 16 \times 2 \times 16\\ &= 512\,\text{bits} = 64\,\text{bytes}\end{aligned} \quad (3)$$

The DSL application (Scala code) is input into a Delite framework compiler framework [43] which runs a *scalac* to convert the DSL into Java bytecode. The produced Java bytecode is then executed (staged) to create Intermediate Representation (IR) and perform optimizations. The Delite IR comprises useful artefacts, namely, dependency graph and a sea of nodes representing DSL operations which are transformed into a dataflow model for the system. Delite is a highly extensible compiler framework and runtime environment developed by Stanford PPL. It provides developers with reusable components like parallel patterns, optimizations, and automatic code generators to facilitate construction of parallel embedded DSLs. The current support of resultant languages includes C++, CUDA, OpenCL, and Scala. Support for configuration of FPGAs and deployment of Delite-generated executables onto FPGA platforms for deployment onto Altera FPGA chips is being developed.

Recent breakthroughs in this work include defining a new FPGA intermediate representation language called Dataflow

```
// import SdrHls DSL compiler
import sdrg.dsl.sdrhls._
import sdrg.dsl.sdrhls.dsp.fir.Compensator
// include lms, delite libraries
object SdrHlsMainRunner extends SdrHlsApplicationRunner with SdrHlsMain
trait SdrHlsMain extends SdrHlsApplication {
     def main() {
          // generate a list of compensating filter coefficients
          val filter = Compensator(128, 10, 1)
          // set generic parameters for DDC
          valddcParams = ("DIN_WIDTH" -> 16, "DOUT_WIDTH" -> 16,
                    "COEFFS" -> filter.coefficients,
                    // other parameters
                      ...
                    )
          // set generic parameters for Gigabit Ethernet
          valgbeParams = ("TX_BYTES", 64)
          // define FM receiver components
          valadc = Component("fmc150", Nil)
          valddc = Component("DDC", ddcParams)
          valgbe = Component("UDP1GbE", gbeParams)
          // the chain of system components
          val radio = Chain(adc[0:1], ddc[1:1], gbe[16:0])
          // generate VHDL code for the FM Receiver
          radio.synthesize("FmReceiver")
     }
 }
```

Listing 1: SdrHls source code for the FM Receiver①.

Hardware Description Language (DHDL) and generating hardware code for Maxeler platform in MaxJ [44, 45]. However, DHDL only generates MaxJ for Maxeler platforms other than VHDL or Verilog which are used universally. DHDL is also targeted for applications in domains of machine learning, image processing, financial analytics, and internet search, all of which are not naturally related to SDR applications. In this work, we intend to improve productivity and performance of reconfigurable designs for SDR while also increasing portability using platform independent synthesizable VHDL.

9.2. System Modeling. The application model is specified using a Dataflow Interchange Format (DIF) Language [46] shown in Listing 2. A DIF specification formally captures the dataflow semantics of various dataflow models and performs analysis of topological information contained in a dataflow. We use SDF in our design for its straightforward static dataflow scheduling and analysability of throughput and buffer requirements. The DIF performs analysis, scheduling, and functional verification of the system modeled in SDF. When all these functions are complete, the application modeled in SDF is now ready for mapping onto a hardware.

9.3. Code Generation. This process involves generation of the VHDL code shown in Listing 3 which represents the design described using a DSL. The SDF dataflow-based design is converted into hardware description using SDR IP cores and FIFO buffers. The SDR IP cores become SDF actors and FIFO buffers act as SDF channels. We use VHDL Manipulation and Generation Interface (vMagic) [47] to read IP core library as well as writing the VHDL code of FPGA FM receiver system. The final step is the conversion of VHDL code into a bitstream using a third-party tool called Xilinx ISE. The bitstream is then loaded on the FPGA device for execution.

10. Conclusion

In this paper, we presented the design of a modular, reusable, and parameterizable library of SDR HDL cores. These cores provide both wishbone-compatible interfaces and direct parallel interfaces. DSP cores for processing and I/O cores for connecting to an ADC sampling daughterboard are provided together with an Ethernet data streaming core for sending data from the FPGA to a host computer. Functional validation testing was done for each core using a reconfigurable computing platform, namely, the RHINO platform (see Section 6.1), and the cores were tested working together in a representative SDR application, namely, an FM receiver shown in Section 7.4. In order to link the SDR processing cores to an input stream of sampled data, the FMC150-ADC core would need to be customized to be compatible with the sampling hardware. The test also demonstrated how the cores and their parameterizability allowed for rapid assembling of the SDR system. The SDR cores were benchmarked in

```
sdfFmReceiver{
    topology {
        nodes = adc, ddc, gbe;
        edges = channel1(adc, ddc),
            channel2(ddc, gbe);
    }
    parameter {
        gbeParams =[(tx_bytes => 64), ...];
        ddcParams =[(din_width => 16), ...];
    }
    production {
        channel1 = 1;
        channel2 = 1;
    }
    consumption {
        channel1 = 1;
        channel2 = 16;
    }
    delay {
        channel1 = 0;
        channel2 = 0;
    }
    actor adc{
        computation = "FMC150";
    }
    actor ddc{
        computation = "DDC";
        generics = ddcParams;
    }
    actor gbe{
        computation = "UDP1GbE";
        generics = gbeParams;
    }
}
```

Listing 2: DIF source code for the FM Receiver[2].

Section 8, confirming that these cores provided adequate performance that was not greatly less than that exhibited by the closed source Xilinx IP cores. While the SDR cores had greater resource utilization than the Xilinx cores, the utilization of a particular type of resource (be it the number of slices, LUTs, registers, etc.) was generally below 200% of those used by the Xilinx cores. The SDR cores provided a better performance in terms of maximum supported clock rate and generally used fewer resources than similar OpenCores cores. The SDR cores thus provide a speed-area trade-off that makes them an open-source alternative to costly commercial IP cores. The SDR cores also provide greater code-based parameterizability than the other cores benchmarked.

In order to facilitate FPGA-based SDR application development for programmers not experienced in HDL coding, and as a possible approach to enhance productivity and reduce the complexity and amount of low-level coding needed for SDR application development using our SDR cores, we have proposed a DSL and accompanying tool-flow, which we are in the process of building. This tool-flow builds upon the Delite DSL framework and our SDR core library. It uses a range of parameters and the automatic code generation of Delite, to raise the level of design abstraction and as a potential means to speed up development time. The proposed DSL aims to capture system specifications easily and to automate the modeling of design characteristics such as system throughput and memory size while also optimizing the system for reduced area and the increased speed of the resulting FPGA design.

While our DSL is still at an early stage, we are hoping that the SDR cores we have provided will be of use to other researchers and developers working in the area of FPGA-based SDR application development and that we may gain feedback from any users of our resources and tools which we can use to further enhance our cores and our proposed DSL and supporting tools to assist in reuse of these cores.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

```
-- FmReceiver library package
use work.SdrHlsPkg.all;
--FmReceiver top level entity
entity FmReceiver is
    port(
        ...
    );
end FmReceiver;
architecture rtl of FmReceiver is
    -- declare registers
    ...
begin
    -- instantiate adc component
    FMC150Inst: FMC150
    port map(···);
    -- instantiate adc to ddcfifo channel
    FMC150Inst_DDCInst_Channel1: sdf_channel
    generic map(
        DATA_BITS => 32, DEPTH => 2, PRD_RATE => 1, CNS_RATE => 1, INIT_DLY => 0)
    port map(···);
    -- instantiate ddc component
    DDInst: ddc
    generic map(
        DIN_WIDTH => 16, DOUT_WIDTH => 16,...)
    port map(···);
    -- instantiate ddc to Gbe fifo channel
    DDCInst_UDP1GbEIns_Channel2: sdf_channel
    generic map(
        DATA_BITS => 32, DEPTH => 18, PRD_RATE => 1, CNS_RATE => 16, INIT_DLY => 0)
    port map(···);
    -- instantiate Gigabit Ethernet
    UDP1GbEIns: UDP1GbE
    generic map(TX_BYTES => 64, RX_BYTES => 0)
    port map(···);
end rtl;
```

Listing 3: VHDL source code for the FM Receiver③.

Acknowledgments

The authors sincerely thank SKA for funding this project and members of the SDRG and RRSG groups at UCT for their support and advice.

References

[1] K. Tan, H. Liu, J. Zhang, Y. Zhang, J. Fang, and G. M. Voelker, "Sora: High-performance software radio using general-purpose multi-core processors," *Communications of the ACM*, vol. 54, no. 1, pp. 99–107, 2011.

[2] E. Blossom, "Gnu radio: tools for exploring the radio frequency spectrum," *Linux journal*, vol. 2004, no. 122, 4 pages, 2004.

[3] A. Haghighat, "A review on essentials and technical challenges of software defined radio," in *Proceedings of MILCOM 2002*, vol. 1, pp. 377–382, 2002.

[4] S. Winberg, A. Langman, and S. Scott, "The RHINO platform - charging towards innovation and skills development in Software Defined Radio," in *Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists (SAICSIT '11)*, pp. 334–337, October 2011.

[5] J. Gao, "10_100_1000 Mbps Tri-mode ethernet MAC specification," *OpenCores*, 2006.

[6] W. H. W. Tuttlebee, "Software-defined radio: Facets of a developing technology," *IEEE Personal Communications*, vol. 6, no. 2, pp. 38–44, 1999.

[7] J. Ghetie, "Fixed wireless and cellular mobile convergence: technologies, solutions, services," in *Proceedings of the 9th International Conference on Telecommunications (ConTel '07)*, 343 pages, 2007.

[8] C. E. Caicedo and P. D. Student, *Software defined radio and software radio technology: Concepts and application*, Department of Information Science and Telecommunications University of Pittsburgh, 2007.

[9] A. C. Tribble, "The software defined radio: fact and fiction," in *Proceedings of the IEEE Radio and Wireless Symposium (RWS '08)*, pp. 5–8, January 2008.

[10] T. J. Rouphael, *RF and digital signal processing for software-defined radio: a multi-standard multi-mode approach*, Newnes, 2009.

[11] O. Romain and B. Denby, “Prototype of a software-defined broadcast media indexing engine,” in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP ’07)*, pp. II813–II816, April 2007.

[12] S. J. Olivieri, J. Aarestad, L. H. Pollard, A. M. Wyglinski, C. Kief, and R. S. Erwin, “Modular FPGA-based software defined radio for CubeSats,” in *Proceedings of the IEEE International Conference on Communications (ICC ’12)*, pp. 3229–3233, June 2012.

[13] A. Azarian and M. Ahmadi, “Reconfigurable computing architecture: survey and introduction,” in *Proceedings of the 2nd IEEE International Conference on Computer Science and Information Technology (ICCSIT ’09)*, pp. 269–274, August 2009.

[14] R. Woods, J. McAllister, G. Lightboy, and Y. Yi, *FPGA-based Implementation of Complex Signal Processing Systems*, John Wiley and Sons, 2008.

[15] R. Saleh, S. Wilton, S. Mirabbasi et al., “System-on-chip: Reuse and integration,” *Proceedings of the IEEE*, vol. 94, no. 6, pp. 1050–1068, 2006.

[16] J. C. G. Pimentel and H. Le-Huy, “A vhdl library of ip cores for power drive and motion control applications,” in *Proceedings of the Electrical and Computer Engineering, Canadian Conference*, vol. 1, pp. 184–188, 2000.

[17] A. Parsons, D. Backer, C. Chang et al., “A new approach to radio astronomy signal processing: Packet switched, fpga-based, upgradeable, modular hardware and reusable, platform-independent signal processing libraries,” in *Proceedings of the 30th General Assembly of the International Union of Radio Science*, pp. 4–7, 2006.

[18] F. Fang, J. Hoe, M. Pueschel, S. Misra, C.-M. U. P. P. D. ELECTRICAL, and C. ENGINEERING., *Generation of Custom DSP Transform IP Cores: Case Study Walsh-Hadamard Transform*, Defense Technical Information Center, 2002.

[19] J. Gaisler, “A dual-use open-source vhdl ip library,” in *Proceedings of the MAPLD International Conference*, pp. 8–10, 2004.

[20] A. López-Parrado and J.-C. Valderrama-Cuervo, “OpenRISC-based system-on-chip for digital signal processing,” in *Proceedings of the 19th Symposium on Image, Signal Processing and Artificial Vision (STSIVA ’14)*, pp. 1–5, September 2014.

[21] W. T. Padgett and D. V. Anderson, “Fixed-point signal processing,” *Synthesis Lectures on Signal Processing*, vol. 9, pp. 1–129, 2009.

[22] U. Meyer-Baese and U. Meyer-Baese, *Digital signal processing with field programmable gate arrays*, vol. 65, Springer, 2007.

[23] mikroelectronika, “Chapter 2: Fir filters - digital filter design,” http://www.mikroe.com/chapters/view/72/chapter-2-fir-filters/.

[24] S. He and M. Torkelson, “A new approach to pipeline fft processor,” in *Proceedings of the 10th International on Parallel Processing Symposium (IPPS ’96)*, pp. 766–770, April 1996.

[25] A. Saeed, M. Elbably, G. Abdelfadeel, and M. I. Eladawy, “Efficient fpga implementation of fft/ifft processor,” in *Proceedings of the International Journal of Circuits*, vol. 3, pp. 103–110, 2009.

[26] I. LogiCORE, *fast fourier transform*, vol. 8, INTECH, 2012.

[27] S.-M. Tseng, J.-C. Yu, and Z.-H. Lin, “Software digital-down-converter design and optimization for dvb-t systems,” *ResearchGate*, pp. 57–61, 2012.

[28] 4DSP, “FMC150 User Manual,” 2013.

[29] N. Alachiotis, S. A. Berger, and A. Stamatakis, “Efficient PC-FPGA communication over Gigabit Ethernet,” in *Proceedings of the 10th IEEE International Conference on Computer and Information Technology, CIT-2010, 7th IEEE International Conference on Embedded Software and Systems, ICESS-2010 and 10th IEEE International Confernce Scalable Computing and Communications (ScalCom ’10)*, pp. 1727–1734, July 2010.

[30] I. LogiCORE, *Tri-mode ethernet mac v4. 5 user guide*, XILINX Inc, 2011.

[31] M. R. Mahmoodi, S. M. Sayedi, and B. Mahmoodi, “Reconfigurable hardware implementation of gigabit UDP/IP stack based on spartan-6 FPGA,” in *Proceedings of the 6th International Conference on Information Technology and Electrical Engineering (ICITEE ’14)*, October 2014.

[32] N. Alachiotis, S. A. Berger, and A. Stamatakis, “A versatile udp/ip based pc-fpga communication platform,” in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig ’12)*, pp. 1–6, December 2012.

[33] M. Inggs, G. Inggs, A. Langman, and S. Scott, “Growing horns: applying the Rhino software defined radio system to radar,” in *Proceedings of the 2011 IEEE Radar Conference: In the Eye of the Storm (RadarCon ’11)*, pp. 951–955, May 2011.

[34] H. K.-H. So, A. Tkachenko, and R. Brodersen, “A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH,” in *Proceedings of the 4th International Conference on Hardware Software Codesign and System Synthesis (CODES+ISSS ’06)*, pp. 259–264, October 2006.

[35] S. Scott, *Rhino: Reconfigurable hardware interface for computation and radio [M.S. thesis]*, University Of Cape Town, 2011.

[36] B. K. Huang, R. G. L. Vann, S. Freethy et al., “FPGA-based embedded Linux technology in fusion: The MAST microwave imaging system,” *Fusion Engineering and Design*, vol. 87, no. 12, pp. 2106–2111, 2012.

[37] R. G. Vaughan, N. L. Scott, and D. R. White, “The theory of bandpass sampling,” *IEEE Transactions on Signal Processing*, vol. 39, no. 9, pp. 1973–1984, 1991.

[38] Altera, “Understanding CIC Compensation Filters,” apn455, 2007.

[39] D. G. Messerschmitt, “Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing Edward Ashford Lee,” *IEEE Transactions on Computers*, vol. C-36, no. 1, pp. 24–35, 1987.

[40] J. Eker, J. W. Janneck, E. A. Lee et al., “Taming heterogeneity—the ptolemy approach,” *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–143, 2003.

[41] H. A. Andrade and S. Kovner, “Software synthesis from dataflow models for g and labview,” in *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, pp. 1705–1709, 1998.

[42] T. E. Dwan and T. E. Bechert, “Introducing simulink into a systems engineering curriculum,” in *Proceedings of the 23rd Annual Conference on Frontiers in Education: Engineering Education: Renewing America’s Technology*, pp. 627–631, November 1993.

[43] K. J. Brown, A. K. Sujeeth, H. J. Lee et al., “A heterogeneous parallel framework for domain-specific languages,” in *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (PACT ’11)*, pp. 89–100, October 2011.

[44] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun, “Automatic generation of efficient accelerators for reconfigurable hardware,” in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA ’16)*, pp. 115–127, June 2016.

[45] R. Prabhakar, D. Koeplinger, K. J. Brown et al., "Generating configurable hardware from parallel patterns," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*, pp. 651–665, ACM, New York, NY, USA, April 2016.

[46] C. Hsu, F. Keceli, M. Ko, S. Shahparnia, and S. S. Bhattacharyya, "DIF: an interchange format for dataflow-based design tools," in *Computer Systems: Architectures, Modeling, and Simulation*, vol. 3133 of *Lecture Notes in Computer Science*, pp. 423–432, Springer, Berlin, Germany, 2004.

[47] C. Pohl, C. Paiz, and M. Porrmann, "vmagic-automatic code generation for vhdl," *International Journal of Reconfigurable Computing*, vol. 2009, pp. 1–9, 2009.