Received 9 September 2021; revised 30 November 2021 and 28 January 2022; accepted 14 February 2022. Date of publication 17 February 2022; date of current version 8 March 2022.

Digital Object Identifier 10.1109/OJCAS.2022.3152399

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

JOSH GOLDSMITH (Graduate Student Member, IEEE), LOUISE H. CROCKETT, AND ROBERT W. STEWART

Department of Electronic and Electrical Engineering, University of Strathclyde, Glasgow, G1 1XW, U.K.

This article was recommended by Associate Editor C. Studer.

CORRESPONDING AUTHOR: J. GOLDSMITH (e-mail: joshua.goldsmith@strath.ac.uk)

ABSTRACT We present a natively fixed-point filter design method that targets FPGA-based Reconfigurable Finite Impulse Response (RFIR) filters for Software Defined Radio applications. The Filter Designer is capable of reconfiguring cut-off frequencies on-the-fly at run-time; with other parameters, such as filter length and window type, configurable at compile-time. The ability to compute filter coefficients directly on FPGAs is compelling, as much lower latencies can be achieved when compared to RFIRs programmed with embedded processors. In this work we discuss several filter design techniques from the literature and investigate their suitability for implementation on FPGAs. A hybrid method combining window and frequency sampling methods is developed and implemented on a Xilinx Zynq-7000 SoC. We explore the limitations of designing filters in fixed-point arithmetic and consider the effects filter length and wordlength have on filter quality. Results show that the proposed algorithm generates good-quality filters that display stopband attenuation up to 88dB, transition bandwidths less than 1% of the sample rate, and low resource utilisation. Most notably, we found that our method is up to three orders of magnitude faster than an equivalent software implementation, with execution times as low as 2.52 μs, enabling radio applications in which latency is a principal constraint.

INDEX TERMS Digital filter design, FPGAs, Circuits and systems for software-defined radio.

I. INTRODUCTION

RECONFIGURABLE Finite Impulse Response (RFIR) filters are an increasingly important component in modern Software Defined Radio (SDR) architectures, where devices are required to support multiple wireless communication standards, while making efficient use of hardware resources and available RF spectrum. Field Programmable Gate Arrays (FPGAs) are well suited for these applications due to their inherent reprogrammability and parallel structure.

A typical SDR system can require multiple FIR filters for both transmit and receive paths. The ability for some, or all, to be reconfigurable at run-time allows for a more dynamic system. For instance, low-pass anti-alias filters may be needed for decimation and interpolation stages, while bandpass filters can help to avoid interference between adjacent channels. If the SDR is required to support multiple bands or standards, the use of RFIRs can help reduce both hardware resources and design complexity when compared to systems that employ separate filters for each band, or standard, they support.

RFIR filters are an active research topic, with many targeting FPGA-based hardware. For example, in [1] the authors developed an RFIR filter architecture based on run-time configurable look-up tables (LUTs). The filter can be reconfigured with arbitrary coefficients, with wordlength and filter length fixed at compile-time. In [2] an RFIR filter is implemented using Wallace tree multipliers, resulting in an increased maximum operating frequency, but to the detriment of hardware resource usage. Other work focuses on keeping resources and power consumption low. In [3] and [4] an RFIR filter is designed using Vedic multipliers, which reduces arithmetic operations to shifts and adds. In this architecture, which primarily targets Application Specific Integrated Circuits (ASICs), the filter length is readily changed at run-time by switching off unused gates, reducing

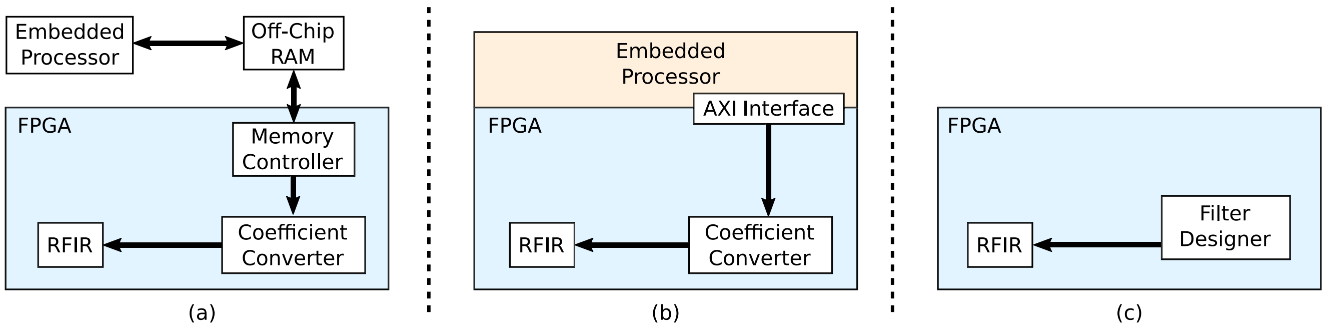

FIGURE 1. Block diagram depicting three methods of programming an FPGA-based RFIR. In (a) an off-chip embedded processor calculates the filter coefficients and passes them to the FPGA via shared memory. In (b) an integrated embedded processor calculates the filter coefficients and passes them to the FPGA via memory-mapped registers. In (c) the coefficients are calculated directly on the FPGA.

power usage. In [5] an RFIR filter is created based on coefficient occurrence probability, where coefficients are represented as canonic signed digits and converted on hardware as a way to reduce hardware resources. In [6] two architectures are proposed using constant and programmable shifts methods, both of which produce a reduction in power usage and resource utilisation.

It must be noted, however, that the work in [1]–[6] focuses primarily on the architecture of RFIR filters, with little attention given to the reconfiguration method employed or how the fixed-point coefficients are calculated. Typically, the use of pre-calculated, memory-stored coefficient lists is assumed. In the remainder of this section we establish a clear motivation for taking a radically different approach in which the filter coefficients are dynamically calculated on the target device instead. Additionally, we discuss applications for which this method is advantageous and highlight the research contributions of the work presented in this paper.

A. MOTIVATION

Pre-calculated coefficient lists for FPGA-based RFIRs would typically be stored using either distributed memory, in the form of LUTs, or block random access memory (BRAM). However, this method is not scalable for systems that require high filter orders and fine-control over parameters. For example, in the Xilinx 7-series FPGAs, BRAMs are either 18 or 36 Kb, dependent on configuration [7]. Therefore, to store even a modest configuration of 100, 16-bit, 101-length filters would require at least five BRAMs. Moreover, if arbitrary bandpass filter responses are required, especially in applications that require high sample rates and large bandwidths, storage requirements could easily become a limiting factor. This signifies that, in some conditions, calculating new coefficients at run-time is more appropriate.

Both off-chip Digital Signal Processing (DSP) and embedded processors are obvious choices for designing filter coefficients on-the-fly at run-time, but require some form of communication between processor and FPGA for the RFIR to be reconfigured. This communication layer results in inherent latencies, while off-chip, memory-based storage and retrieval can be a bottleneck. Additionally, when low latency and high-order filters are required, especially in systems where the embedded processor is responsible for other time-sensitive computations, it may be difficult for smaller devices to meet these constraints.

Considering these limitations it is worth exploring the potential for on-the-fly filter design directly on the FPGA instead. In Fig. 1 we compare the architectural differences between this approach and two other methods of reconfiguring FPGA-based RFIRs at run-time.

In Fig. 1a an embedded processor calculates the filter coefficients and stores them in shared memory. The coefficients are then retrieved by a memory controller on the FPGA before being converted and passed to the RFIR. Fig. 1b depicts an architecture where the embedded processor shares the same silicon as the FPGA, forming a System-on-Chip (SoC). In this method coefficients are calculated on the processor and passed to the FPGA via shared memory-mapped registers. In devices such as Xilinx Zynq SoCs this would be performed by either AXI (Advanced eXtensible Interface) or AXI-Lite interfaces [8]. Finally, in Fig. 1c, filter coefficients are calculated and passed to the RFIR directly on the FPGA.

When comparing the method depicted in Fig. 1a to Fig. 1c the benefits of the latter design are clear. It removes the need for off-chip communication, is less complex in terms of software and hardware requirements, and is likely to use less power. The benefits of the direct FPGA approach over the method depicted in Fig. 1b, however, are less apparent as communication between FPGA and processor remains on-chip. With that said, even without the requirement of off-chip communication there are still potential bottlenecks in this method worth considering. AXI-Lite is limited to single transactions, which could lead to slow read/write times for long length filters. Furthermore, while AXI burst transactions do provide higher throughput, in this work we show that the process of calculating filter coefficients on embedded processors is also a significant source of latency.

Another method is also possible where the FPGA is controlled by software, such as MATLAB, running on an external computer. In this method, the coefficients can be calculated at high speed and passed to the FPGA via an

Ethernet or serial connection. However, this method is cumbersome, requiring a reasonably powered computer to run the software, and would be more suited to a prototyping stage rather than incorporation into an embedded design.

B. APPLICATIONS

The design of filters directly on FPGAs reveals some interesting possibilities for applications in SDR, particularly on devices such as the Xilinx Radio Frequency System-on-Chip (RFSoC), where RF-speed data converters are connected directly to the programmable logic [9]. On these devices, multi-gigahertz bandwidths and large oversampling rates allow for better frequency planning to avoid interference from spurious signals [10]. One possible scenario would be where a tight filter is required around a band of interest but, due to the current device configuration, an interfering spur would be present. This interference can be mitigated by adjusting the sample rate of the system on-the-fly, but would require the filter to be redesigned to match the new configuration.

Interference may also be present from external sources such as jamming signals or transmissions from unlicensed users. The ability to quickly redesign filters and reconfigure the system to avoid these interfering signals would help reduce the amount of down-time experienced on the network.

In scenarios like this, on systems where many frequency bands are supported over gigahertz of spectrum, we can see the benefits of on-the-fly filter design over pre-calculated coefficients. Moreover, the ability to redesign the filter directly on the FPGA not only allows the process to remain on-chip, but the reconfiguration process can also be achieved in significantly less time than an off-chip, or embedded, solution.

These fast reconfiguration times enable radio applications where low latency is a principal constraint such as the automotive, aviation, and mobile communication sectors. For example, the user plane latency requirements of broadband cellular networks have been steadily decreasing between generations, with 6G proposed to be as low as 0.1 ms [11]. With this continuing trend, communication technologies where spectrum allocation is more flexible and which benefit from a high degree of reconfigurability, such as cognitive radio and dynamic spectrum access, will require the low latency that can be provided by FPGA-based filter design.

The design of FIR filter coefficients directly on FPGA hardware, however, is non-trivial. Fixed-point arithmetic can cause cumulative quantisation errors, while optimal design methods—such as the Parks-McClellan algorithm—can result in non-deterministic behaviour. These types of issues are well documented in the works of Kodek [12]–[15], among others, where limitations of finite wordlength filter design were a consequence of the DSP hardware restrictions of the time. The absence of FPGA-specific, fixed-point filter design algorithms from the literature provides further motivation to investigate the practicality of this approach.

C. RESEARCH CONTRIBUTIONS

This work proposes an FPGA-based filter design algorithm in which fixed-point filter coefficients are computed on-the-fly at run-time. For testing purposes we target the FIR Compiler v7.2 Intellectual Property (IP) core on a Xilinx Zynq-7000 SoC, using the AXI-Stream protocol for filter reconfiguration. However, our design attempts to be as universal as possible and could be readily modified to operate on other FPGAs and FPGA-based RFIRs.

Our proposed design uses a hybrid of frequency sampling and window filter design methods, calculated entirely in fixed-point arithmetic. The Filter Designer allows for the cut-off frequency to be reconfigurable at run-time, with other parameters such as filter length and window type configurable at compile-time. The filters produced by this method are shown to be deterministic and of good quality, with stopband performance and transition bandwidths well suited for SDR applications. By performing filter computation directly on the FPGA instead of an embedded processor, we are able to greatly reduce the latencies incurred by calculation and transfer of the coefficients, with our method demonstrating execution times up to three orders of magnitude faster than the software equivalents tested.

In summary, the contributions of this work are as follows.

- An FPGA-based Filter Designer capable of generating deterministic, fixed-point, linear-phase filter coefficients for RFIR filters; displaying both low resource utilisation and frequency responses suitable for SDR applications.
- A filter design and reconfiguration algorithm that is capable of execution times as short as 2.52 μs, suitable for applications with very low latency requirements.
- A comprehensive analysis of the filters generated by this method, which may act as a design guide for users.
- To the best of the authors' knowledge, the only natively fixed-point filter design method developed specifically for FPGA hardware.

The rest of this paper is organised as follows. In Section II we discuss several FIR filter design techniques, explore their suitability for FPGA implementation, and detail the algorithm used within this work. Section III details how the algorithm was translated to a Hardware Description Language (HDL) IP and integrated within a larger SoC design. In Section IV we analyse the quality of filter coefficients produced by the algorithm, provide details of FPGA resource utilisation and execution times, and compare these results to a processor-based approach. Finally, the paper is concluded in Section V.

II. FILTER DESIGN TECHNIQUES

FIR filters have two useful properties which make them a practical choice for SDR applications: they do not require recursion, making them inherently stable, and they can be easily constrained to achieve linear phase. An FIR filter has linear phase if its impulse response displays either even or odd symmetry, or even or odd anti-symmetry around its middle point. This results in four linear phase filter types,

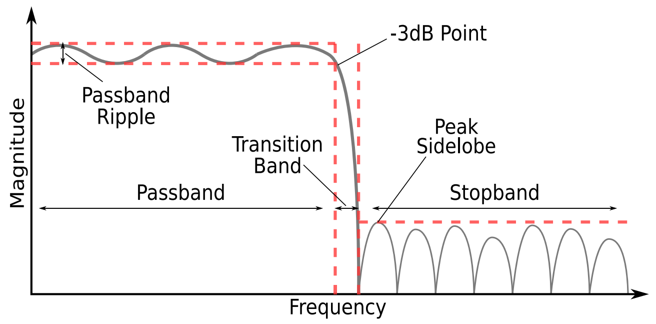

FIGURE 2. Key components of a filter frequency response.

usually denoted I-IV [16], [17]. Type I is the most universal and, for the purposes of this work, only low-pass Type I filters will be considered.

The ideal frequency response of an FIR filter is denoted as $H_d(\omega)$, where $0 < \omega \leq 2\pi$, and can be written in the form

$$H_d(\omega) = \sum_{n=-\infty}^{\infty} h_d(n)e^{-j\omega n}, \tag{1}$$

which is an infinite series and non-causal, and therefore not realisable. The actual filter, $H(\omega)$, must then be truncated to a length, N, and shifted such that $n \geq 0$, which then becomes

$$H(\omega) = \sum_{n=0}^{N-1} h(n)e^{-j\omega n}, \tag{2}$$

meaning that the ideal filter can only ever be approximated by the actual filter [16]. This approximation problem causes undesirable effects in the frequency response such as ripple in the passband, a decrease in stopband attenuation, and an increase in transition bandwidth, as shown in Fig. 2.

Various filter design techniques have been developed over the years which attempt to minimise the effects caused by approximation to within acceptable levels for a given application, while keeping N small enough to be viable for implementation on hardware. These factors are even more significant when considering filter design techniques viable for FPGAs, where arithmetic is generally limited to fixed-point, and wordlengths may need to be restricted to prevent unlimited bit growth. These restrictions can cause quantisation errors to accumulate (e.g., product roundoff, overflow) therefore care must be taken to minimise these effects by allowing reasonable bit growth where possible, and deciding in what manner values should be quantised.

In the case of filter design, these quantisation errors will move $H(\omega)$ further from $H_d(\omega)$, which will result in poorer performance in the stopband—though these errors can be somewhat reduced by various forms of rounding [18]–[21]. Moreover, an increase in filter length will result in a larger overall error, which then must be mitigated by increasing the wordlength. All these factors must be taken into account when considering an appropriate filter design technique for FPGAs.

The goal of this work then is to find a filter design technique that can satisfy three conditions.

1) It can produce low-pass filters with deterministic frequency responses, programmable at run-time.
2) It is suitable for implementation on FPGA hardware.
3) The execution time is deterministic at compile-time (i.e., at IP generation).

This section considers both optimal and sub-optimal filter design techniques, which we use to determine a suitable algorithm that can both minimise the above mentioned effects and satisfy all three conditions.

A. OPTIMAL ALGORITHMS

Optimal design techniques, such as the Parks-McClellan and Least Squares (LS) algorithms [16], look at filter design as an optimisation problem, where the error between the desired response, $H_d(\omega)$, and the actual response, $H(\omega)$, is minimised to find an optimal fit.

These methods can produce high quality filters and are the most versatile. However, the algorithms required to compute the filters are inherently unsuited to FPGA implementation, so will only be briefly stated here.

In the case of the Parks-McClellan algorithm, an iterative approach is taken where the filter is recalculated until a solution is found. The iterative nature of the algorithm can result in long convergence times or, in some cases, no convergence is possible. This issue of convergence can be overcome by restricting the number of iterations, or introducing less strict parameters. However, this can result in low-quality filters and only allows there to be an upper limit of execution time, rather than being absolutely deterministic, thus failing Condition 3. Furthermore, the Parks-McClellan algorithm is not computable with wordlength restrictions [13] and, although it is possible to reduce the effects of quantisation errors when converting to fixed-point, it is still necessary to compute the original algorithm first.

The LS algorithm relies on a linear algebra pseudo-inverse calculation that is inherently difficult to achieve on FPGAs due to the division operation, which is both costly and slow. Moreover, the size of the matrices required for long length filters can also be a limiting factor, failing Condition 2 [22].

It has also been shown that optimal finite wordlength filters can be computed using integer programming [13], simulated annealing [23], and genetic algorithms [24] but, again, these methods rely on optimisation techniques that fail to satisfy Condition 3.

B. SUB-OPTIMAL ALGORITHMS

Two of the most common methods of sub-optimal filter design are the frequency sampling and window methods. Both approach the problem of filter design from the same perspective, where an ideal response is truncated to result in a realisable form. These methods typically have a reduced set of parameters to be tuned, and require longer length filters to achieve comparable quality to the optimal methods.

TABLE 1. Characteristics of various window functions, where ω_S is the sampling frequency in radians/sample. Replicated from [17].

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

As neither of these sub-optimal methods use recursive algorithms, they do not suffer from the same problems as the optimisation techniques. Therefore, it is worth looking at the window and frequency sampling methods more closely to determine their suitability for FPGA implementation.

1) THE WINDOW METHOD

In the case of the window method an ideal impulse response, $h_d(n)$, is multiplied by a window function, $w(n)$, of desired filter length, N, to create a realisable filter, $h(n)$,

$$h(n) = w(n).h_d(n). \tag{3}$$

In the case of a low-pass filter, $h_d(n)$ is the sinc function and can be written in the form

$$h_d(n) = \frac{\sin(\omega_c n)}{\pi n}, \tag{4}$$

where ω_c is the cut-off frequency in radians/sample.

The simplest case is to just truncate (4) to N samples—equivalent to multiplying by the rectangular window. However, this typically results in poor quality filters, where the abrupt discontinuities cause excessive passband ripple and low stopband attenuation (i.e., Gibbs Phenomenon [25]), regardless of filter length. The most popular windows used in this technique are the generalised cosine windows, such as the Hamming, Hanning, and Blackman. These windows display a smooth curve that decays towards zero more gradually which, when multiplied by $h_d(n)$, helps to both reduce passband ripple and increase stopband attenuation [16].

Each window has its own well-defined characteristics, where the trade-off is usually between transition bandwidth (BW) and stopband attenuation. The characteristics of the four windows mentioned in this section are tabulated in Table 1.

As most modern FPGAs contain an abundance of dedicated DSP slices (containing both multipliers and accumulators) [26], multiplication can be considered a trivial operation to implement. Moreover, if we constrain the filter length to be fixed at compile-time, the number of cycles required to complete the multiplication will also be fixed, therefore satisfying Condition 3.

The window characteristics in Table 1 allow us to predict the effects these windows will have on the filter at compile-time, which indicates that this method is able to satisfy Condition 1 by producing consistent quality filters. However, if we consider Condition 1 where the parameter ω_c must be tunable at run-time, an interesting problem arises. In the case of (3), the length of $w(n)$ is fixed and does not include the ω_c term, therefore its coefficients can be readily stored in memory. However, in the case of (4), $h_d(n)$ requires a division operation and, due to the ω_c term being present, it is not possible to compute and store these values at compile-time. While it is possible to implement division operations on FPGAs, it is costly in both resources and clock cycles, causing it to fail Condition 2.

2) THE FREQUENCY SAMPLING METHOD

In the case of the frequency sampling method the ideal frequency response, $H_d(\omega)$, is sampled at equally spaced frequencies, $\omega_k = \frac{2\pi k}{N}$, where $k = 0, 1, \ldots, N-1$, and can be written in the form

$$H_d(\omega_k) = \sum_{n=0}^{N-1} h_d(n) e^{-j2\pi kn/N}, \tag{5}$$

which equates to the Discrete Fourier Transform (DFT). Therefore the time-domain coefficients can be retrieved by computing the inverse transform [27].

However, because the sampled, ideal response from (5) is only defined at the sample points, the spaces between the points are effectively interpolated. This becomes an issue in areas of the response that change amplitude abruptly, most notably at the transition band, which can result in undesirable filter properties [16]. This can be compensated for by the use of transition coefficients, as described in [28], but the algorithm to calculate these coefficients requires a minimax optimisation technique that would be difficult to implement on an FPGA, failing to satisfy Condition 2.

Furthermore, for long length filters the DFT is impractical and thus the FFT algorithm must be used. This then raises another problem where the filter length would generally be limited to power-of-2 values, both severely restricting the flexibility of the filter designer, and eliminating the possibility of odd-symmetry Type I filters. It should be noted that algorithms do exist that can compute non-power-of-2 FFTs, but these are typically less efficient (i.e., slower) and can increase design complexity. That being said, it has been shown that under certain conditions non-power-of-2 FFTs can result in a reduction in hardware resources and power consumption [29].

C. A HYBRID APPROACH

In practice, the frequency sampling and window methods can be combined to produce filters with frequency responses suitable for most applications. In this work we define this combination of the two techniques a *hybrid* approach, and use it to create a filter design algorithm that can satisfy all three conditions, allowing it to be implemented on FPGA hardware.

Initially, the desired frequency response is sampled at equally spaced frequencies of length N_{FFT}, where $\log_2(N_{FFT})$ is an integer value, and is symmetric about $N_{FFT}/2$ in order to retain phase linearity. The inverse FFT

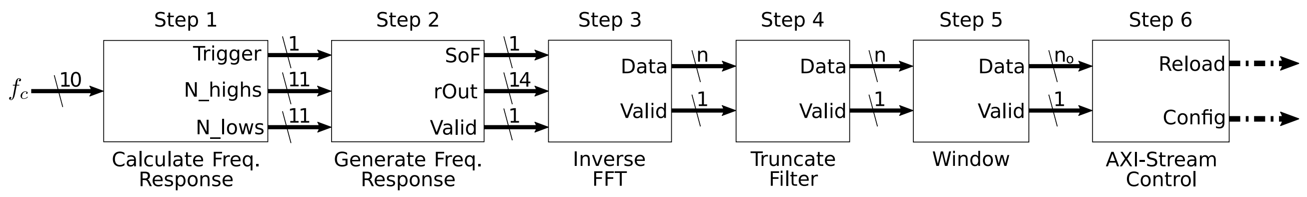

FIGURE 3. Simplified block diagram of the Filter Designer model developed in System Generator [30]. Note that *n* is the number of bits dependent on system configuration, and n_O is the number of bits constrained at compile-time.

is then performed to retrieve the N_{FFT}-length time-domain coefficients, which are then truncated to N values around the centre coefficient. Finally, the filter is then multiplied by an N-length window function. In this work, a Blackman window is used due to its high stopband attenuation.

Due to the use of windowing, as we have seen, filters of consistent quality can be produced, satisfying Condition 1. Furthermore, as we now use the frequency-domain representation of the sinc function (i.e., the rectangular function), this is readily computable at run-time where, for a low pass filter, it can be written in the form

$$\mathrm{rect}\left(\frac{\omega_k}{2\omega_c}\right) = \begin{cases} 0 & \omega_c/2 < \omega_k < \omega_s - \omega_c/2, \\ 1 & \text{otherwise.} \end{cases} \tag{6}$$

The only other operations required are the inverse FFT and the multiplication of the window—also readily computable on FPGAs—allowing this method to now satisfy Condition 2. Finally, if we constrain the filter length, N, at compile-time, Condition 3 can also be satisfied.

Although the filter length is fixed in our proposed algorithm, it is worth noting that constraining N at compile-time is not always necessary to satisfy Condition 3. In some cases, the potential for N to be reconfigurable at run-time may be desirable for certain RFIR filter architectures. For the execution time to be deterministic the number of samples must be a fixed size at compile-time. However, the actual coefficients representing the filter can be of any length (less than the maximum), and the remaining samples can be padded with zero values. For example, in the work of [4], as described in Section I, zero-valued coefficients will result in the gates representing them to be switched off, lowering the overall operating power. This has some obvious benefits for systems in which power is a principal constraint, where the ability to switch between higher and lower order filters, where appropriate, could help to reduce power consumption.

III. HARDWARE DESIGN

This section describes how the overall hardware design was developed for the Zynq device. First we describe how the Filter Designer algorithm was translated to an HDL IP using the System Generator for DSP software [30]. We then go on to explain how the HDL IP was integrated into a larger test environment on the PL and PS, using both the Vivado IDE [31] and PYNQ framework [32].

The entire codebase developed in this work, including all hardware IP, has been released as an open-source project [33], where the interested reader may view or download the source code. Furthermore, the repository that hosts the project also includes additional implementation documentation which may aid in reproducibility.

A. IP DESIGN

The filter design algorithm described in Section II-C can be divided into six steps.

1) Calculate desired frequency response.
2) Generate frequency response.
3) Perform inverse FFT.
4) Truncate the filter to desired length.
5) Perform window operation.
6) Load coefficients to the RFIR filter.

Fig. 3 depicts the algorithm as a simplified, system-level block diagram, which was developed in System Generator. The rest of this section describes each component in detail.

1) CALCULATING THE DESIRED FREQUENCY RESPONSE

In Step 1, in order to calculate the frequency response, two values are required: the FFT length, N_{FFT}, and the normalised cut-off frequency, f_c. As f_c is the cut-off frequency in relation to the sampling period, and N_{FFT} is effectively the sampling period of the filter at this stage of the algorithm, the number of high values (i.e., the 1 values from (6)), N_{highs}, can be calculated by

$$N_{highs} = \|f_c.N_{FFT}\|, \tag{7}$$

where $\|\|$ denotes a rounding operation.

Since $h(n)$ is purely real the frequency response must be symmetrical around $N_{FFT}/2$, meaning the total number of high values must be $2f_c.N_{FFT} - 1$. From this the number of low values (i.e., the 0 values from (6)), N_{lows}, can be calculated as

$$N_{lows} = N_{FFT} - \big(2.N_{highs} - 1\big). \tag{8}$$

In this design the value of N_{FFT} is fixed at compile time and the value of f_c is taken as a user (or system) input to the IP. As the value of N_{FFT} is always a power of two, both (7) and (8) can be implemented by a single multiply operation and a series of shifts and adds, requiring a total of three clock cycles to complete.

2) GENERATING THE DESIRED FREQUENCY RESPONSE

In Step 2 the frequency response is generated using a Finite State Machine (FSM). The FSM takes N_{highs}, N_{lows}, and a trigger as inputs, and outputs the AXI-Stream signals required by the FFT IP [34]. The state machine adds an additional clock cycle to the total latency of the system.

3) INVERSE FFT

Step 3 performs the inverse FFT that is required to transform the frequency response into the filter impulse response. This is performed by the LogiCORE Fast Fourier Transform v9.1 IP block available in System Generator. The IP inputs follow the AXI-Stream protocol and requires *tdata*, *tvalid*, and *tlast* signals. The *tlast* signal is pulsed-high for one sample and signifies the end of a data block which, in this case, is of size N_{FFT}. Note that this IP is the principal source of latency within the algorithm and is proportional to the size of N_{FFT}.

4) TRUNCATING TO AN N-TAP FILTER

In Step 4 the FFT frame output is written to a single-port RAM, implemented using BRAM. As the FFT frame is output by the IP in bit reversed order, a ROM containing the bit reversed position values is used to write each FFT sample to its correct RAM address. This ensures the desired N-length filter can be readily retrieved by the next stage in the correct order. For Type I symmetric filters only half the N-point filter is required to reload the FIR Compiler IP [35], $M = (N + 1)/2$, around the centre tap. An FSM is used to read these M values from the RAM in the correct order, introducing a delay of $N_{FFT} + 3$ clock cycles.

5) WINDOWING THE FILTER

Step 5 multiplies the M-length filter with a window function. This is performed by multiplying the output of the RAM with an M-length window (i.e., half the window function), stored in a ROM, and read out using a counter. This introduces a two cycle delay. The desired output wordlength is set after this operation.

6) RELOADING THE FILTER COEFFICIENTS

Step 6 transfers the coefficients to the RFIR filter which, in this work, is the AXI-Stream compliant LogiCORE FIR Compiler v7.2 IP from Xilinx. The FIR Compiler IP uses two AXI-Stream slave ports to reload the filter coefficients, named *Reload* and *Config*. The *Reload* port receives the coefficients within the *tdata* signal, along with the required *tvalid* and *tlast* signals. The *Config* port is then used to instruct the IP to load the new filter by sending an empty 8-bit packet along with a single-cycle *tvalid* signal [35]. An FSM is used to perform this reconfiguration handshake, introducing a one cycle delay.

It should be noted that, although this step is specific to the FIR Compiler IP, the algorithm can be readily modified to accommodate other RFIR filter architectures.

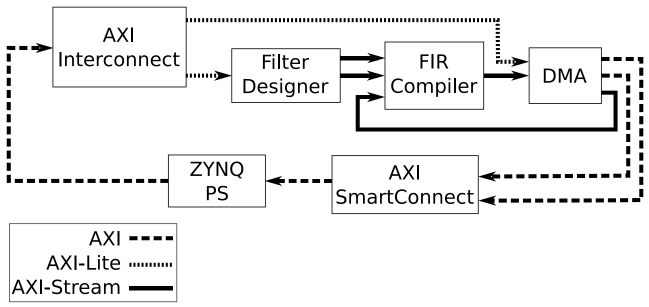

FIGURE 4. Simplified Vivado IPI block diagram for the test environment.

B. SYSTEM DESIGN

The Filter Designer was developed for the PYNQ-Z2 development board that features the Xilinx Zynq XC7Z020 SoC, containing both FPGA Programmable Logic (PL) and Processing System (PS) on a single chip. The PL is an equivalent Artix-7 FPGA, consisting of Configurable Logic Blocks (CLBs), DSP slices, BRAMs, and routing logic. The PS consists of a 667 MHz dual-core Arm Cortex-A9 processor, peripheral and memory interfaces, and clock generation—capable of running a Linux operating system. Communication between the PL and PS is facilitated by multiple AXI communication interfaces, providing both memory-mapped and high-speed streaming data transfer [36], [37].

This section describes the design of the PL and PS components of the system, and how the Filter Designer IP was integrated into a larger test design for the Zynq SoC. The hardware and software systems presented in this section are readily portable to other Zynq-based devices, such as the MPSoC and RFSoC platforms—both suited to SDR applications.

1) PL DESIGN

The PL design was developed in the Vivado Design Suite IP Integrator (IPI) software tool, building around the Filter Designer IP that was created in System Generator.

The Filter Designer's two AXI-Stream master ports (*Reload* and *Config*) are connected to the corresponding slave ports of the FIR Compiler IP. The memory-mapped port, used to control the cut-off frequency, is connected to a standard AXI Interconnect IP. Test signals are transferred between the FIR Compiler and the PS via the off-chip memory, facilitated by an AXI Direct Memory Access (DMA) controller IP.

The Zynq PS IP is configured with a general purpose AXI master port and a high performance AXI slave port, while a single PL fabric clock is used for all IPs. Two frequencies were implemented during testing: 100 MHz and 250 MHz. A simplified diagram of the Vivado IPI block design is shown in Fig. 4.

2) PS DESIGN

As the Filter Designer has only one input, the cut-off frequency, controlled from a shared AXI-Lite register, very little software is required to test its output. However, to adequately test the IP in a full system, as depicted in Fig. 4, the software requirements become more complex. For this reason the software developed for this design was written within the PYNQ framework.

PYNQ is an open-source embedded software framework from Xilinx that targets their line of Zynq and Zynq Ultrascale+ SoCs. The framework includes an Ubuntu-based Linux operating system, a software library written in the Python programming language, and a Jupyter Web server that allows uers to run embedded code in a Web browser-based Integrated Development Environment (IDE) [32], [38].

One immediate benefit of using PYNQ is the speed in which embedded code can be written and tested. In this work, to confirm that the Filter Designer functioned as expected, a cut-off frequency is written to a memory-mapped AXI-Lite register to generate filter coefficients and program the FIR filter IP. A test signal is then generated, written to memory, then sent to the FIR filter's data port by a DMA transaction. The filtered signal is then written back into memory and read back in the PS by a second DMA transaction. This entire program can be achieved in many fewer lines of code using the PYNQ framework, than with a conventional C implementation.

Another benefit of PYNQ is the ability to integrate standard Python modules into the codebase alongside embedded code written using the PYNQ library. An example of this is the plotting library *matplotlib*, which allowed us to quickly analyse the results of the test signals passed through the filter visually, in both time and frequency domains.

The PYNQ framework was also used for the results in Section IV-F, where execution times were compared between fixed-point filters designed on the PL and the PS. To differentiate between filters designed on the PL, calculated entirely in fixed-point, and those designed on the PS, calculated in floating-point then converted to fixed-point, we refer to them as *native* and *non-native* respectively.

IV. RESULTS

In this section we look at various filters designed by the system, consider the quality of the coefficients and the effects of quantisation, make comparisons between native and non-native fixed-point filters, and compare execution time between software and hardware implementations.

For the purposes of this work stopband attenuation is measured at the peak of the highest sidelobe, while the transition bandwidth is measured between the −3dB point and the end of the main lobe (see Fig. 2). To show the data more clearly, stopband attenuation measurements in this section are displayed as a second-order polynomial fit, which shows the general trend of the data, and at least some deviation from the trendline should be expected.

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

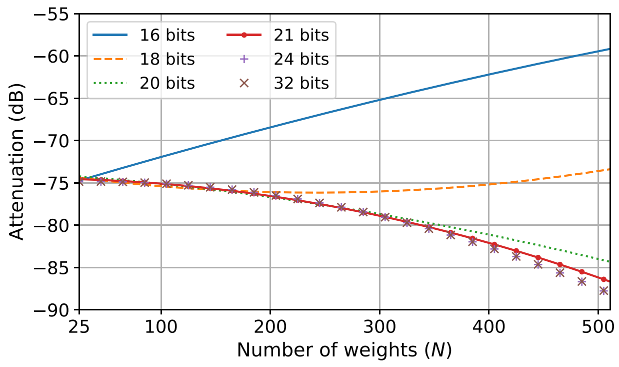

FIGURE 5. Trendlines of stopband attenuation over N with respect to wordlength constraint. $N_{FFT} = 1024$, $f_c = 0.33$.

A. EFFECTS OF WORDLENGTH CONSTRAINTS ON FILTER QUALITY

Throughout the signal path of the Filter Designer IP the wordlengths are left unconstrained to reduce quantisation errors, except at the output where the wordlength is constrained at compile-time. With N_{FFT} fixed at 1024, f_c fixed at 0.33, and with varying values of N, stopband attenuation and transition bandwidth were measured over a range of constrained output wordlengths.

As shown in Fig. 5, at shorter wordlengths the stopband attenuation decreases as N increases, caused by cumulative quantisation errors. These errors become less significant as the wordlength increases, with stopband attenuation improving over higher filter orders, and with no improvement shown past 24 bits. There is approximately 30dB difference between highest and lowest stopband attenuation measurements.

It was found that output wordlength had little effect on transition bandwidth, with all wordlengths displaying the same downward trend as N increases, which is typical for the window function used in the design.

B. EFFECTS OF FFT LENGTH ON FILTER QUALITY

The Filter Designer allows for the FFT size and filter length to be selected at compile-time. Results were obtained to measure how stopband attenuation and transition bandwidth changed with regards to the ratio between these two factors, N/N_{FFT}. It was discovered during development that filter quality degraded once this ratio passed 50% (this is also documented in [28]), therefore only results below this ratio are displayed. Additionally, a fixed wordlength of 24-bits was used as this produced the best performance in the stopband.

As shown in Fig. 6, stopband attenuation displays a similar downward trend as the ratio between N and N_{FFT} increases. From this result alone, one would assume that a shorter FFT length would be more beneficial as this reduces both hardware resources and the amount of latency in the FFT calculation, allowing increased stopband attenuation

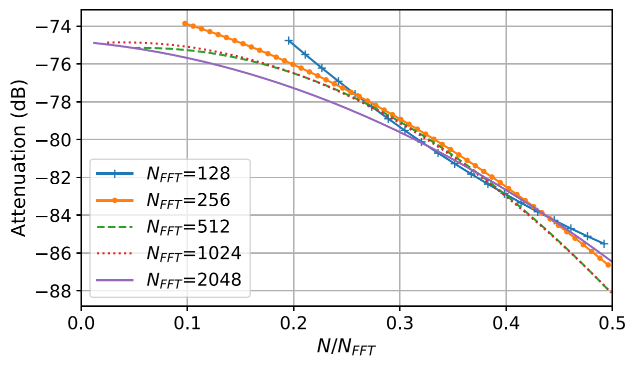

FIGURE 6. Trendlines of stopband attenuation over N/N_{FFT} with respect to N_{FFT}. $f_c = 0.33$, wordlength = 24.

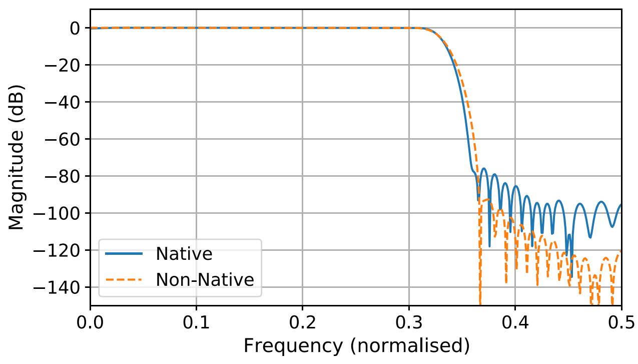

FIGURE 9. Comparison between a native and non-native fixed-point filter. $N = 101$, $N_{FFT} = 1024$, $f_c = 0.33$, wordlength = 24.

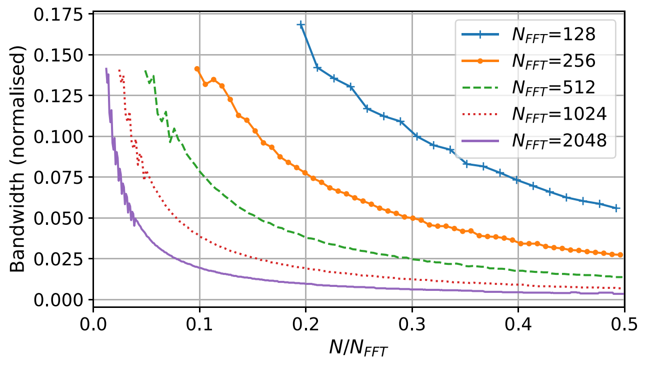

FIGURE 7. Transition bandwidth over N/N_{FFT} with respect to N_{FFT}. $f_c = 0.33$, wordlength = 24.

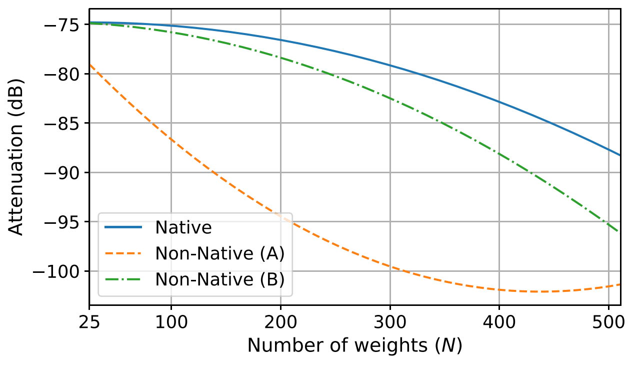

FIGURE 10. Trendline comparison of stopband attenuation over N between native filters and two variants of non-native fixed-point filters. $N_{FFT} = 1024$, $f_c = 0.33$, wordlength = 24.

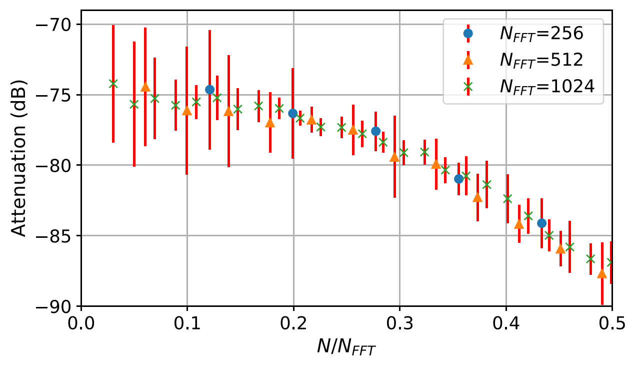

FIGURE 8. Mean and standard deviation of stopband attenuation over N/N_{FFT} with respect to N_{FFT}. Each data point uses 25 values of f_c to calculate the mean. Wordlength = 24.

with a relatively smaller value of N. However, as Fig. 7 shows, longer FFT lengths are required to reach transition bandwidths less than 1% of f_s.

Although data from both Figs. 6 and 7 were measured with a fixed value of f_c, it is worth noting that stopband attenuation is not constant for every f_c value. With this is mind, a number of measurements were taken to understand what effect changes in f_c had on filter quality.

In Fig. 8, 25 values of f_c were taken for each value of N, between 1% and 50% of f_s, and the mean and standard deviation were calculated. These results show that there is much higher deviation from the mean at shorter filter lengths, further implying that a longer FFT is beneficial to keep the frequency response of the filter consistent.

C. COMPARISON BETWEEN NATIVE AND NON-NATIVE COEFFICIENTS

When considering the quality of natively designed filters, it is worth comparing them to filters designed by non-native means. To make this comparison the MATLAB function `fir2` was used to design the non-native filters, as it employs an algorithm similar to the work in this paper, albeit using floating point arithmetic. For all the filters designed in this section, N_{FFT} and f_c were fixed at 1024 and 0.33 respectively, with the floating point filters converted to 24-bits after computation.

Fig. 9 shows a comparison between a native and non-native filter, where $N = 101$. This shows that, while the non-native filter displays much higher stopband attenuation, the native filter displays better performance in the transition band. Moreover, when we compare stopband attenuation and transition bandwidth over a number of filter lengths, we can see this trend continue. As shown in Fig. 10, as N increases the non-native filters (shown as Non-Native (A)) display continued improvement of stopband attenuation over the native equivalents, with over 20dB difference at its maximum. Similarly, although not to the same extent, the native filters display continued improvement of transition bandwidth

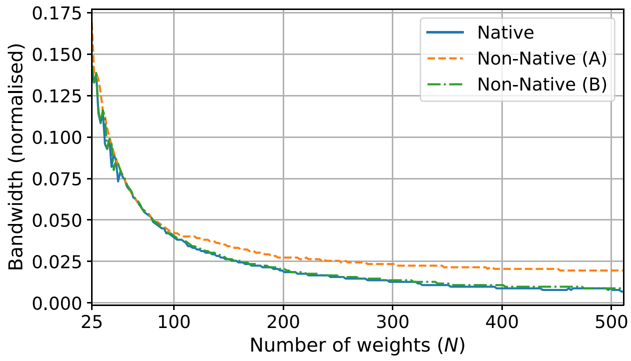

FIGURE 11. Comparison of transition bandwidth over N between native filters and two variants of non-native fixed-point filters. $N_{FFT} = 1024$, $f_c = 0.33$, wordlength = 24.

TABLE 2. FPGA utilisation for varying values of N and N_{FFT}. Wordlength = 24 bits.

N	N/N_{FFT}	LUTs	BRAM	DSPs
		N_{FFT}=256		
25	0.1	1779 (3.34%)	3 (2.14%)	11 (5.0%)
51	0.2	1778 (3.34%)	3 (2.14%)	11 (5.0%)
101	0.3	1782 (3.35%)	3 (2.14%)	11 (5.0%)
		N_{FFT}=512		
51	0.1	2179 (4.10%)	3 (2.14%)	14 (6.36%)
153	0.2	2184 (4.10%)	3 (2.14%)	14 (6.36%)
205	0.3	2183 (4.10%)	3 (2.14%)	14 (6.36%)
		N_{FFT}=1024		
101	0.1	2607 (4.90%)	3.5 (2.50%)	14 (6.36%)
307	0.2	2607 (4.90%)	3.5 (2.50%)	14 (6.36%)
409	0.3	2610 (4.91%)	3.5 (2.50%)	14 (6.36%)

over the non-native filters for $N > 100$, with over 1% of f_s difference at its maximum, as shown in Fig. 11.

The native filters display better performance in the transition band due to an implementation difference between the work in this paper and `fir2`, where the latter allows for more control over this region [39]. However, changing the `fir2` parameters to allow the non-native filters to match the transition bandwidth performance of the native filters (shown as Non-Native (B)) results in a considerable drop in stopband performance. As can be seen in Fig. 10, the difference in stopband attenuation between native and non-native filters drops from more than 20dB at its maximum, to less than 9dB. This demonstrates that, under certain conditions, natively designed filters can achieve a quality approaching that of non-native filters.

D. EFFECTS OF N AND FFT LENGTH ON FPGA RESOURCES

Here we examine how changes in N and N_{FFT} affect IP hardware resources on the Zynq-7020—a mid-range device within the Zynq family of devices. Three FFT lengths were used: 256, 512, and 1024. For each value of N_{FFT}, three filter lengths were used: 10%, 20%, and 30% of N_{FFT} (to the nearest odd value). The wordlength was constrained at 24-bits.

As can be seen from Table 2, the value of N has little impact on resources except for a small increase in LUTs.

TABLE 3. Measured execution time of filter designer, in clock cycles and absolute time, over two clock frequencies, with varying values of N and N_{FFT}. Wordlength = 24 bits.

N	N/N_{FFT}	Clock Cycles	$t(\mu s)$ @ 100 MHz	$t(\mu s)$ @ 250 MHz
		N_{FFT}=256		
25	0.1	629	6.29	2.52
51	0.2	643	6.43	2.57
101	0.3	668	6.68	2.67
		N_{FFT}=512		
51	0.1	1164	11.64	4.66
153	0.2	1215	12.15	4.86
205	0.3	1241	12.41	5.96
		N_{FFT}=1024		
101	0.1	2218	22.18	8.87
307	0.2	2322	23.22	9.29
409	0.3	2373	23.73	9.49

This is due to the filter length remaining equal to N_{FFT} for most of the signal path, and only being truncated before the window operation.

Each increase in N_{FFT} uses approximately 400 additional LUTs. There is a small increase in DSPs and BRAMs caused by the single-port RAM requiring more resources, as well as the FFT calculation itself. Overall this resource usage is low, using as little as 3.34% of LUTs, 2.14% of BRAMs, and 5% of DSPs. This leaves ample space for additional DSP algorithms, even for the relatively modest resources available on the XC7020 device.

E. EFFECTS OF N AND FFT LENGTH ON EXECUTION TIME

Using the same hardware implementation as the previous section, the execution time of the Filter Designer was measured to determine how changes in N and N_{FFT} affected the system. A simple counter was used to record the number of clock cycles between when the value of f_c was written to a register and the rising edge of the `m_axis_config_tvalid` signal—signifying that the filter had been reloaded. Two clock frequencies were tested: 100 MHz and 250 MHz.

As shown in Table 3, there is a near linear increase of execution time as the values of N and N_{FFT} increase. This result is expected as the latency incurred in each step of the algorithm is directly proportional to the length of the filter, both before and after truncation. The reason why there is not an exact doubling in cycles between changes in N_{FFT} is due to optimisation in the FFT implementation, which allows it to moderately reduce the number of calculations required [34]. These results demonstrate markedly short execution times when compared to software implementations, discussed in the following section.

F. NATIVE AND NON-NATIVE EXECUTION TIME COMPARISON

In order to make meaningful comparisons of execution time between native and non-native filter design methods, we assume that the system setup requires the FIR Compiler to be reconfigured from the PS side of the Zynq SoC.

For the Filter Designer IP, the f_c value is written to an AXI-Lite register that is shared between the PS and PL. For the comparison case, the filter coefficients are calculated in software then sent to the RFIR filter on the PL via off-chip memory. The necessary AXI-Stream handshake required by the FIR Compiler, explained in Section III, is managed by the use of two AXI DMAs on the PL: one for the *Reload* channel, and one for the *Config* channel. PYNQ is the assumed target framework.

The following operations are required to create a filter and reprogram the RFIR on the PL using the method described above:

1) Calculate N filter coefficients for given parameters.
2) Convert coefficients to integer values.
3) Copy coefficients to the *Reload* DMA buffer.
4) Initiate *Reload* DMA transfer.
5) Copy configuration packet to the *Config* DMA buffer.
6) Initiate *Config* DMA transfer.

These operations were split into two functions: `design_filter` and `reload_filter`. The former performs items 1 and 2, while the latter takes the quantised filter coefficients and performs the DMA transactions in items 3 to 6. Two values of N were used for these results: 51 and 101.

Because the transfer of data between PS and PL portions of the Zynq SoC can be viewed as a bottleneck, it is important to take a cautious approach when comparing native and non-native execution times on the device. This includes making comparisons between execution times on the PS and PL, as well as the system as a whole, all of which is discussed in the following sections.

1) MEASURING DMA TRANSACTION TIME

Two AXI DMAs are required to perform the AXI-Stream handshake necessary to reprogram the FIR filter. The PYNQ DMA class allows users to interact with DMAs and pass data between the PS and PL via shared memory. This is done by creating a contiguous memory array, copying data to the array, initiating the DMA transfer, then waiting for the transfer to complete. As two DMA transfers are needed to complete this process—one to transfer the coefficients (the *Reload* channel) and the other to instruct the filter to load these coefficients (the *Config* channel)—this wait time between DMA transactions can be a substantial source of latency.

The time taken to perform the operations within the `reload_filter` function was measured using Python's `timeit` module. This module computes the assigned code over a large number of iterations and outputs an average of the best three execution times. Using this technique, it was found that it took an average of 1.91 ms for $N = 51$, and 1.97 ms for $N = 101$.

To confirm this result in hardware a small counter IP was created on the PL. The counter recorded the number of clock cycles between the rising edges of the *Reload* DMA and *Config* DMA *tvalid* signals, indicating that both DMA transactions had completed.

TABLE 4. Hardware measured execution time of software filter design. Wordlength = 16 bits, clock = 100 MHz, sample size = 20.

N	Mean (Cycles)	Mean (ms)	Best (ms)	Worst (ms)
	w/o `design_filter()`			
51	685,768	6.86	3.38	17.63
101	740,443	7.40	3.39	65.60
	w/ `design_filter()`			
51	990,137	9.90	6.87	39.02
101	2,037,494	20.37	7.16	148.67

The execution time was shown to be non-deterministic and, over 20 measurements, a mean value of 685,768 cycles for $N = 51$ and 740,443 cycles for $N = 101$ was recorded. These results, along with a 'best' and 'worst' recorded time, are shown in the upper half of Table 4. The results recorded on hardware are approximately twice those measured using the `timeit` function, suggesting the latter stops timing the process before the second DMA transaction has completed.

These non-deterministic execution times are due to the overhead of the Linux operating system running on the PS, which gives no guarantees on when sequential commands are executed. This overhead could be mitigated by moving to a real-time operating system, or removing the operating system entirely from the PS (i.e., using a 'baremetal' approach), but would likely increase development time.

The results presented in this section solely took into account the execution time of the DMA transactions (i.e., the `reload_filter` function). In the next section we discuss the execution time to calculate the filter coefficients in software (i.e., the `design_filter` function).

2) MEASURING SOFTWARE FILTER DESIGN TIME

The Python function `firwin2` (from SciPy's Signal library) was used to calculate the filter coefficients within `design_filter`. This function, like MATLAB's `fir2` used in the simulation results, employs an algorithm similar to the work in this paper [40].

By using `timeit` on `design_filter` it was found that it took an average of 2.72 ms for $N = 51$ and 2.51 ms for $N = 101$. Similar results were confirmed in hardware, shown in the lower half of Table 4.

This apparent discrepancy between execution times is again due to the overhead of the Linux operating system. However, a basic implementation of the window method was created using the C language and run as a baremetal program on the SoC Arm processor, where similar order of magnitude results were obtained. For $N = 51$ and $N = 101$ a best time was measured to be 2.64 ms and 9.907 ms respectively—up to 1000 times slower than the FPGA implementation shown in Table 3. Moreover, as the baremetal results do not include RFIR reconfiguration time, the total execution time would be even longer than stated here. A comparison between

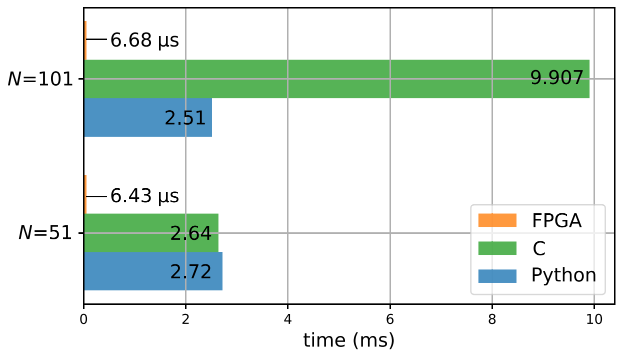

FIGURE 12. **Comparison between software and hardware execution times of the filter design algorithms (software times do not include RFIR reconfiguration time). Note that the *FPGA* times are taken from Table 3, where N_{FFT} = 256, with a system clock rate of 100 MHz.**

the execution times of the FPGA implementation and those discussed in this section is shown in Fig. 12.

V. CONCLUSION

This paper has presented an FPGA-based fixed-point filter design method that targets reconfigurable FIR filters. We investigated various filter techniques from the literature and developed an algorithm based on a hybrid between the window and frequency sampling methods, suitable for implementation on FPGAs. We focused our work on low-pass filters exclusively, but the algorithm could be readily modified to include high pass, bandpass, and bandstop filters. Although we placed this work in the context of software defined radio, it is not limited to these architectures alone. Any DSP design that includes run-time reconfigurable filters, and requires very low latency, could leverage this algorithm.

The algorithm is simple to implement and requires only four parameters to be set by the user. The filter length, FFT length, and output wordlength are set at compile-time, while the cut-off frequency can be changed on-the-fly at run-time. The cut-off frequency is controlled by writing to a shared PS/PL AXI-Lite memory address, but could be readily modified to be set solely on the FPGA. We have provided a comprehensive analysis of the effects these parameters have on filter quality and hardware resources, that can be used as a reference.

The implementation uses modest hardware resources which allows it to fit easily within a larger radio architecture. Our results show that the proposed algorithm generates deterministic filters of good quality and are well suited for SDR applications, with stopband attenuation as high as 88dB and transition bandwidths as low as 0.7% of f_s. Moreover, the algorithm demonstrates very low and deterministic latency, with execution times as low as 2.52 µs—orders of magnitude lower than its software equivalent.

Finally, the work in this paper has been released as an open-source project, under a permissive license, where the interested reader may view and use our work [33].

ACKNOWLEDGMENT

The authors would like to thank the Xilinx University Program for their donation of hardware and software licenses for this project.

REFERENCES

[1] M. Kumm, K. Möller, and P. Zipf, "Dynamically reconfigurable FIR filter architectures with fast reconfiguration," in *Proc. 8th Int. Workshop Reconfigurable Commun. Centric Syst. Chip (ReCoSoC)*, 2013, pp. 1–8.
[2] A. Jawahar and P. P. Latha, "Implementation of high-order FIR digital filtering for software defined radio receivers," in *Proc. Int. Conf. Signal Process. Commun. Power Embedded Syst. (SCOPES)*, 2016, pp. 1452–1456.
[3] N. V. Menon and S. Agrawal, "A speed efficient FIR filter for reconfigurable applications," in *Proc. IEEE Int. Conf. Comput. Intell. Comput. Res. (ICCIC)*, 2017, pp. 1–5.
[4] B. R. S. Rao and B. B. T. Sundari, "An efficient reconfigurable FIR filter for dynamic filter order variation," in *Proc. Int. Conf. Commun. Electron. Syst. (ICCES)*, 2019, pp. 1724–1728.
[5] R. Jia, H.-G. Yang, C. Y. Lin, R. Chen, X.-G. Wang, and Z.-H. Guo, "A computationally efficient reconfigurable FIR filter architecture based on coefficient occurrence probability," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 8, pp. 1297–1308, Aug. 2016.
[6] R. Mahesh and A. P. Vinod, "New reconfigurable architectures for implementing FIR filters with low complexity," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 29, no. 2, pp. 275–288, Feb. 2010.
[7] "7 series FPGAs data sheet: Overview," Xilinx Inc., San Jose, CA, USA, Rep. DS180, Sep. 2020.
[8] "AXI reference guide," Xilinx Inc., San Jose, CA, USA, Rep. UG761, Nov. 2012.
[9] "Zynq ultrascale+ RFSoC data sheet: Overview," Xilinx Inc., San Jose, CA, USA, Rep. DS889, Sep. 2020.
[10] "Understanding key parameters for RF-sampling data converters," Xilinx Inc., San Jose, CA, USA, Rep. WP509, Feb. 2019.
[11] W. Jiang, B. Han, M. A. Habibi, and H. D. Schotten, "The road towards 6G: A comprehensive survey," *IEEE Open J. Commun. Soc.*, vol. 2, pp. 334–366, Feb. 2021, doi: 10.1109/OJCOMS.2021.3057679. [Online]. Available: https://ieeexplore.ieee.org/document/9349624
[12] D. M. Kodek, "Limits of finite wordlength FIR digital filter design," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, vol. 3, 1997, pp. 2149–2152.
[13] D. Kodek, "Design of optimal finite wordlength FIR digital filters using integer programming techniques," *IEEE Trans. Acoust. Speech Signal Process.*, vol. 28, no. 3, pp. 304–308, Jun. 1980.
[14] D. Kodek and K. Steiglitz, "Filter-length word-length tradeoffs in FIR digital filter design," *IEEE Trans. Acoust. Speech Signal Process.*, vol. 28, no. 6, pp. 739–744, Dec. 1980.
[15] D. Kodek and K. Steiglitz, "Comparison of optimal and local search methods for designing finite wordlength FIR digital filters," *IEEE Trans. Circuits Syst.*, vol. CS-28, no. 1, pp. 28–32, Jan. 1981.
[16] T. Parks and C. Burrus, *Digital Filter Design* (Topics in Digital Signal Processing). New York, NY, USA: Wiley, 1987.
[17] L. Jackson, *Digital Filters and Signal Processing*. Dordrecht, The Netherlands: Kluwer Academic Publ., 1986.
[18] M. Gerken, "On fixed-point quantization schemes," in *Proc. 38th Midwest Symp. Circuits Syst.*, vol. 1, 1995, pp. 350–353.
[19] M. Haseyama and D. Matsuura, "A filter coefficient quantization method with genetic algorithm, including simulated annealing," *IEEE Signal Process. Lett.*, vol. 13, no. 4, pp. 189–192, Apr. 2006.
[20] W. Padgett and D. Anderson, *Fixed-Point Signal Processing* (Synthesis Lectures on Signal Processing). Williston, VT, USA: Morgan & Claypool Publ., 2009.
[21] J. J. Nielsen, "Design of linear-phase direct-form FIR digital filters with quantized coefficients using error spectrum shaping techniques," *IEEE Trans. Acoust. Speech Signal Process.*, vol. 37, no. 7, pp. 1020–1026, Jul. 1989.
[22] F. Taylor, *Digital Filters: Principles and Applications With MATLAB*. New York, NY, USA: Wiley, 2012.

[23] B. W. Jung, H. J. Yang, and J. Chun, “Finite wordlength digital filter design using simulated annealing,” in *Proc. 42nd Asilomar Conf. Signals Syst. Comput.*, 2008, pp. 546–550.
[24] J. I. Ababneh and M. H. Bataineh, “Linear phase FIR filter design using particle swarm optimization and genetic algorithms,” *Digit. Signal Process.*, vol. 18, no. 4, pp. 657–668, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1051200407000966
[25] E. Hewitt and R. E. Hewitt, “The Gibbs-Wilbraham phenomenon: An episode in Fourier analysis,” *Archive History Exact Sci.*, vol. 21, no. 2, pp. 129–160, 1979.
[26] “7 Series DSP48E1 slice,” Xilinx Inc., San Jose, CA, USA, Rep. UG479, Mar. 2018.
[27] J. Proakis and D. Manolakis, *Digital Signal Processing*. Hoboken, NJ, USA: Prentice Hall, 2007.
[28] L. Rabiner, B. Gold, and C. McGonegal, “An approach to the approximation problem for nonrecursive digital filters,” *IEEE Trans. Audio Electroacoustics*, vol. 18, no. 2, pp. 83–106, Jun. 1970.
[29] V. Anis, J. Guo, S. Weiss, and L. H. Crockett, “FPGA implementation of a TVWS up- and downconverter using non-power-of-two FFT modulated filter banks,” in *Proc. 27th Eur. Signal Process. Conf. (EUSIPCO)*, 2019, pp. 1–5.
[30] “Model composer and system generator user guide,” Xilinx Inc., San Jose, CA, USA, Rep. UG1483, Nov. 2020.
[31] “Vivado design suite user guide,” Xilinx Inc., San Jose, CA, USA, Rep. UG893, May 2019.
[32] Xilinx Inc., San Jose, CA, USA. “Python Productivity for Zynq (PYNQ) Documentation.” [Online]. Available: https://buildmedia.readthedocs.org/media/pdf/pynq/latest/pynq.pdf (accessed Jan. 16, 2022).
[33] J. Goldsmith. “FIR Filter Design for FPGAs.” [Online]. Available: https://github.com/jogomojo/fpga_filter_design (accessed Jan. 16, 2022).
[34] “Fast Fourier transform v9.1 logiCORE IP product guide,” Xilinx Inc., San Jose, CA, USA, Rep. PG109, Aug. 2021.
[35] “FIR compiler v7.2 LogiCORE IP product guide,” Xilinx Inc., San Jose, CA, USA, Rep. PG149, Jan. 2020.
[36] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, *The Zynq Book: Embedded Processing With the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*. Glasgow, U.K.: Strathclyde Academic, 2014.
[37] “Zynq-7000 SoC Data Sheet: Overview,” Xilinx Inc., San Jose, CA, USA, Rep. DS190, Jul. 2018.
[38] P. Lysaght and C. McCabe, *Exploring Zynq MPSoC: With PYNQ and Machine Learning Applications*, ch. 22. Glasgow, U.K.: Strathclyde Academic, 2019, pp. 525–552.
[39] The MathWorks, Inc., Natick, MA, USA. “FIR2: Frequency Sampling-Based FIR Filter Design.” [Online]. Available: https://www.mathworks.com/help/signal/ref/fir2.html (accessed Jan. 16, 2022).
[40] The SciPy Community. “SciPy v1.7.1 Documentation.” [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.firwin2.html (accessed Jan. 16, 2022).

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. In 2019, he completed a six month internship with Xilinx, Longmont, developing hardware systems and training material for the RFSoC within the PYNQ team. He is a contributing author of the *Exploring Zynq MPSoC*. His research topic is focused on run-time reconfigurable hardware, specifically for FPGA radio applications, and has related interests in signal processing and embedded systems.

LOUISE H. CROCKETT received the master’s and Ph.D. degrees in electronic and electrical engineering from the University of Strathclyde, where he is a Senior Teaching Fellow. She has core research interests in the hardware implementation of Digital Signal Processing systems, in particular for communications and SDR, and is the Principal Author of *The Zynq Book*. Her teaching focuses on digital systems design using hardware description language, Simulink block-based design, and FPGA/SoC technology, and builds practical skills to equip graduates for roles in industry.

ROBERT W. STEWART received the bachelor’s and Ph.D. degrees from the University of Strathclyde.

He is a Professor of Signal Processing with the University of Strathclyde, where he served as the Head of the Department of Electronic and Electrical Engineering from 2014 to 2017. He 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. 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.