RF-DSP: A Reconfigurable and Highly Flexible FPGA-based Digital Signal Processor

Zhiyuan Ma[1], Dong Yuan[2], Hongwei Wang[2], Shujin Cao[1]*

[1] School of Earth Sciences and Spatial Information Engineering, Hunan University of Science and Technology, China
[2] North Information Control Research Academy Group Co., Ltd, China
21010103006@mail.hnust.edu.cn, y-dong@163.com, whw639@126.com, shujin.cao@hnust.edu.cn

Abstract

Digital Signal Processor (DSP) is becoming an increasingly important chip in the digital electronic world, showcasing its capabilities in fields such as video processing, mobile communications, radar systems, and more. Application Specific Integrated Circuit (ASIC) and Programmable Digital Signal Processors (PDSP) continue to be commonly used implementation mechanisms for many digital signal processing applications. However, both of these platforms have their limitations. The non-reconfigurable arithmetic units of ASIC circuits compromise flexibility, making it difficult to support new algorithms. The performance of DSP chips is limited by a serial instruction stream, making it challenging to break through performance bottlenecks. To address these challenges, we have proposed RF-DSP, a reconfigurable and highly flexible digital signal processor based on Field Programmable Gate Arrays (FPGA), aimed at accelerating digital signal processing algorithms. We have customized an instruction set that enables RF-DSP to support various digital signal processing algorithms. RF-DSP features multiple parallel arithmetic units that can be reused to support different algorithms. Additionally, RF-DSP exhibits excellent scalability, allowing for the incorporation of dedicated components and enhancements to the instruction set to support continuously evolving digital signal processing algorithms. We have implemented a hardware prototype on Xilinx Alveo U200 and conducted performance testing on RF-DSP. Compared to DSP chips, RF-DSP achieves acceleration of 1.3-136 times and improves efficiency by 4-400 times. Compared to Xilinx IP, it achieves acceleration of 7-16 times and improves efficiency by 6-14 times.

Keywords: Digital signal processor (DSP), Field Programmable Gate Arrays (FPGA), Instruction set architecture (ISA)

1 Introduction

Over the past two decades, digital signal processing (DSP) has gained widespread application in various fields such as data encryption [1], video processing [2], mobile communications [3], radar systems [4], and more. With advancements in information technology, DSP has experienced rapid development. Digital signal processing involves the use of processors or specialized application-specific integrated circuit (ASIC) to transform, filter, estimate, and enhance signals to obtain the desired signal characteristics. It enables the manipulation and analysis of digital signals to extract valuable information, improve signal quality, and enable advanced signal processing techniques.

The steady development of integrated circuit (IC) technology and design tools has significantly expanded the application areas of DSP. However, ASIC and programmable digital signal processor (PDSP) remain popular implementation mechanisms for many DSP applications. ASICs are designed based on user requirements and specific system needs, aiming to minimize size and power consumption while maximizing performance. However, ASICs are non-programmable and lack flexibility and their development costs are considerably high [5]. They are typically only cost-effective when produced in large quantities and deployed extensively. PDSP are dedicated processors designed using a Harvard architecture, specifically for highspeed real-time processing of digital signals. PDSP can be programmed using high-level languages to implement various signal-processing algorithms. However, their performance is constrained by the serial instruction flow, and they are not reconfigurable at the hardware level [6]. Increasingly, there is a growing interest in exploring new system implementations based on reconfigurable computing. These flexible platforms offer a combination of hardware efficiency and software programmability. FPGA is a programmable device that provides powerful signal-processing capabilities through parallel computing units [7]. Additionally, the hardware circuits generated on FPGAs are reconfigurable, significantly reducing the development costs compared to ASIC and DSP chips. Designing a digital signal processor based on an FPGA can simultaneously enhance performance and provide flexibility.

However, developing a digital signal processor based on an FPGA presents several challenges. These challenges primarily revolve around three aspects:

(1) Support for multiple digital signal processing algorithms: The designed processor needs to

*Corresponding Author: Shujin Cao; Email: shujin.cao@hnust.edu.cn
DOI: https://doi.org/10.70003/160792642026052703001

support a variety of digital signal processing algorithms and adapt to their evolving nature. Different algorithm structures result in distinct memory access patterns and data flows. Ensuring compatibility with multiple algorithms is a critical challenge in processor design.

(2) Unified computational units: Designing computational units that can be reused across different algorithms is a key issue. This implies reducing resource utilization and enhancing system integration. By achieving this, computational resources can be optimized, resulting in improved overall system performance.

(3) Latency and power consumption reduction for higher performance: Leveraging pipeline designs and parallel computing techniques can lead to lower latency. Designers need to explore the potential parallelism within computations and strike a balance between area utilization and power consumption. This entails considering area-power trade-offs and designing parallel strategies that achieve the desired computational goals.

Based on the aforementioned challenges, we propose an FPGA-based digital signal processor called RF-DSP. RF-DSP consists of a customized instruction set, compiler, and several microarchitectural designs. User computation requirements are translated into corresponding instructions by the compiler, which schedule operations within the computation modules to perform various digital signal processing algorithms. We have designed a unified computational unit that supports heterogeneous operations and can be reused for different algorithms, effectively reducing resource utilization and enhancing system integration. By exploiting the inherent parallelism in algorithms, we have devised parallel strategies that improve processor performance. Importantly, RF-DSP can easily be extended to new algorithms by adding new modules and enhancing the instruction set, demonstrating strong scalability.

To summarize, the main contributions of this work are as follows:

- Software-Hardware Co-Design: We have designed RF-DSP, a processor capable of handling multiple digital signal processing algorithms. RF-DSP includes a specific instruction set architecture and microarchitecture optimized for digital signal processing, ensuring efficient computation and data communication.
- User Friendliness: User computation requirements are translated into instructions by the compiler, which schedules the modules within RF-DSP to perform the computations. The compiler provides hardware-friendly optimizations such as data processing and functional fusion to improve computation efficiency.
- Flexible computing unit: We have designed a computing unit capable of supporting heterogeneous operations, which can be utilized during the execution of any algorithm. This effectively reduces resource utilization and enhances system integration.
- Competitive performance and scalability: We have implemented a hardware prototype of RF-DSP on Xilinx Alveo U200 and conducted comprehensive evaluations. RF-DSP demonstrates highly competitive performance, achieving acceleration improvements of 1.3-136 times and energy efficiency improvements of 4-400 times compared to DSP chips. Additionally, compared to Xilinx IP cores, RF-DSP achieves acceleration improvements of 7-16 times and energy efficiency improvements of 6-14 times. RF-DSP exhibits remarkable scalability.

The rest of this article is organized as follows. Section II reviews the background of DSP and outlines our motivations. Section III explains RF-DSP microarchitecture and the compiler Section IV describes the RF-DSP instructions. Section V presents our experiment results. Section VI concludes this article. For ease of understanding, the main symbols and abbreviations used in this paper are summarized in the Abbreviations section.

2 Background and Motivation

2.1 Background

Since the early 1960s, the implementation of digital signal processing has experienced rapid development. As DSP has transitioned from primarily military and scientific applications to numerous low-cost consumer applications, cost has become increasingly important. In the past decade, energy consumption has emerged as a significant metric, given the widespread use of DSP technology in portable systems such as media players and hearing aids. Flexibility has also become a key differentiating factor in DSP implementations, as it allows for changes in system functionality at different stages of the design lifecycle [8]. These cost considerations have led to three main implementation options: ASIC, PDSP, and Reconfigurable Hardware. Each implementation option entails different trade-offs in terms of performance, cost, power, and flexibility.

In Table 1, we have compiled the characteristics of these three solutions. High-level ASIC designs offer advantages in terms of area and power performance. However, their fixed hardware circuits limit their reconfigurability. Furthermore, for small-scale or prototype implementations, it is necessary to evaluate whether the performance improvement provided by ASICs justifies the design costs involved. The initial batch of PDSP was introduced by Texas Instruments. These early processor architectures were primarily based on CISC pipelines and incorporated special architectural features and instructions to support filtering and transformation computations. One of the key changes in the second generation of PDSP was the adjustment to the Harvard architecture, effectively separating the program bus from the data bus.

This optimization breaks through the von Neumann bottleneck, thus designing dedicated channels for data transfer from local memory to the processor pipeline.

However, there are specific limitations when using DSP. Generally, to achieve optimal performance, application programs must be written to utilize the available resources in the DSP, but their performance is limited by the sequential instruction stream. A recent approach to DSP hardware implementation is utilizing reconfigurable computing platforms. Reconfigurable platforms for DSP allow reconfigurability and specialized performance on a per-application basis, offering significant potential at the application level. The direct benefits of reconfigurable methods for DSP can be summarized in three key aspects: functional specialization, platform reconfigurability, and fine-grained parallelism. FPGA is one representative of reconfigurable methods and has been widely researched and experimented with. Common digital signal processing algorithms are presented in Table 2, with the descriptions of the corresponding variables provided in Table 3; much work has been done to implement them on FPGA. For instance, Saeed et al. [9] implemented a compatible FFT/IFFT processor on an FPGA with lower area and latency. These algorithms are crucial in signal processing. Paul et al. [10] migrated different forms of FIR and IIR filters to an FPGA, achieving better performance. Vijay et al. [11] proposed a parallel pipeline based on FIR filters to implement IIR filters, improving the efficiency of IIR filters. Safarian et al. [12] presented an FPGA implementation design of an LMS-based adaptive filter using Xilinx DSP48, achieving improvements in terms of resources, power consumption, and frequency. Kavitha et al. [13] focused on the design of LMS adaptive filter algorithm without multipliers in FPGA devices, reducing area and power consumption.

Table 1. Comparison of the characteristics of different platforms

	Performance	Cost	Power	Flexibility
ASIC	High	High	Low	Low
Programmable DSP	Medium	Medium	Medium	Medium
FPGA	High	Medium	Medium	High

Table 2. Expression of algorithms

Algorithm	Expression
FFT	$X(k)=\sum_{n=0}^{N-1}x(n)W_N^{kn}$
FIR	$y(n)=\sum_{k=0}^{N-1}h(k)x(n-k)$
IIR	$y(n)=\sum_{k=0}^{N}b_k(n-k)-\sum_{k=1}^{N}a_k y(n-k)$
LMS	$y(n)=w^T(n)x(n)$ $e(n)=d(n)-y(n)$ $w(n+1)=w(n)+2\mu e(n)x(n)$

2.2 Motivation

All of the aforementioned works utilize FPGA to implement algorithms in digital signal processing, but they are specific to the designed algorithms. When different algorithms are required in a particular application scenario [14-16], the above solutions would need to redesign RTL circuits and reconfigure (layout and wiring) on the FPGA, making it difficult to directly apply to many digital signal processing applications. In multifunctional scenarios, compatibility with multiple digital signal processing algorithms is usually required. DSP can achieve this by invoking the corresponding computing resources through software programming, but it is limited by the sequential instruction stream and struggles to overcome performance bottlenecks. Although designing high-quality ASIC circuits can also adapt to multi-algorithm scenarios, it sacrifices flexibility and incurs expensive development costs because algorithms are constantly being updated.

While ASIC hardware circuits are fixed and cannot be reconfigured. Utilizing FPGA, a reconfigurable platform, to design a digital signal processor compatible with multiple algorithms is a reliable solution. FPGA has sufficient parallel computing resources and can be reconfigured to maintain comprehensiveness when supporting new algorithms, perhaps by only modifying a small portion of the design. In summary, FPGA can maintain its flexibility at a small cost and is suitable for multi-algorithm application scenarios. However, designing an FPGA-based digital signal processor poses many challenges. Key issues include scheduling data when switching algorithms, considering power consumption and area in parallel computations, and designing computational units for reusability when implementing different algorithms.

3 Architecture Overview

3.1 Overview of RF-DSP

Our work primarily targets various applications and systems that require digital signal processing, such as communication systems, audio processing, video processing, image processing, radar, and signal processing, among others. Specific applications and systems often need diverse requirements for data processing. Therefore, the selection of filter types and algorithms necessitates a comprehensive evaluation in accordance with the specific context. To facilitate this, we have deployed several mainstream filters and algorithms on FPGA and designed a dedicated compiler on the software side. This intricate arrangement ensures users possess to flexibly schedule and utilize the functionalities deployed in the hardware.

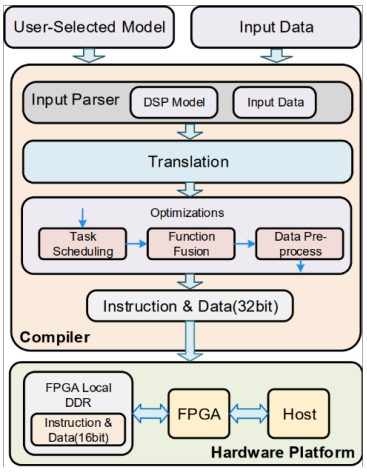


Figure 1. Overview of RF-DSP

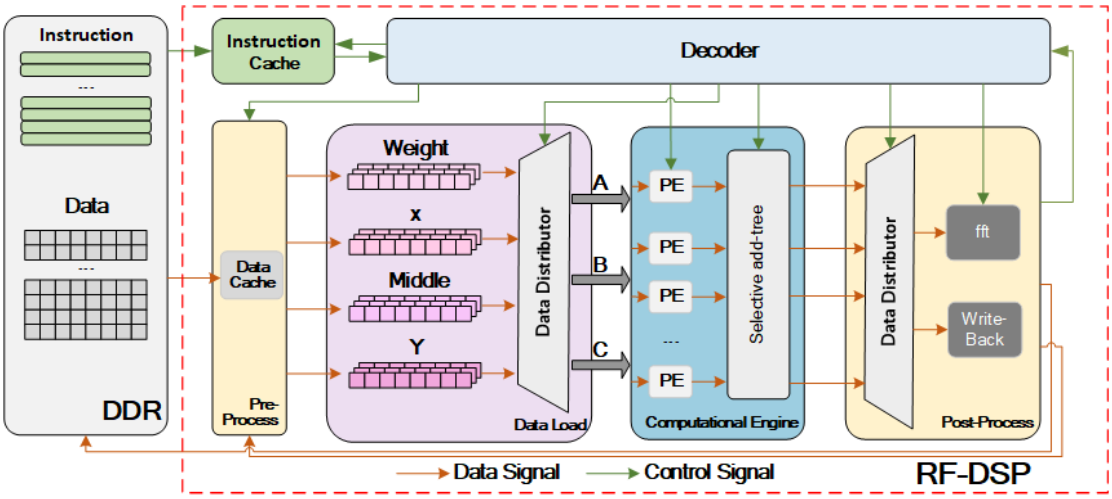


Figure 2. The microarchitecture of RF-DSP

The overall architecture, as illustrated in Figure 1. The hardware platform consists of an FPGA device and FPGA local DDR memory. The proposed hardware accelerator is deployed on the FPGA device. FPGA local DDR memory stores the input images, audio signals, and other data awaiting processing and binary files generated by the compiler. After receiving user-provided parameters and the data to be processed, the compiler initially translates them into corresponding instructions. Subsequently, it progresses through three optimization steps (Task Scheduling, Function Fusion, Data Pre-process) to generate an optimized instruction sequence, which is subsequently executed on the hardware processor.

3.2 Compiler

Compilers are essential tools that bridge the gap between user requirements and underlying hardware computations. In the popular field of deep learning, the use of custom domain-specific compilers to alleviate the burden of model optimization has become a popular approach, avoiding some limitations of deep learning libraries and tools [17]. Several popular deep learning compilers have emerged rapidly, including TVM [18], Tensor Comprehension [19], Glow [20], nGraph [21], and XLA [22]. These compilers are designed to target model specifications and hardware architectures, with highly optimized transformations between model definitions and concrete implementations, significantly reducing the manual burden of optimizing models. Drawing inspiration from these works, we have developed a specialized compiler to alleviate the overhead of algorithm execution. The compiler of RF-DSP derives the appropriate instructions by parsing the parameters and data provided by the user. Additionally, in order to enhance the efficiency of instruction and data handling by the hardware and to optimize the utilization of hardware resources, the compiler performs optimization on both instructions and data subsequent to the translation of user-provided parameters into their corresponding instructions. The compiler performs three-step optimizations and generates the output instruction sequence.

1) *Task Scheduling*: For algorithms similar to FIR, the data processing flow they follow does not have dependencies on the previously completed operations. Hence, when executing such operations, the compiler compares the resources required for the current task with the hardware resources deployed. If there are available hardware resources while executing user tasks, the compiler generates an additional 16-bit instruction to efficiently schedule the remaining resources and expedite the task progress.

2) *Function Fusion*: To meet the requirement of allowing users to simultaneously execute two different filtering or algorithm operations, the compiler concatenates the corresponding 16-bit instructions for both operations into a single 32-bit instruction, provided that the user-defined parameters are within reasonable limits and fall within the hardware resource constraints. This enables the hardware to execute both computations concurrently. However, if the user-defined parameters exceed the hardware resource limits, an error report is returned, prompting the user to reset the parameters within the allowable range.

3) *Data Pre-process*: During the execution of the aforementioned first and second operations, as well as specific algorithms such as LMS and FFT, the hardware necessitates the input of two distinct sets of data to perform the calculations. For instance, FFT requires both real and imaginary data to be provided for each computation. In cases where these two data sets cannot be simultaneously inputted, significant resources and time would be wasted by the hardware in filtering and selecting the required data for different algorithms or steps. To streamline the data filtering and processing process within the hardware, the compiler identifies these scenarios and concatenates the two 16-bit data sets into a unified 32-bit data format. This combined data, along with its corresponding instruction, is then transmitted to the hardware to facilitate the seamless execution of the computations.

3.3 Microarchitecture

For the RF-DSP, the greatest challenge is the overlay

microarchitecture design. The overlay microarchitecture needs to incur as less control overhead as possible while maintaining easy runtime adjustable and functionality. To meet these requirements, we design our modules to be parameter customizable and switch modes at runtime based on parameter registers that directly accept parameters provided by instructions, so that it enables seamless switching between different data paths to accomplish diverse operations. As shown in Figure 2, the microarchitecture of RF-DSP primarily comprises six modules: Instruction Cache, decoder, pre-process, data load, computational engine, and post-process.

1) *Instruction Cache and Decoder*: The Instruction-cache module retrieves the 32-bit instructions generated by the compiler from the DDR and stores them in the instruction cache module. Upon fetching these instructions from the Instruction cache, the decoder module first splits them into 16-bit instructions and then decodes them, generating control signals that are passed to their respective control modules. Additionally, after receiving the first instruction, the decoder module relies on control signals provided by the post-processing module to determine the timing for reading each subsequent instruction.

2) *Pre-process and Data Cache*: Due to the varying data requirements of different computations, the pre-process module retrieves the corresponding quantity of input data x, computation results y, and intermediate data middle from DDR or the post-process module, storing them in the data cache based on the control instructions sent by the decoder. It then determines the data type of the current data to be processed and splits it accordingly, sending it to the corresponding data array within the data-load module. The data-load module consists of four data arrays: the Weight data array, which primarily stores filter coefficients and rotation factors for Fourier transforms; the X data array, which stores input data x; the Middle data array, which stores intermediate data middle or input data x; and the Y data array, which stores the computed results. Let's take a 5th-order FIR filter and a 5×5 matrix multiplication as examples to illustrate how the input data is stored in the data arrays through the pre-process module.

For FIR filters, according to the formula in Table 2, it can be observed that each computation necessitates the update of only one input data value. Therefore, in each computation, the pre-process module places the input value x in the first position of the X data array, shifting the other positions downward. In the case of matrix multiplication, each computation requires updating the number of input values corresponding to the number of columns. Hence, in each computation, the preprocess module splits the required input values x from the data cache into row elements and column elements and then stores them sequentially in the X data array and *Middle* data array, respectively.

The data dispatcher in the data-load module routes different data arrays to their corresponding data channels based on instructions. Each data channel assigns members from the data arrays to the PE array for parallel computation. For example, during FFT calculations, the real part of the input data x is stored in the X data array and sent to Channel A, while the imaginary part is stored in the Middle data array and sent to Channel B. The rotation factors from the weight data array are stored in Channel C. When performing computations that require only two data arrays, the remaining idle channel will be filled with either 0 or 1. Further details will be discussed in the next subsection.

3) *Computational Engine*: The Computational Engine module comprises a PE array and an adder tree selection unit. In order to cater to various algorithms covered by the system, a comprehensive analysis of their common characteristics was conducted, leading to the specialized design of our PE units. As shown in Figure 3, each PE unit within the PE array receives inputs from three data channels: A, B, and C. These units are equipped with an adder, a subtractor, a multiplier, and an output selection.

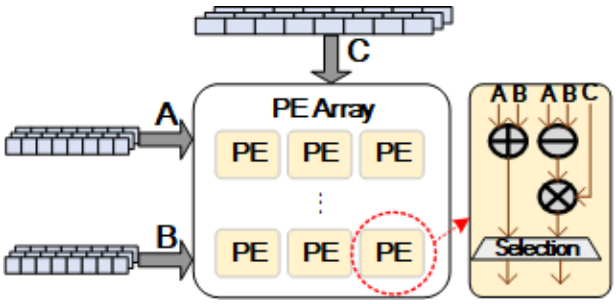


Figure 3. The architecture of PE

During computation, the C channel is filled with ones when a PE unit performs addition or subtraction operations, while the B channel is filled with zeros during multiplication operations. Each PE unit operates independently, enabling the completion of a set of computations for a specific group of numbers. Consequently, the system's level of parallelism is solely determined by the number of PE units present in the PE array.

After the computation is completed in the PE, the output selector determines the current operation type based on instructions and sends the corresponding data to the Selective Add-Tree module. The Selective Add-Tree module, upon receiving the data, evaluates the instruction to determine whether an addition operation is required. If not, it directly outputs the data. Otherwise, the data enters the adder tree for further computation.

To accommodate filters of different orders and matrix operations of varying sizes, we have designed an adder tree that can dynamically adjust its levels based on instructions. We have placed a selector before and after each adder in the tree, allowing it to determine whether the output should be generated at that level or passed to the next level, based on the required number of levels specified in the instruction. For instance, when performing the summation of a group of 8 numbers or the summation of two groups of 4 numbers, both calculations can be achieved using an adder tree with 3 levels. The difference lies in whether the data should be output as a summation result at the last level of the adder tree or at the second-to-last level, producing two summation results.

4) *Post-process*: The post-process module is responsible for handling the computed data from the Computational Engine module. Due to the varying overall flow between different algorithms, the data outputted by the Computational Engine can be the final result, feedback value, or weight update value. To accommodate this variability, a data distributor is designed within the post-process module. It utilizes instructions to determine the nature of the computed data, the operation it belongs to, and the step within that operation, which enables the determination of its subsequent flow.

During FFT (Fast Fourier Transform) operations, the data is allocated to a dedicated post-processing module for FFT. As FFT involves multiple rounds of computation, each subsequent round requires the results of the previous round as input. However, the order of the data gets rearranged after each round, making it unsuitable for direct use. Therefore, upon receiving the data, this module determines the current computation round based on the instructions and generates a corresponding address for each data point. These addresses are then transmitted back to the pre-process module, which utilizes the received addresses to store the data in the appropriate locations within the data cache. For other computations, the data is allocated to the writeback module. This module determines the current computation flow based on the instructions. If the computation flow has reached the final stage, the data is outputted to the DDR (Double Data Rate) memory and transmitted back to the host through PCIe (Peripheral Component Interconnect Express). Otherwise, the data is returned to the data cache and awaits the next round of pre-process instructions to be read into the corresponding data groups.

Furthermore, to ensure the proper alignment of each instruction with its corresponding data, the post-process module generates a control signal alongside each data output. This control signal is transmitted back to the decoder module, indicating the completion of the current instruction execution and enabling the decoder to read the next instruction. It is worth mentioning that RF-DSP exhibits strong scalability by enabling compatibility with new digital signal processing algorithms through the addition of new components in the post-process module.

4 Instruction Set Architecture

Another challenging aspect of the design is the instruction set, as it controls the data execution paths in the RF-DSP. The diversity of data paths is essential for accommodating the requirements of various algorithms. We have been inspired by some neural network processor work [23-24]. Therefore, we conducted an analysis of several algorithms that the DSP needs to cover and categorized the steps of these algorithms to design a concise and efficient instruction set that satisfies their needs. In the following subsection, we will use a complete LMS (Least Mean Squares) operation as an example to provide a detailed explanation of our instructions, focusing on their composition and their execution within the hardware.

The composition of each instruction part and the significance represented by each instruction part are shown in Figure 4(a) and the execution flow of instructions in various hardware modules is depicted in Figure 4(b). To facilitate computation by the PE module, we first rewrite the equation in Table 2 into the following equations.

$$y(n) = w^T(n)x(n) \tag{1}$$

$$e(n) = 2\mu(d(n) - y(n)) \tag{2}$$

$$k(n) = e(n)x(n) \tag{3}$$

$$w(n+1) = w(n) + k(n) \tag{4}$$

During the execution of Equation (1), the decoder module examines the instruction opcode and sel-func to determine that the current operation corresponds to the first step of the LMS algorithm. As a result, the input $x(n)$ and input $d(n)$ are stored in the X and middle data arrays, respectively. The A and C channels in the PE module are enabled, while the B channel is filled with 0. The add tree is activated, with x being routed to the A channel and the coefficient w being routed to the C channel. Subsequently, the computed result is transmitted back to the Y data array through control signals from the post-process module.

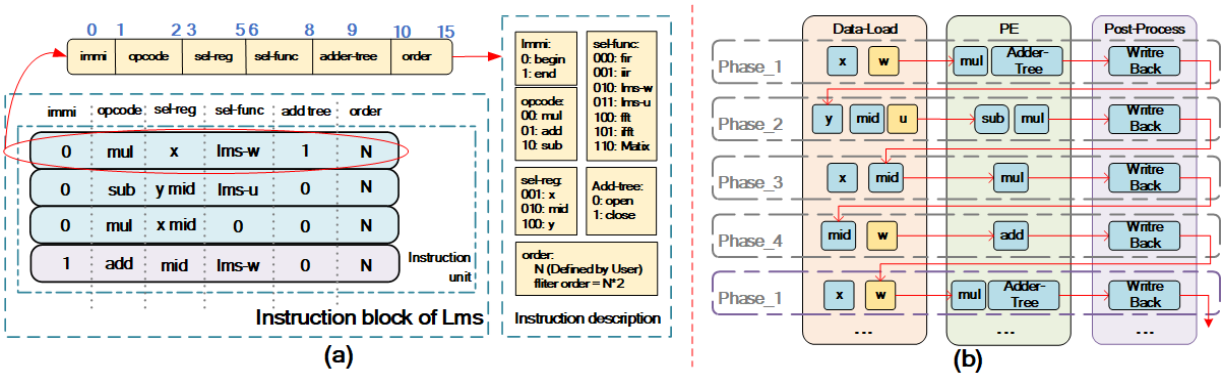


Figure 4. Instruction set of LMS: (a) Instruction description; (b) Instruction execution in hardware

Similarly, during the execution of Equation (2), the A, B, and C channels in the PE module are enabled, the add tree is disabled, the mid data array is routed to the A channel, the y data array is routed to the B channel, and the coefficient 2μ is routed to the C channel. The result is then transmitted back to the middle data array.

In the case of Equation (3), the PE channels are configured in the same manner as in Equation (1), with x data array being routed to the A channel and the middle data array being routed to the C channel. The computed result k(n) is transmitted back to the middle data array.

Lastly, during the execution of Equation (4), the A and B channels in the PE module are enabled, while the C channel is filled with 1. The middle data array is routed to the A channel, and the coefficient w is routed to the B channel. If the current LMS calculation is complete, the immi field in the instruction is set to 1, and the DSP module concludes the current operation and outputs the result, as shown in Figure 4(a). Otherwise, the post-process module transmits the updated coefficient w back to the w data array to continue the next round of calculations, as illustrated in Figure 4(b).

5 Experiment

5.1 Experimental Setup

The proposed RF-DSP is implemented on the Alveo U200 FPGA. We encode our design using Verilog HDL and synthesis it by Vivado 2020.1. For our compiler, we design and test it in C++. Our experimental evaluation focuses on resource utilization, including LUTs, BRAM, and DSPs, as well as performance metrics such as latency and throughput.

5.2 Latency and Resource Consumption

In Table 4, it can be observed that we utilized only a minimal amount of resources. This resource utilization was obtained with the use of 96 PE units. In Table 5, we further measure the actual results on different platforms. We compare the frequency (MHz), latency (μs), throughput (Ms/s), power (W), LUTs, FFs, DSPs, and BRAMs for the four algorithms: FFT, LMS, IIR, and FIR.

In the evaluation results, we employed a 1024-point radix-2 FFT as the reference benchmark. The filter algorithms used were a 32nd-order LMS, 5th-order IIR, and 32nd-order FIR filters. We conducted performance testing on the C6678 chip for these four algorithms. Additionally, we performed tests on the algorithms supported by Xilinx IP. The latency results are obtained by calculating the time required for the first sample to be inputted until the last sample is outputted. When deployed at the edge or with the utilization of more PE units, our architecture exhibits lower power consumption and higher energy efficiency ratio. This is attributed to the static power consumption of 2.4W of the U200 FPGA we utilize.

In Table 5, we used the DSP TMS320C6678 as a baseline for performance comparison. It can be observed that in terms of FFT performance, our architecture achieves a ten-fold increase in throughput compared to the C6678. Compared to Xilinx IP, although we utilize approximately eight times more resources than the IP, our throughput is sixteen times higher. Compared to Pakize's architecture [25] deployed on the same FPGA, although our throughput is comparable, we utilize fewer DSP resources. Moreover, it is crucial to note that, unlike the designs by Jovanovic [15] and Pakize [25], our design accommodates a generic DSP algorithm that can be seamlessly switched. Previous designs were limited to covering specific algorithms, resulting in reduced flexibility. In terms of filter performance, with the same filter order, our architecture achieves a throughput increase of 1.3-136 times compared to the C6678, while consuming less than one-third of its power. This advantage becomes even more significant at the edge or with the utilization of more PE units.

Table 3. The descriptions for DSP algorithms

FFT	$X(k)$: The k-th component of the frequency domain signal.
	N: The length of the signal or number of sample points.
	$x(n)$: The n-th sample of the time domain signal.
	W_N: Complex weighting factor, typically defined as $e^{-j\frac{2n}{N}}$
FIR	$y(n)$: The n-th sample of the filter output.
	$h(n)$: The impulse response of the filter, i.e., the filter coefficients.
	$x(n-k)$: The $(n-k)$ th sample of the input signal, representing a delay.
IIR	$y(n)$: The n-th sample of the filter output.
	b_k: The forward (or feedback) coefficients of the filter.
	a_k: The reverse (or feedthrough) coefficients of the filter.
	$y(n-k)$: The delayed version of the $(n-k)$ th sample of the output signal.
LMS	$y(n)$: The n-th sample of the filter output or estimate.
	w: Weight vector or filter coefficients.
	$e(n)$: Error signal, which is the difference between the desired output $d(n)$ and the actual output $y(n)$.
	$d(n)$: The desired or reference output signal.
	μ: The convergence factor or step size of the algorithm, which controls the rate of weight update.

Table 4. The resource utilization of our RF-DSP

Resource	Usage	Available	Utilization (%)
LUT	23560	1182240	1.99
FF	18321	2364480	0.77
BRAM	10.5	2160	0.49
DSP	96	6840	1.4

5.3 Efficiency Comparison

In Figure 5, we compare the efficiency of the RF-DSP architecture with the C6678 chip. The efficiency is calculated using the following formula:

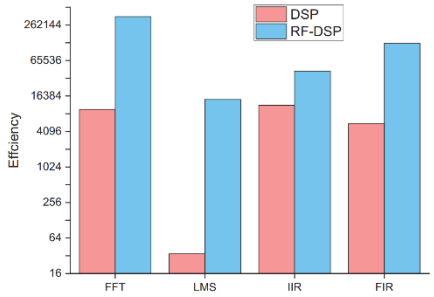


Figure 5. Comparison of efficiency

$$\text{Efficiency} = 1/(\text{Latency} \times \text{Power}) \quad (5)$$

It is worth noting that we have used logarithmic scale in the graph. Therefore, the actual differences are larger than what is seen in the graph. The efficiency of our architecture is 4-400 times greater than that of C6678, making it more suitable for deployment at the edge.

6 Conclusion

In this paper, we propose RF-DSP, an FPGA-based processor for implementing DSP algorithms. Firstly, we present a hardware architecture for implementing DSP algorithms. A unified computational unit is introduced, which can be reused across different algorithms. Secondly, we design an instruction set for our architecture, ensuring the flexibility of RF-DSP. Compared to C6678, RF-DSP achieves acceleration of 1.3-136 times and improves efficiency by 4-400 times. Compared to Xilinx IP, it achieves acceleration of 7-16 times and improves efficiency by 6-14 times. In the future, we plan to support additional DSP algorithms and enhance our performance. Concurrently, we are considering diversifying our deployments, such as accommodating FPGAs with higher memory bandwidth like HBM FPGAs, and typical edge FPGAs, such as Zynq and others.

Abbreviations

ASIC	Application Specific Integrated Circuit
CISC	Complex instruction set computer
DDR	Double Data Rate Synchronous Dynamic Random-Access Memory
DSP	Digital Signal Processor
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
IC	Integrated Circuit
IFFT	Inverse Fast Fourier Transform
IIR	Infinite Impulse Response
LMS	Least Mean Squares
LUT	Look-up Table
PCIe	Peripheral Component Interconnect Express
PDSP	Programmable Digital Signal Processor
PE	Processing Engine
RTL	Register-transfer level
TVM	Tensor Virtual Machine

Acknowledgements

This research was funded by the National Natural Science Foundation of China under Grant 41704138, Grant 41974148, and in part by the Hunan Provincial Science & Technology Department of China under Grant 2017JJ3069, and in part by the Project of Doctoral Foundation of Hunan University of Science and Technology under Grant E51651, and in part by the Hunan Provincial Key Laboratory of Shale Gas Resource Exploitation under Grant E21722.

Table 5. Comparisons of performance on different platforms

Designs	Platform	Function	Order	Point	Frequency (MHz)	Latency (μs)	Throughput (MS/s)	Power (W)	LUT	FF	DSP	BRAM
RF-DSP	Xilinx Alveo U200	**FFT**	\	**1024**	**225**	**0.834**	**1227.82**	**3.3**	**23560**	**18321**	**96**	**10.6**
		LMS	**32**	**1024**		**21.186**	**48.33**					
		IIR	**5**	**1024**		**7.062**	**145.00**					
		FIR	**32**	**1024**		**2.372**	**431.70**					
DSP	TMS320C6678	FFT	\	1024	1000	10.287	99.54	10	\	\	\	\
		LMS	32	1024		2899.013	0.35					
		IIR	5	1024		8.819	116.11					
		FIR	32	1024		18.012	56.85					
Xilinx IP	Xilinx Alveo U200	FFT	\	1024	231	13.838	74.00	2.754	2002	3477	16	4.5
		FIR	32	1024	170	18.071	56.67	2.492	159	660	6	0
Pakize (2021) [25]	Virtex-7 XC7VS330T	FFT	\	1024	339	0.812	1261.00	1.6	12.5K	22.5K	150	6
Jovanovic (2022) [15]	Cyclone V	FIR	7	\	122.88	/	/	/	2322	2445	14	3

References

[1] X. Li, J. Mou, S. Banerjee, Z. Wang, Y. Cao, Design and DSP implementation of a fractional-order detuned laser hyperchaotic circuit with applications in image encryption, *Chaos, Solitons & Fractals*, Vol. 159, Article No. 112133, June, 2022.
https://doi.org/10.1016/j.chaos.2022.112133

[2] M. Budagavi, W. R. Heinzelman, J. Webb, R. Talluri, Wireless MPEG-4 video communication on DSP chips, *IEEE Signal Processing Magazine*, Vol. 17, No. 1, pp. 36-53, January, 2000.
https://doi.org/10.1109/79.814645

[3] X. Liu, H. Zeng, N. Chand, F. Effenberger, Efficient Mobile Fronthaul via DSP-Based Channel Aggregation, *Journal of Lightwave Technology*, Vol. 34, No. 6, pp. 1556-1564, March, 2016.
https://doi.org/10.1109/JLT.2015.2508451

[4] C. Garripoli, M. Mercuri, P. Karsmakers, P. J. Soh, G. Crupi, G. A. E. Vandenbosch, C. Pace, P. Leroux, D. Schreurs, Embedded DSP-Based Telehealth Radar System for Remote In-Door Fall Detection, *IEEE Journal of Biomedical and Health Informatics*, Vol. 19, No. 1, pp. 92-101, January, 2015.
https://doi.org/10.1109/JBHI.2014.2361252

[5] S. Shao, P. Hailes, T.-Y. Wang, J.-Y. Wu, R. G. Maunder, B. M. Al-Hashimi, L. Hanzo, Survey of Turbo, LDPC, and Polar Decoder ASIC Implementations, *IEEE Communications Surveys & Tutorials*, Vol. 21, No. 3, pp. 2309-2333, third quarter, 2019.
https://doi.org/10.1109/COMST.2019.2893851

[6] G. Frantz, Digital signal processor trends, *IEEE Micro*, Vol. 20, No. 6, pp. 52-59, November-December, 2000.
https://doi.org/10.1109/40.888703

[7] M. Ling, Q. Lin, R. Chen, H. Qi, M. Lin, Y. Zhu, J. Wu, Vina-FPGA: A Hardware-Accelerated Molecular Docking Tool With Fixed-Point Quantization and Low-Level Parallelism, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 31, No. 4, pp. 484-497, April, 2023.
https://doi.org/10.1109/TVLSI.2022.3217275

[8] R. Tessier, W. Burleson, Reconfigurable Computing for Digital Signal Processing: A Survey, *The Journal of VLSI Signal Processing-Systems for Signal, Image, and Video Technology*, Vol. 28, No. 1-2, pp. 7-27, May, 2001.
https://doi.org/10.1023/A:1008155020711

[9] A. Saeed, M. Elbably, G. Abdelfadeel, M. I. Eladawy, Efficient FPGA implementation of FFT/IFFT Processor, *International Journal of Circuits, Systems and Signal Processing*, Vol. 3, No. 3, pp. 103-110, July, 2009.
https://www.naun.org/main/NAUN/circuitssystemssignal/19-102.pdf

[10] A. Paul, T. Z. Khan, P. Podder, Md. M. Hasan, T. Ahmed, Reconfigurable architecture design of FIR and IIR in FPGA, *2015 2nd International Conference on Signal Processing and Integrated Networks (SPIN)*, Noida, India, 2015, pp. 958-963.
https://doi.org/10.1109/SPIN.2015.7095408

[11] V. Vijay, V. R. S. Rao, K. Chaitanya, S. C. Venkateshwarlu, C. S. Pittala, R. R. Vallabhuni, High-Performance IIR Filter Implementation Using FPGA, *2021 4th International Conference on Recent Trends in Computer Science and Technology (ICRTCST)*, Jamshedpur, India, 2022, pp. 354-358.
https://doi.org/10.1109/ICRTCST54752.2022.9781944

[12] C. Safarian, T. Ogunfunmi, W. J. Kozacky, B. K. Mohanty, FPGA implementation of LMS-based FIR adaptive filter for real time digital signal processing applications, *2015 IEEE International Conference on Digital Signal Processing (DSP)*, Singapore, 2015, pp. 1251-1255.
https://doi.org/10.1109/ICDSP.2015.7252081

[13] S. Kavitha, G. Sinduja, M. Srimathi, K. Yogalakshmi, An Efficient FPGA Implementation of the Multiplier-less LMS Adaptive Filter, *2023 7th International Conference on Computing Methodologies and Communication (ICCMC)*, Erode, India, 2023, pp. 441-445.
https://doi.org/10.1109/ICCMC56507.2023.10084108

[14] T.-T. Lee, H.-H. Chiang, J.-W. Perng, J.-J. Jiang, B.-F. Wu, Multi-sensor information integration on DSP platform for vehicle navigation safety and driving aid, *2009 International Conference on Networking, Sensing and Control*, Okayama, Japan, 2009, pp. 653–658.
https://doi.org/10.1109/ICNSC.2009.4919354

[15] B. Jovanovic, S. Milenkovic, Transmitter IQ imbalance mitigation and PA linearization in software defined radios, *Radioengineering*, Vol. 31, No. 1, pp. 144-154, April, 2022.
https://doi.org/10.13164/re.2022.0144

[16] J. Kubak, J. Stastny, P. Sovka, An embedded implementation of discrete zolotarev transform using hardware-software codesign, *Radioengineering*, Vol. 30, No. 2, pp. 364-371, June, 2021.
https://doi.org/10.13164/re.2021.0364

[17] M. Li, Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, D. Qian, The Deep Learning Compiler: A Comprehensive Survey, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 32, No. 3, pp. 708-727, March, 2021.
https://doi.org/10.1109/TPDS.2020.3030548

[18] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu , L. Ceze, C. Guestrin, A. Krishnamurthy, TVM: An Automated End-to-End Optimizing Compiler for Deep Learning, *13th USENIX Conference on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, USA, 2018, pp. 579-594.
https://dl.acm.org/doi/10.5555/3291168.3291211

[19] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, A. Cohen, *Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions*, arXiv, pp. 1-37, June, 2018.
https://doi.org/10.48550/arXiv.1802.04730

[20] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, J. Montgomery, B. Maher, S. Nadathur, J. Olesen, J. Park, A. Rakhov, M. Smelyanskiy, M. Wang, *Glow: Graph lowering compiler techniques for neural networks*, arXiv, pp. 1-12, April, 2019.
https://doi.org/10.48550/arXiv.1805.00907

[21] S. Cyphers, A. K. Bansal, A. Bhiwandiwalla, J. Bobba, M. Brookhart, A. Chakraborty, W. Constable, C. Convey, L. Cook, O. Kanawi, R. Kimball, J. Knight, N. Korovaiko, V. Kumar, Y. Lao, C. R. Lishka, J. Menon, J. Myers, S. A. Narayana, A. Procter, T. J. Webb, Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning, *The Conference on Systems and Machine Learning (SysML)*, Stanford, CA, USA, 2018, pp. 1-3.
https://mlsys.org/Conferences/doc/2018/132.pdf

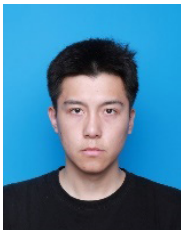
[22] R. Frostig, M. J. Johnson, C. Leary, Compiling machine learning programs via high-level tracing, *The Conference on Systems and Machine Learning (SysML)*, Stanford, CA, USA, 2018, pp. 1-3. https://mlsys.org/Conferences/doc/2018/146.pdf

[23] R. Chen, H. Zhang, Y. Ma, E. Tang, S. Li, Y. Zhu, J. Yu, K. Wang, Graph-OPU: An FPGA-Based Overlay Processor for Graph Neural Networks, *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, USA, 2023, p. 49. https://doi.org/10.1145/3543622.3573152

[24] R. Chen, H. Zhang, S. Li, E. Tang, J. Yu, K. Wang, Graph-OPU: A Highly Integrated FPGA-Based Overlay Processor for Graph Neural Networks, *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*, Gothenburg, Sweden, 2023, pp. 228-234. https://doi.org/10.1109/FPL60245.2023.00039

[25] P. Ergül, H. F. Uğurdağ, D. Davutoğlu, HC-FFT: highly configurable and efficient FFT implementation on FPGA, *Turkish Journal of Electrical Engineering and Computer Sciences*, Vol. 29, No. 7, pp. 3150-3164. January, 2021. https://doi.org/10.3906/elk-2101-56

Biographies



Zhiyuan Ma received a bachelor's degree from Chengdu University of Technology and a master's degree from Hunan University of Science and Technology in 2016 and 2024, respectively. His current research interests include computer architecture, fine processing and inversion of gravity and magnetic data, digital signal processing, and terahertz imaging.

Dong Yuan received the M.A. degree from Southeast University in 2017. His current research interests include vehicle information detection system and video image processing circuit design.

Hongwei Wang received the B.S. degree from Jiangsu University in 2006. His current research interests include embedded system software technology, video image processing, and servo control systems.

Shujin Cao obtained his Bachelor's degree, Master's degree and PhD from the Central South University in 2006, 2009, and 2015, respectively. His research interests include fine processing and inversion of gravity and magnetic data, joint inversion of seismic and gravity data, and their applications in deep structure and tectonic research.